z/OS
2.5

TSO/E REXX Reference

SA32-0972-50

Note

Before using this information and the product it supports, read the information in "Notices" on page 473.

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

Last updated: 2021-09-30

Contents

Figures

Tables

About this document

Who should read this document

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

This document describes the z/OS TSO/E REXX Interpreter (hereafter referred to as the interpreter or language processor) and the REstructured eXtended eXecutor (called REXX) language. Together, the language processor and the REXX language are known as TSO/E REXX. This document is intended for experienced programmers, particularly those who have used a block-structured, high-level language (for example, PL/I, Algol, or Pascal).

TSO/E REXX is the implementation of SAA REXX on the MVS™ system. Although TSO/E provides support for REXX, you can run REXX programs (called REXX execs) in any MVS address space. That is, you can run a REXX exec in TSO/E and non-TSO/E address spaces.

Descriptions include the use and syntax of the language and explain how the language processor "interprets" the language as a program is running. The document also describes TSO/E external functions and REXX commands you can use in a REXX exec, programming services that let you interface with REXX and the language processor, and customizing services that let you customize REXX processing and how the language processor accesses and uses system services, such as storage and I/O requests.

How to use this document

This document is a reference rather than a tutorial. It assumes you are already familiar with REXX programming concepts. The material in this document is arranged in chapters:

- Chapter 1, “Introduction,” on page 1
- Chapter 2, “REXX general concepts,” on page 5
- Chapter 3, “Keyword instructions,” on page 41 (in alphabetic order)
- Chapter 4, “Functions,” on page 73 (in alphabetic order)
- Chapter 5, “Parsing,” on page 157 (a method of dividing character strings, such as commands)
- Chapter 6, “Numbers and arithmetic,” on page 171
- Chapter 7, “Conditions and condition traps,” on page 181
- Chapter 8, “Using REXX in different address spaces,” on page 187
- Chapter 9, “Reserved keywords, special variables, and command names,” on page 197
- Chapter 10, “TSO/E REXX commands,” on page 201
- Chapter 11, “Debug aids,” on page 233
- Chapter 12, “TSO/E REXX programming services,” on page 239
- Chapter 13, “TSO/E REXX customizing services,” on page 305
- Chapter 14, “Language processor environments,” on page 311
- Chapter 15, “Initialization and termination routines,” on page 371
- Chapter 16, “Replaceable routines and exits,” on page 389

There are several appendixes covering:

- Appendix A, “Double-byte character set (DBCS) support,” on page 429
- Appendix B, “IRXTERMA routine,” on page 445
- Appendix C, “Writing REXX Execs to perform MVS operator activities,” on page 449
- Appendix D, “Additional variables that GETMSG sets,” on page 455

- Appendix E, “REXX symbol and hexadecimal code cross-reference,” on page 463
- Appendix F, “Accessibility,” on page 469

This introduction and Chapter 2, “REXX general concepts,” on page 5 provide general information about the REXX programming language. The two chapters provide an introduction to TSO/E REXX and describe the structure and syntax of the REXX language; the different types of clauses and instructions; the use of expressions, operators, assignments, and symbols; and issuing commands from a REXX program.

Other chapters in the document provide reference information about the syntax of the keyword instructions and built-in functions in the REXX language, and the external functions TSO/E provides for REXX programming. The keyword instructions, built-in functions, and TSO/E external functions are described in Chapter 3, “Keyword instructions,” on page 41 and Chapter 4, “Functions,” on page 73.

Other chapters provide information to help you use the different features of REXX and debug any problems in your REXX programs. These chapters include:

- Chapter 5, “Parsing,” on page 157
- Chapter 6, “Numbers and arithmetic,” on page 171
- Chapter 7, “Conditions and condition traps,” on page 181
- Chapter 9, “Reserved keywords, special variables, and command names,” on page 197
- Chapter 11, “Debug aids,” on page 233

TSO/E provides several REXX commands you can use for REXX processing. The syntax of these commands is described in Chapter 10, “TSO/E REXX commands,” on page 201.

Although TSO/E provides support for the REXX language, you can run REXX execs in any MVS address space (TSO/E and non-TSO/E). Chapter 8, “Using REXX in different address spaces,” on page 187 describes various aspects of using REXX in TSO/E and non-TSO/E address spaces and any restrictions.

In addition to REXX language support, TSO/E provides programming services you can use to interface with REXX and the language processor, and customizing services that let you customize REXX processing and how the language processor accesses and uses system services, such as I/O and storage. The programming services are described in Chapter 12, “TSO/E REXX programming services,” on page 239. The customizing services are introduced in Chapter 13, “TSO/E REXX customizing services,” on page 305 and are described in more detail in the following chapters:

- Chapter 14, “Language processor environments,” on page 311
- Chapter 15, “Initialization and termination routines,” on page 371
- Chapter 16, “Replaceable routines and exits,” on page 389

Throughout the document, examples that include data set names are provided. When an example includes a data set name that is not enclosed in either single quotation marks or double quotation marks, the prefix is added to the beginning of the data set name to form the final, fully qualified data set name. If the data set name is enclosed within quotation marks (single or double), it is considered to be fully qualified, and the data set name is used exactly as specified. In the examples, the user ID is the prefix.

How to read syntax diagrams

This section describes how to read syntax diagrams. It defines syntax diagram symbols, items that might be contained within the diagrams (keywords, variables, delimiters, operators, fragment references, operands) and provides syntax examples that contain these items.

Syntax diagrams pictorially display the order and parts (options and arguments) that comprise a command statement. They are read from left to right and from top to bottom, following the main path of the horizontal line.

Symbols

The following symbols can be displayed in syntax diagrams:

Symbol
Definition

►►——
Indicates the beginning of the syntax diagram.

——►
Indicates that the syntax diagram is continued to the next line.

►——
Indicates that the syntax is continued from the previous line.

——►◄
Indicates the end of the syntax diagram.

Syntax items

Syntax diagrams contain many different items. Syntax items include:

- Keywords - a command name or any other literal information.
- Variables - variables are italicized, appear in lowercase and represent the name of values you can supply.
- Delimiters - delimiters indicate the start or end of keywords, variables, or operators. For example, a left parenthesis is a delimiter.
- Operators - operators include add (+), subtract (-), multiply (*), divide (/), equal (=), and other mathematical operations that might need to be performed.
- Fragment references - a part of a syntax diagram, separated from the diagram to show greater detail.
- Separators - a separator separates keywords, variables or operators. For example, a comma (,) is a separator.

Keywords, variables, and operators can be displayed as required, optional, or default. Fragments, separators, and delimiters can be displayed as required or optional.

Item type
Definition

Required
Required items are displayed on the main path of the horizontal line.

Optional
Optional items are displayed below the main path of the horizontal line.

Default
Default items are displayed above the main path of the horizontal line.

Syntax examples

The following table provides syntax examples.

Table 1. Syntax examples

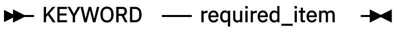
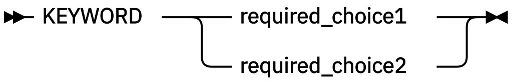
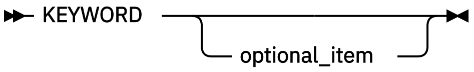
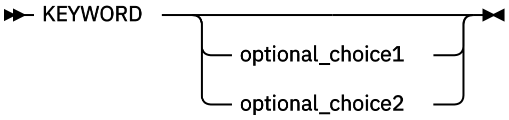
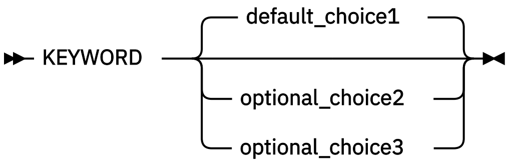
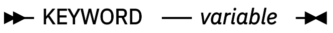
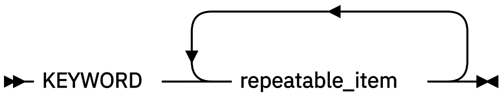
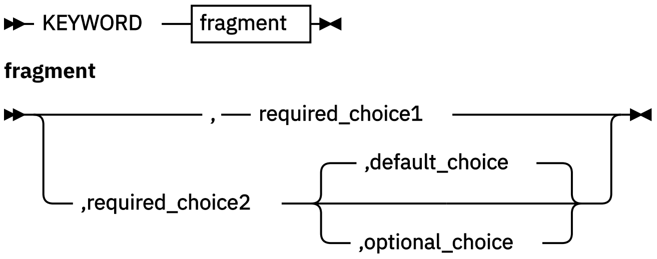
Item	Syntax example
Required item. Required items appear on the main path of the horizontal line. You must specify these items.	
Required choice. A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	

Table 1. Syntax examples (continued)

Item	**Syntax example**
Optional item. Optional items appear below the main path of the horizontal line.	
Optional choice. A optional choice (two or more items) appear in a vertical stack below the main path of the horizontal line. You can choose one of the items in the stack.	
Default. Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.	 default_choice1 KEYWORD optional_choice2 optional_choice3
Variable. Variables appear in lowercase italics. They represent names or values.	
Repeatable item. An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated. An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.	
Fragment. The fragment symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.	 KEYWORD fragment **fragment** , required_choice1 ,default_choice ,required_choice2 ,optional_choice

Where to find more information

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

How to send your comments to IBM

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

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

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

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

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

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

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

- Your name, company/university/institution name, and email address
- The following deliverable title and order number: z/OS TSO/E REXX Reference, SA32-0972-50
- The section title of the specific information to which your comment relates
- The text of your comment.

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

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

If you have a technical problem

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

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

Summary of changes

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

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

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

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

New

The following content is new.

- Added information to demonstrate a generalized version of in-stream REXX execs. For more information, see "Using IRXJCL to execute an in-stream REXX exec" on page 246.

Changed

The following content is changed.

- None

Deleted

The following content was deleted.

- None

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

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

New

Prior to June 2020 refresh

- "OUTTRAP versus MSG function when trapping or suppressing output" on page 39 is added.
- "LISTDSI" on page 115 is updated with the addition of variables, SYSDSVERSION, SYSENCRYPT, SYSKEYLABEL, and SYSMAXGENS.

Changed

September 2020 refresh

- "Host command environments for linking to and attaching programs" on page 30 is updated by adding a clarification regarding linking to and attaching programs.

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

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

New

- The limit for TSO/E user IDs is changed to 8 characters. For more information, see "USERID" on page 104.

Changed

- The example is updated for "ABBREV (Abbreviation)" on page 78.
- The MVSVAR function, with the SYMDEF argument as the first operand, now includes support the long symbol names. See "MVSVAR" on page 128 for more information.

Chapter 1. Introduction

This introduction gives a brief overview of the Systems Application Architecture® (SAA) solution.

What the SAA Solution Is

The SAA solution is based on a set of software interfaces, conventions, and protocols that provide a framework for designing and developing applications.

The purpose of SAA REXX is to define a common subset of the language that can be used on several environments. TSO/E REXX is the implementation of SAA REXX on the z/OS system. If you plan on running your REXX programs on other environments, however, some restrictions can apply and you should review the publication *SAA Common Programming Interface REXX Level 2 Reference*.

The SAA solution:

- Defines a Common Programming Interface that you can use to develop consistent, integrated enterprise software
- Defines Common Communications Support that you can use to connect applications, systems, networks, and devices
- Defines a Common User Access architecture that you can use to achieve consistency in screen layout and user interaction techniques
- Offers some applications and application development tools written by IBM

Supported environments

Several combinations of IBM hardware and software have been selected as SAA environments. These are environments in which IBM manages the availability of support for applicable SAA elements, and the conformance of those elements to SAA specifications. The SAA environments are the following:

- z/OS
  - Base system (TSO/E, APPC/MVS, batch)
  - CICS® TS
  - IMS
- z/VM®
- i5/OS
- Operating System/2® (OS/2)

Common programming interface

The Common Programming Interface (CPI) provides languages and services that programmers can use to develop applications that take advantage of SAA consistency.

The components of the interface currently fall into two general categories:

- Languages
  - Application Generator
  - C
  - COBOL
  - FORTRAN
  - PL/I
  - REXX (formerly called Procedures Language)

  - RPG
- Services
  - Communications
  - Database
  - Dialog
  - Language Environment®
  - Presentation
  - PrintManager
  - Query
  - Repository
  - Resource Recovery

The CPI is not in itself a product or a piece of code. But—as a definition—it does establish and control how IBM products are being implemented, and it establishes a common base across the applicable SAA environments.

Benefits of using a compiler

The IBM Compiler for REXX/370 and the IBM Library for REXX/370 provide significant benefits for programmers during program development and for users when a program is run. The benefits are:

- Improved performance
- Reduced system load
- Protection for source code and programs
- Improved productivity and quality
- Portability of compiled programs
- SAA compliance checking

Improved performance

The performance improvements that you can expect when you run compiled REXX programs depend on the type of program. A program that performs large numbers of arithmetic operations of default precision shows the greatest improvement. A program that mainly issues commands to the host shows minimal improvement because REXX cannot decrease the time taken by the host to process the commands.

Reduced system load

Compiled REXX programs run faster than interpreted programs. Because a program has to be compiled only one time, system load is reduced and response time is improved when the program is run frequently.

For example, a REXX program that performs many arithmetic operations might take 12 seconds to run interpreted. If the program is run 60 times, it uses about 12 minutes of processor time. The same program when compiled might run six times faster, using only about 2 minutes of processor time.

Protection for source code and programs

Your REXX programs and algorithms are assets that you want to protect.

The Compiler produces object code, which helps you protect these assets by discouraging other users from making unauthorized changes to your programs. You can distribute your REXX programs in object code only.

Load modules can be further protected by using a security product, such as the Resource Access Control Facility (RACF®).

Improved productivity and quality

The Compiler can produce source listings, cross-reference listings, and messages, which help you more easily develop and maintain your REXX programs.

The Compiler identifies syntax errors in a program before you start testing it. You can then focus on correcting errors in logic during testing with the REXX interpreter.

Portability of compiled programs

A REXX program compiled under MVS/ESA can run under CMS. Similarly, a REXX program compiled under CMS can run under MVS/ESA.

SAA compliance checking

The Systems Application Architecture (SAA) definitions of software interfaces, conventions, and protocols provide a framework for designing and developing applications that are consistent within and across several operating systems. SAA REXX is a subset of the REXX language supported by the interpreter under TSO/E and can be used in this operating environment.

To help you write programs for use in all SAA environments, the Compiler can optionally check for SAA compliance. With this option in effect, an attention message is issued for each non-SAA item found in a program.

For more information, see *IBM Compiler for REXX/370: Introducing the Next Step in REXX Programming*.

Chapter 2. REXX general concepts

The REstructured eXtended eXecutor (REXX) language is particularly suitable for:

- Command procedures
- Application front ends
- User-defined macros (such as editor subcommands)
- Prototyping
- Personal computing. Individual users can write programs for their own needs.

REXX is a general purpose programming language like PL/I. REXX has the usual structured-programming instructions — IF, SELECT, DO WHILE, LEAVE, and so on — and a number of useful built-in functions.

The language imposes no restrictions on program format. There can be more than one clause on a line, or a single clause can occupy more than one line. Indentation is allowed. You can, therefore, code programs in a format that emphasizes their structure, making them easier to read.

While architecturally there is no limit to the length of the values of variables, as long as all variables fit into the storage available, there is a TSO/E implementation limit of 16MB.

Implementation maximum: Since no single request for storage can exceed the fixed limit of 16 MB in TSO/E REXX, this limits the size of a variable plus any control information to 16MB. This limit also applies to buffers obtained to hold numeric results.

The limit on the length of symbols (variable names) is 250 characters.

You can use compound symbols, such as

```
NAME.Y.Z
```

(where Y and Z can be the names of variables or can be constant symbols), for constructing arrays and for other purposes.

Issuing host commands from within a REXX program is an integral part of the REXX language. For example, in the TSO/E address space, you can use TSO/E commands in a REXX exec. The exec can also use ISPF commands and services if the exec runs in ISPF. In execs that run in both TSO/E and non-TSO/E address spaces, you can use the TSO/E REXX commands, such as MAKEBUF, DROPBUF, and NEWSTACK. You can also link to or attach programs. “Host commands and host command environments” on page 23 describes the different environments for using host services.

TSO/E REXX execs can reside in a sequential data set or in a member of a partitioned data set (PDS). Partitioned data sets containing REXX execs can be allocated to either the system file SYSPROC (TSO/E address space only) or SYSEXEC. In the TSO/E address space, you can also use the TSO/E ALTLIB command to define alternate exec libraries for storing REXX execs. For more information about allocating exec data sets, see *z/OS TSO/E REXX User's Guide*.

In TSO/E, you can call an exec explicitly using the EXEC command followed by the data set name and the "exec" keyword operand of the EXEC command. The "exec" keyword operand distinguishes the REXX exec from a TSO/E CLIST, which you also call using the EXEC command.

You can call an exec implicitly by entering the member name of the exec. You can call an exec implicitly only if the PDS in which the exec is stored has been allocated to a system file (SYSPROC or SYSEXEC). SYSEXEC is a system file whose data sets can contain REXX execs only. SYSPROC is a system file whose data sets can contain either CLISTs or REXX execs. If an exec is in a data set that is allocated to SYSPROC, the exec must start with a comment containing the characters "REXX" within the first line (line 1). This enables the TSO/E EXEC command to distinguish a REXX exec from a CLIST. For more information, see “Structure and general syntax” on page 6.

SYSEXEC is the default load ddname from which REXX execs are loaded. If your installation plans to use REXX, store your REXX execs in data sets that are allocated to SYSEXEC, not SYSPROC. This makes them easier to maintain. For more information about the load ddname and searching SYSPROC or SYSEXEC, see "Using SYSPROC and SYSEXEC for REXX execs" on page 353.

REXX programs are run by a language processor (interpreter). That is, the program is run line-by-line and word-by-word, without first being translated to another form (compiled). The advantage of this is you can fix the error and rerun the program faster than you can with a compiler.

When an exec is loaded into storage, the load routine checks for sequence numbers in the data set. The routine removes the sequence numbers during the loading process. For information about how the load routine checks for sequence numbers, see "Exec load routine" on page 392.

There is also a set of z/OS UNIX extensions to the TSO/E REXX language which enables REXX programs to access z/OS UNIX callable services. The z/OS UNIX extensions, called syscall commands, have names that correspond to the names of the callable services that they invoke—for example, access, chmod, and chown. For more information about the z/OS UNIX extensions, see *z/OS Using REXX and z/OS UNIX System Services*.

Structure and general syntax

If you store a REXX exec in a data set that is allocated to SYSPROC, the exec must start with a comment and the comment must contain the characters "REXX" within the first line (line 1) of the exec. This is known as the *REXX exec identifier* and is required in order for the TSO/E EXEC command to distinguish REXX execs from TSO/E CLISTs, which are also stored in SYSPROC.

The characters "REXX" must be in the first line (line 1) even if the comment spans multiple lines. In Figure 1 on page 6, example A on the left is correct. The program starts with a comment and the characters "REXX" are in the first line (line 1). Example B on the right is incorrect. The program starts with a comment. However, although the comment contains the characters "REXX", they are *not* in the first line (line 1).

```
     Example A (Correct)                         Example B (Incorrect)

/* REXX program to check ...              /* This program checks ...
   ...  The program then ... */              ... in REXX and ...      */
ADDRESS CPICOMM                           ADDRESS CPICOMM
   ...                                       ...
   ...                                       ...
   ...                                       ...
EXIT                                      EXIT
```

Figure 1. Example of using the REXX exec identifier

If the exec is in a data set that is allocated to a file containing REXX execs only, not CLISTs (for example, SYSEXEC), the comment including the characters "REXX" is not required. However, it is suggested that you start all REXX execs with a comment in the first column of the first line and include the characters "REXX" in the comment. In particular, this is suggested if you are writing REXX execs for use in other SAA environments. Including "REXX" in the first comment also helps users identify that the program is a REXX program and distinguishes a REXX exec from a TSO/E CLIST. For more information about how the EXEC command processor distinguishes REXX execs and CLISTs, see *z/OS TSO/E Command Reference*.

A REXX program is built from a series of **clauses** that are composed of:

- Zero or more blanks (which are ignored)
- A sequence of tokens (see "Tokens" on page 8)
- Zero or more blanks (again ignored)
- A semicolon (;) delimiter that can be implied by line-end, certain keywords, or the colon (:)

Conceptually, each clause is scanned from left to right before processing, and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and multiple blanks (except within literal strings) are converted to single blanks. Blanks adjacent to operator characters and special characters (see "Tokens" on page 8) are also removed.

Characters

A character is a member of a defined set of elements that is used for the control or representation of data. You can usually enter a character with a single keystroke. The coded representation of a character is its representation in digital form. A character, the letter A, for example, differs from its *coded representation* or encoding. Various coded character sets (such as ASCII and EBCDIC) use different encoding for the letter A (decimal values 65 and 193, respectively). This book uses characters to convey meanings and not to imply a specific character code, except where otherwise stated. The exceptions are certain built-in functions that convert between characters and their representations. The functions C2D, C2X, D2C, X2C, and XRANGE have a dependence on the character set in use.

A code page specifies the encoding for each character in a set. Appendix E, "REXX symbol and hexadecimal code cross-reference," on page 463 shows the REXX symbols and their hexadecimal values as found in the U.S. Code Page (037).

You should be aware that:

- Some code pages do not contain all characters that REXX defines as valid (for example, ¬, the logical NOT character).
- Some characters that REXX defines as valid have different encoding in different code pages (for example, `!`, the exclamation point).

One result of these differences is that users in countries using code pages other than 037 might have to enter a character different from the one shown in this manual to accomplish the same result.

Another result of character sensitivity in different code pages concerns operating environments. When REXX programs are transferred between different environments, such as PC REXX and TSO/E REXX, adjustments might need to be made as special characters might have different meanings in the different environments.

For information about Double-Byte Character Set characters, see Appendix A, "Double-byte character set (DBCS) support," on page 429.

Comments

A comment is a sequence of characters (on one or more lines) delimited by `/*` and `*/`. Within these delimiters any characters are allowed. Comments can contain other comments, as long as each begins and ends with the necessary delimiters. They are called **nested comments**. Comments can be anywhere and can be of any length. They have no effect on the program, but they do act as separators. (Two tokens with only a comment in between are not treated as a single token.)

```
/* This is an example of a valid REXX comment */
```

Take special care when commenting out lines of code containing `/*` or `*/` as part of a literal string. Consider the following program segment:

```
01    parse pull input
02    if substr(input,1,5) = '/*123'
03      then call process
04    dept = substr(input,32,5)
```

To comment out lines 2 and 3, the following change would be incorrect:

```
01    parse pull input
02 /* if substr(input,1,5) = '/*123'
03      then call process
04 */ dept = substr(input,32,5)
```

This is incorrect because the language processor would interpret the `/*` that is part of the literal string `/*123` as the start of a nested comment. It would not process the rest of the program because it would be looking for a matching comment end (`*/`).

You can avoid this type of problem by using concatenation for literal strings containing /* or */; line 2 would be:

```
if substr(input,1,5) = '/' || '*123'
```

You could comment out lines 2 and 3 correctly as follows:

```
01    parse pull input
02 /* if substr(input,1,5) = '/' || '*123'
03       then call process
04 */ dept = substr(input,32,5)
```

For information about Double-Byte Character Set characters, see Appendix A, “Double-byte character set (DBCS) support,” on page 429 and the OPTIONS instruction in topic“OPTIONS” on page 56.

Tokens

A token is the unit of low-level syntax from which clauses are built. Programs written in REXX are composed of tokens. They are separated by blanks or comments or by the nature of the tokens themselves. The classes of tokens are:

Literal Strings:

A literal string is a sequence including *any* characters and delimited by the single quotation mark (') or the double quotation mark ("). Use two consecutive double quotation marks ("") to represent a " character within a string delimited by double quotation marks. Similarly, use two consecutive single quotation marks ('') to represent a ' character within a string delimited by single quotation marks. A literal string is a constant and its contents are never modified when it is processed.

A literal string with no characters (that is, a string of length 0) is called a **null string**.

These are valid strings:

```
'Fred'
"Don't Panic!"
'You shouldn''t'        /* Same as "You shouldn't" */
''                      /* The null string         */
```

Note that a string followed immediately by a (is considered to be the name of a function. If followed immediately by the symbol X or x, it is considered to be a hexadecimal string. If followed immediately by the symbol B or b, it is considered to be a binary string. Descriptions of these forms follow.

Implementation maximum: A literal string can contain up to 250 characters. (But note that the length of computed results is limited only by the amount of storage available.)

Hexadecimal Strings:

A hexadecimal string is a literal string, expressed using a hexadecimal notation of its encoding. It is any sequence of zero or more hexadecimal digits (0–9, a–f, A–F), grouped in pairs. A single leading 0 is assumed, if necessary, at the front of the string to make an even number of hexadecimal digits. The groups of digits are optionally separated by one or more blanks, and the whole sequence is delimited by single or double quotation marks, and immediately followed by the symbol X or x. (Neither x nor X can be part of a longer symbol.) The blanks, which can be present only at byte boundaries (and not at the beginning or end of the string), are to aid readability. The language processor ignores them. A hexadecimal string is a literal string formed by packing the hexadecimal digits given. Packing the hexadecimal digits removes blanks and converts each pair of hexadecimal digits into its equivalent character, for example: 'C1'X to A.

Hexadecimal strings let you include characters in a program even if you cannot directly enter the characters themselves. These are valid hexadecimal strings:

```
'ABCD'x
"1d ec f8"X
"1 d8"x
```

A hexadecimal string is *not* a representation of a number. Rather, it is an escape mechanism that lets a user describe a character in terms of its encoding (and, therefore, is machine-dependent). In EBCDIC, '40'X is the encoding for a blank. In every case, a string of the form '.....'x is simply an alternative to a straightforward string. In EBCDIC 'C1'x and 'A' are identical, as are '40'x and a blank, and must be treated identically.

Implementation maximum: The packed length of a hexadecimal string (the string with blanks removed) cannot exceed 250 bytes.

Binary Strings:

A binary string is a literal string, expressed using a binary representation of its encoding. It is any sequence of zero or more binary digits (0 or 1) in groups of 8 (bytes) or 4 (nibbles). The first group can have fewer than four digits; in this case, up to three 0 digits are assumed to the left of the first digit, making a total of four digits. The groups of digits are optionally separated by one or more blanks, and the whole sequence is delimited by matching single or double quotation marks and immediately followed by the symbol b or B. (Neither b nor B can be part of a longer symbol.) The blanks, which can be present only at byte or nibble boundaries (and not at the beginning or end of the string), are to aid readability. The language processor ignores them.

A binary string is a literal string formed by packing the binary digits given. If the number of binary digits is not a multiple of eight, leading zeros are added on the left to make a multiple of eight before packing. Binary strings allow you to specify characters explicitly, bit by bit.

These are valid binary strings:

```
'11110000'b        /* == 'f0'x                 */
"101 1101"b        /* == '5d'x                 */
'1'b               /* == '00000001'b and '01'x  */
'10000 10101010'b  /* == '0001 0000 1010 1010'b */
''b                /* == ''                    */
```

Symbols:

Symbols are groups of characters, selected from the:

- English alphabetic characters (A–Z and a–z[1])
- Numeric characters (0–9)
- Characters @ # $ ¢ . ![2] ? and underscore.
- Double-Byte Character Set (DBCS) characters (X'41'–X'FE')—ETMODE must be in effect for these characters to be valid in symbols.

Any lowercase alphabetic character in a symbol is translated to uppercase (that is, lowercase a–z to uppercase A–Z) before use.

These are valid symbols:

```
Fred
Albert.Hall
WHERE?
```

If a symbol does not begin with a digit or a period, you can use it as a variable and can assign it a value. If you have not assigned it a value, its value is the characters of the symbol itself, translated to uppercase (that is, lowercase a–z to uppercase A–Z). Symbols that begin with a number or a period are constant symbols and cannot be assigned a value.

One other form of symbol is allowed to support the representation of numbers in exponential format. The symbol starts with a digit (0–9) or a period, and it can end with the sequence E or e, followed immediately by an optional sign (- or +), followed immediately by one or more digits (which cannot be followed by any other symbol characters). The sign in this context is part of the symbol and is not an operator.

These are valid numbers in exponential notation:

[1] Note that some code pages do not include lowercase English characters a–z.

[2] The encoding of the exclamation point character depends on the code page in use.

```
17.3E-12
.03e+9
```

Implementation maximum: A symbol can consist of up to 250 characters. (But note that its value, if it is a variable, is limited only by the amount of storage available.)

Numbers:

These are character strings consisting of one or more decimal digits, with an optional prefix of a plus or minus sign, and optionally including a single period (.) that represents a decimal point. A number can also have a power of 10 suffixed in conventional exponential notation: an E (uppercase or lowercase), followed optionally by a plus or minus sign, then followed by one or more decimal digits defining the power of 10. Whenever a character string is used as a number, rounding may occur to a precision specified by the NUMERIC DIGITS instruction (default nine digits). See topics Chapter 6, "Numbers and arithmetic," on page 171-"Errors" on page 178 for a full definition of numbers.

Numbers can have leading blanks (before and after the sign, if any) and can have trailing blanks. Blanks might not be embedded among the digits of a number or in the exponential part. Note that a symbol (see preceding) or a literal string can be a number. A number cannot be the name of a variable.

These are valid numbers:

```
12
'-17.9'
127.0650
73e+128
' + 7.9E5 '
```

A **whole number** is a number that has a zero (or no) decimal part and that the language processor would not usually express in exponential notation. That is, it has no more digits before the decimal point than the current setting of NUMERIC DIGITS (the default is 9).

Implementation maximum: The exponent of a number expressed in exponential notation can have up to nine digits.

Operator Characters:

The characters: + - \ / % * | & = ¬ > < and the sequences >= <= \> \< \= >< <> == \== // && || ** ¬> ¬< ¬= ¬== >> << >>= \<< ¬<< \>> ¬>> <<= /= /== indicate operations (see topic "Operators" on page 12). A few of these are also used in parsing templates, and the equal sign is also used to indicate assignment. Blanks adjacent to operator characters are removed. Therefore, the following are identical in meaning:

```
345>=123
345 >=123
345 >= 123
345 > = 123
```

Some of these characters might not be available in all character sets, and, if this is the case, appropriate translations can be used. In particular, the vertical bar (|) **or** character is often shown as a split vertical bar.

Throughout the language, the **not** character, ¬, is synonymous with the backslash (\). You can use the two characters interchangeably according to availability and personal preference.

Special Characters:

The following characters, together with the individual characters from the operators, have special significance when found outside of literal strings:

```
,   ;   :   )   (
```

These characters constitute the set of special characters. They all act as token delimiters, and blanks adjacent to any of these are removed. There is an exception: a blank adjacent to the outside of a parenthesis is deleted only if it is also adjacent to another special character (unless the character is a parenthesis and the blank is outside it, too). For example, the language processor does not remove the blank in A (Z). This is a concatenation that is not equivalent to A(Z), a function call. The language processor does remove the blanks in (A) + (Z) because this is equivalent to (A)+(Z).

The following example shows how a clause is composed of tokens.

```
'REPEAT'   A + 3;
```

This is composed of six tokens—a literal string ('REPEAT'), a blank operator, a symbol (A, which can have a value), an operator (+), a second symbol (3, which is a number and a symbol), and the clause delimiter (;). The blanks between the A and the + and between the + and the 3 are removed. However, one of the blanks between the 'REPEAT' and the A remains as an operator. Thus, this clause is treated as though written:

```
'REPEAT' A+3;
```

Implied semicolons

The last element in a clause is the semicolon delimiter. The language processor implies the semicolon: at a line-end, after certain keywords, and after a colon if it follows a single symbol. This means that you need to include semicolons only when there is more than one clause on a line or to end an instruction whose last character is a comma.

A line-end usually marks the end of a clause and, thus, REXX implies a semicolon at most end of lines. However, there are the following exceptions:

- The line ends in the middle of a string.
- The line ends in the middle of a comment. The clause continues on to the next line.
- The last token was the continuation character (a comma) and the line does not end in the middle of a comment. (Note that a comment is not a token.)

REXX automatically implies semicolons after colons (when following a single symbol, a label) and after certain keywords when they are in the correct context. The keywords that have this effect are: ELSE, OTHERWISE, and THEN. These special cases reduce typographical errors significantly.

Restriction: The two characters forming the comment delimiters, /* and */, must not be split by a line-end (that is, / and * should not appear on different lines) because they could not then be recognized correctly; an implied semicolon would be added. The two consecutive characters forming a literal quotation mark within a string are also subject to this line-end ruling.

Continuations

A clause can be continued onto the next line by using the comma, which is referred to as the **continuation character**. The comma is functionally replaced by a blank, and, thus, no semicolon is implied. One or more comments can follow the continuation character before the end of the line. The continuation character cannot be used in the middle of a string or it will be processed as part of the string itself. The same situation holds true for comments. Note that the comma remains in execution traces.

The following example shows how to use the continuation character to continue a clause.

```
say 'You can use a comma',
    'to continue this clause.'
```

This displays:

```
You can use a comma to continue this clause.
```

A clause can also be continued onto the next line by using nulls. That is, the clause can be a literal string that contains blank characters and spans multiple lines until the ending quote is encountered. A literal string can contain up to 250 characters.

Expressions and operators

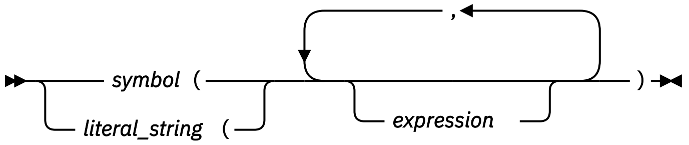
Expressions in REXX are a general mechanism for combining one or more pieces of data in various ways to produce a result, usually different from the original data. Appendix E, "REXX symbol and hexadecimal code cross-reference," on page 463 shows the REXX symbols and their hexadecimal values as found in the U.S. Code Page (037).

Expressions

Expressions consist of one or more **terms** (literal strings, symbols, function calls, or subexpressions) interspersed with zero or more operators that denote operations to be carried out on terms. A **subexpression** is a term in an expression bracketed within a left and a right parenthesis.

Terms include:

- **Literal Strings** (delimited by quotation marks), which are constants
- **Symbols** (no quotation marks), which are translated to uppercase. A symbol that does not begin with a digit or a period may be the name of a variable; in this case the value of that variable is used. Otherwise a symbol is treated as a constant string. A symbol can also be **compound**.
- **Function calls** (see Chapter 4, "Functions," on page 73), which are of the form:



Evaluation of an expression is left to right, modified by parentheses and by operator precedence in the usual algebraic manner (see "Parentheses and operator precedence" on page 15). Expressions are wholly evaluated, unless an error occurs during evaluation.

All data is in the form of "typeless" character strings (typeless because it is not—as in some other languages—of a particular declared type, such as Binary, Hexadecimal, Array, and so forth). Consequently, the result of evaluating any expression is itself a character string. Terms and results (except arithmetic and logical expressions) can be the **null string** (a string of length 0). Note that REXX imposes no restriction on the maximum length of results. However, there is usually some practical limitation dependent upon the amount of storage available to the language processor.

Operators

An **operator** is a representation of an operation, such as addition, to be carried out on one or two terms. The following topics describe how each operator (except for the prefix operators) acts on two terms, which can be symbols, strings, function calls, intermediate results, or subexpressions. Each prefix operator acts on the term or subexpression that follows it. Blanks (and comments) adjacent to operator characters have no effect on the operator; thus, operators constructed from more than one character can have embedded blanks and comments. In addition, one or more blanks, where they occur in expressions but are not adjacent to another operator, also act as an operator. There are four types of operators:

- Concatenation
- Arithmetic
- Comparison
- Logical

String concatenation

The concatenation operators combine two strings to form one string by appending the second string to the right-hand end of the first string. The concatenation might occur with or without an intervening blank. The concatenation operators are:

(blank)
Concatenate terms with one blank in between

||
Concatenate without an intervening blank

(abuttal)
Concatenate without an intervening blank

You can force concatenation without a blank by using the `||` operator.

The **abuttal** operator is assumed between two terms that are not separated by another operator. This can occur when two terms are syntactically distinct, such as a literal string and a symbol, or when they are separated only by a comment.

Examples:

An example of syntactically distinct terms is: if `Fred` has the value `37.4`, then `Fred'%'` evaluates to `37.4%`.

If the variable PETER has the value 1, then `(Fred)(Peter)` evaluates to `37.41`.

In EBCDIC, the two adjoining strings, one hexadecimal and one literal,

```
'c1 c2'x'CDE'
```

evaluate to `ABCDE`.

In the case of:

```
Fred/* The NOT operator precedes Peter. */¬Peter
```

there is no abuttal operator implied, and the expression is not valid. However,

```
(Fred)/* The NOT operator precedes Peter. */(¬Peter)
```

results in an abuttal, and evaluates to `37.40`.

Arithmetic

You can combine character strings that are valid numbers (see “Tokens” on page 8) using the arithmetic operators:

+
Add

-
Subtract

*
Multiply

/
Divide

%
Integer divide (divide and return the integer part of the result)

//
Remainder (divide and return the remainder—not modulo, because the result may be negative)

**
Power® (raise a number to a whole-number power)

Prefix -
Same as the subtraction: `0 - number`

Prefix +
Same as the addition: `0 + number`

See Chapter 6, “Numbers and arithmetic,” on page 171 for details about precision, the format of valid numbers, and the operation rules for arithmetic. Note that if an arithmetic result is shown in exponential notation, it is likely that rounding has occurred.

Comparison

The comparison operators compare two terms and return the value 1 if the result of the comparison is true, or 0 otherwise.

The strict comparison operators all have one of the characters defining the operator doubled. The ==, \==, /==, and ¬== operators test for an exact match between two strings. The two strings must be identical (character by character) and of the same length to be considered strictly equal. Similarly, the strict comparison operators such as >> or << carry out a simple character-by-character comparison, with no padding of either of the strings being compared. The comparison of the two strings is from left to right. If one string is shorter than and is a leading substring of another, then it is smaller than (less than) the other. The strict comparison operators also do not attempt to perform a numeric comparison on the two operands.

For all the other comparison operators, if *both* terms involved are numeric, a numeric comparison (in which leading zeros are ignored, and so forth—see “Numeric comparisons” on page 176) is effected. Otherwise, both terms are treated as character strings (leading and trailing blanks are ignored, and then the shorter string is padded with blanks on the right).

Character comparison and strict comparison operations are both case-sensitive, and for both the exact collating order may depend on the character set used for the implementation. For example, in an EBCDIC environment, lowercase alphabetics precede uppercase, and the digits 0–9 are higher than all alphabetics. In an ASCII environment, the digits are lower than the alphabetics, and lowercase alphabetics are higher than uppercase alphabetics.

The comparison operators and operations are:

=
: True if the terms are equal (numerically or when padded, and so forth)

\=, ¬=, /=
: True if the terms are not equal (inverse of =)

>
: Greater than

<
: Less than

><
: Greater than or less than (same as not equal)

<>
: Greater than or less than (same as not equal)

>=
: Greater than or equal to

\<, ¬<
: Not less than

<=
: Less than or equal to

\>, ¬>
: Not greater than

==
: True if terms are strictly equal (identical)

\==, ¬==, /==
: True if the terms are NOT strictly equal (inverse of ==)

>>
Strictly greater than

<<
Strictly less than

>>=
Strictly greater than or equal to

\<<, ¬<<
Strictly NOT less than

<<=
Strictly less than or equal to

\>>, ¬>>
Strictly NOT greater than

Guideline: Throughout the language, the **not** character, ¬, is synonymous with the backslash (\). You can use the two characters interchangeably, according to availability and personal preference. The backslash can appear in the following operators: \ (prefix not), \=, \==, \<, \>, \<<, and \>>.

Logical (boolean)

A character string is taken to have the value false if it is `0`, and true if it is `1`. The logical operators take one or two such values (values other than `0` or `1` are not allowed) and return `0` or `1` as appropriate:

&
AND

Returns `1` if both terms are true.

|
Inclusive OR

Returns `1` if either term is true.

&&
Exclusive OR

Returns `1` if either (but not both) is true.

Prefix \,¬
Logical NOT

Negates; `1` becomes `0`, and `0` becomes `1`

Parentheses and operator precedence

Expression evaluation is from left to right; parentheses and operator precedence modify this:

- When parentheses are encountered (other than those that identify function calls) the entire subexpression between the parentheses is evaluated immediately when the term is required.
- When the sequence:

  ```
  term1 operator1 term2 operator2 term3
  ```

  is encountered, and `operator2` has a higher precedence than `operator1`, the subexpression (`term2 operator2 term3`) is evaluated first. The same rule is applied repeatedly as necessary.

  Note, however, that individual terms are evaluated from left to right in the expression (that is, as soon as they are encountered). The precedence rules affect only the order of **operations**.

For example, `*` (multiply) has a higher priority than + (add), so 3+2`*`5 evaluates to 13 (rather than the 25 that would result if strict left to right evaluation occurred). To force the addition to occur before the multiplication, you could rewrite the expression as `(3+2)*5`. Adding the parentheses makes the first three tokens a subexpression. Similarly, the expression -3**2 evaluates to 9 (instead of -9) because the prefix minus operator has a higher priority than the power operator.

The order of precedence of the operators is (highest at the top):

**+ - ¬ **
: (prefix operators)

******
: (power)

*** / % //**
: (multiply and divide)

+ -
: (add and subtract)

(blank) || (abuttal)
: (concatenation with or without blank)

= > <
: (comparison operators)

== >> <<

\\= ¬=

>< <>

\\> ¬>

\\< ¬<

\\== ¬==

\\>> ¬>>

\\<< ¬<<

>= >>=

<= <<=

/= /==

&
: (and)

| &&
: (or, exclusive or)

Examples:

Suppose the symbol A is a variable whose value is 3, DAY is a variable whose value is Monday, and other variables are uninitialized. Then:

```
A+5                  ->    '8'
A-4*2                ->    '-5'
A/2                  ->    '1.5'
0.5**2               ->    '0.25'
(A+1)>7              ->    '0'         /* that is, False */
' '=''               ->    '1'         /* that is, True  */
' '==''              ->    '0'         /* that is, False */
' '¬==''               ->    '1'         /* that is, True  */
(A+1)*3=12             ->    '1'         /* that is, True  */
'077'>'11'             ->    '1'         /* that is, True  */
'077' >> '11'          ->    '0'         /* that is, False */
'abc' >> 'ab'          ->    '1'         /* that is, True  */
'abc' << 'abd'         ->    '1'         /* that is, True  */
'ab ' << 'abd'         ->    '1'         /* that is, True  */
Today is Day           ->    'TODAY IS Monday'
'If it is' day         ->    'If it is Monday'
Substr(Day,2,3)        ->    'ond'    /* Substr is a function */
'!'xxx'!'              ->    '!XXX!'
'000000' >> '0E0000'   ->    '1'         /* that is, True  */
```

Note: The last example would give a different answer if the > operator had been used rather than >>. Because '0E0000' is a valid number in exponential notation, a numeric comparison is done; thus '0E0000' and '000000' evaluate as equal. The REXX order of precedence usually causes no difficulty because it is the same as in conventional algebra and other computer languages. There are two differences from common notations:

- The prefix minus operator always has a higher priority than the power operator.
- Power operators (like other operators) are evaluated left-to-right.

For example:

```
-3**2      ==  9  /* not -9  */
-(2+1)**2  ==  9  /* not -9  */
2**2**3    == 64  /* not 256 */
```

Clauses and instructions

Clauses can be subdivided into the following types:

Null clauses

A clause consisting only of blanks or comments or both is a **null clause**. It is completely ignored (except that if it includes a comment it is traced, if appropriate).

Tip: A null clause is not an instruction; for example, putting an extra semicolon after the THEN or ELSE in an IF instruction is not equivalent to using a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.

Labels

A clause that consists of a single symbol followed by a colon is a **label**. The colon in this context implies a semicolon (clause separator), so no semicolon is required. Labels identify the targets of CALL instructions, SIGNAL instructions, and internal function calls. More than one label may precede any instruction. Labels are treated as null clauses and can be traced selectively to aid debugging.

Any number of successive clauses can be labels. This permits multiple labels before other clauses. Duplicate labels are permitted, but control passes only to the first of any duplicates in a program. The duplicate labels occurring later can be traced but cannot be used as a target of a CALL, SIGNAL, or function invocation.

Appendix E, "REXX symbol and hexadecimal code cross-reference," on page 463 shows the REXX symbols and their hexadecimal values as found in the U.S. Code Page (037).

You can use DBCS characters in labels. See Appendix A, "Double-byte character set (DBCS) support," on page 429 for more information.

Instructions

An **instruction** consists of one or more clauses describing some course of action for the language processor to take. Instructions can be: assignments, keyword instructions, or commands.

Assignments

A single clause of the form *symbol=expression* is an instruction known as an **assignment**. An assignment gives a variable a (new) value. See "Assignments and symbols" on page 18.

Keyword instructions

A **keyword instruction** is one or more clauses, the first of which starts with a keyword that identifies the instruction. Keyword instructions control the external interfaces, the flow of control, and so forth. Some keyword instructions can include nested instructions. In the following example, the DO construct (DO, the group of instructions that follow it, and its associated END keyword) is considered a single keyword instruction.

```
DO
   instruction
   instruction
   instruction
END
```

A **subkeyword** is a keyword that is reserved within the context of some particular instruction, for example, the symbols TO and WHILE in the DO instruction.

Commands

A **command** is a clause consisting of only an expression. The expression is evaluated and the result is passed as a command string to some external environment.

Assignments and symbols

A **variable** is an object whose value can change during the running of a REXX program. The process of changing the value of a variable is called **assigning** a new value to it. The value of a variable is a single character string, of any length, that might contain *any* characters.

You can assign a new value to a variable with the ARG, PARSE, or PULL instructions, the VALUE built-in function, or the Variable Access Routine (IRXEXCOM), but the most common way of changing the value of a variable is the assignment instruction itself. Any clause of the form:

symbol=expression;

is taken to be an assignment. The result of *expression* becomes the new value of the variable named by the symbol to the left of the equal sign. On TSO/E, if you omit `expression`, the variable is set to the null string. However, it is suggested that you explicitly set a variable to the null string: `symbol=''`.

Appendix E, "REXX symbol and hexadecimal code cross-reference," on page 463 shows the REXX symbols and their hexadecimal values as found in the U.S. Code Page (037).

Variable names can contain DBCS characters. For information about DBCS characters, see Appendix A, "Double-byte character set (DBCS) support," on page 429.

Example:

```
/* Next line gives FRED the value "Frederic" */
Fred='Frederic'
```

The symbol naming the variable cannot begin with a digit (0–9) or a period. (Without this restriction on the first character of a variable name, you could redefine a number; for example `3=4;` would give a variable called 3 the value 4.)

You can use a symbol in an expression even if you have not assigned it a value, because a symbol has a defined value at all times. A variable you have not assigned a value is **uninitialized**. Its value is the characters of the symbol itself, translated to uppercase (that is, lowercase a–z to uppercase A–Z). However, if it is a compound symbol (described under "Compound symbols" on page 19), its value is the derived name of the symbol.

Example:

```
/* If Freda has not yet been assigned a value,   */
/* then next line gives FRED the value "FREDA"   */
Fred=Freda
```

The meaning of a symbol in REXX varies according to its context. As a term in an expression (rather than a keyword of some kind, for example), a symbol belongs to one of four groups: constant symbols, simple symbols, compound symbols, and stems. Constant symbols cannot be assigned new values. You can use simple symbols for variables where the name corresponds to a single value. You can use compound symbols and stems for more complex collections of variables, such as arrays and lists.

Constant symbols

A **constant symbol** starts with a digit (0–9) or a period.

You cannot change the value of a constant symbol. It is simply the string consisting of the characters of the symbol (that is, with any lowercase alphabetic characters translated to uppercase).

These are constant symbols:

```
77
827.53
.12345
12e5       /* Same as 12E5 */
3D
17E-3
```

Appendix E, "REXX symbol and hexadecimal code cross-reference," on page 463 shows the REXX symbols and their hexadecimal values as found in the U.S. Code Page (037).

Simple symbols

A **simple symbol** does not contain any periods and does not start with a digit (0–9).

By default, its value is the characters of the symbol (that is, translated to uppercase). If the symbol has been assigned a value, it names a variable and its value is the value of that variable.

These are simple symbols:

```
FRED
Whatagoodidea?    /* Same as WHATAGOODIDEA? */
?12
```

Appendix E, "REXX symbol and hexadecimal code cross-reference," on page 463 shows the REXX symbols and their hexadecimal values as found in the U.S. Code Page (037).

Compound symbols

A **compound symbol** permits the substitution of variables within its name when you refer to it. A compound symbol contains at least one period and at least two other characters. It cannot start with a digit or a period, and if there is only one period in the compound symbol, it cannot be the last character.

The name begins with a **stem** (that part of the symbol up to and including the first period). This is followed by a **tail**, parts of the name (delimited by periods) that are constant symbols, simple symbols, or null. The **derived name** of a compound symbol is the stem of the symbol, in uppercase, followed by the tail, in which all simple symbols have been replaced with their values. A tail itself can be comprised of the characters A–Z, a–z, 0–9, and @ # $ ¢ . ! ? and underscore. The value of a tail can be any character string, including the null string and strings containing blanks. For example:

```
taila='* ('
tailb=''
stem.taila=99
stem.tailb=stem.taila
say stem.tailb        /* Displays: 99                 */
```

```
/* But the following instruction would cause an error */
/*         say stem.* (                              */
```

You cannot use constant symbols with embedded signs (for example, 12.3E+5) after a stem; in this case, the whole symbol would not be a valid symbol.

These are compound symbols:

```
FRED.3
Array.I.J
AMESSY..One.2.
```

Before the symbol is used (that is, at the time of reference), the language processor substitutes the values of any simple symbols in the tail (I, J, and One in the examples), thus generating a new, derived name. This derived name is then used just like a simple symbol. That is, its value is by default the derived name, or (if it has been used as the target of an assignment) its value is the value of the variable named by the derived name.

The substitution into the symbol that takes place permits arbitrary indexing (subscripting) of collections of variables that have a common stem. Note that the values substituted can contain *any* characters (including periods and blanks). Substitution is done only one time.

To summarize: the derived name of a compound variable that is referred to by the symbol

```
s0.s1.s2. --- .sn
```

is given by

```
d0.v1.v2. --- .vn
```

where d0 is the uppercase form of the symbol s0, and v1 to vn are the values of the constant or simple symbols s1 through sn. Any of the symbols s1-sn can be null. The values v1-vn can also be null and can contain *any* characters (in particular, lowercase characters are not translated to uppercase, blanks are not removed, and periods have no special significance).

Some examples follow in the form of a small extract from a REXX program:

```
a=3       /* assigns '3' to the variable A    */
z=4                 /*   '4'      to Z        */
c='Fred'            /*   'Fred'   to C        */
a.z='Fred'          /*   'Fred'   to A.4      */
a.fred=5            /*   '5'      to A.FRED   */
a.c='Bill'          /*   'Bill'   to A.Fred   */
c.c=a.fred          /*   '5'      to C.Fred   */
y.a.z='Annie'       /*   'Annie'  to Y.3.4    */

say  a  z  c  a.a  a.z  a.c  c.a  a.fred y.a.4
/* displays the string:                     */
/*    "3 4 Fred A.3 Fred Bill C.3 5 Annie"  */
```

You can use compound symbols to set up arrays and lists of variables in which the subscript is not necessarily numeric, thus offering great scope for the creative programmer. A useful application is to set up an array in which the subscripts are taken from the value of one or more variables, effecting a form of associative memory (content addressable).

Implementation maximum: The length of a variable name, before and after substitution, cannot exceed 250 characters.

Appendix E, “REXX symbol and hexadecimal code cross-reference,” on page 463 shows the REXX symbols and their hexadecimal values as found in the U.S. Code Page (037).

Stems

A **stem** is a symbol that contains just one period, which is the last character. It cannot start with a digit or a period.

These are stems:

```
FRED.
A.
```

By default, the value of a stem is the string consisting of the characters of its symbol (that is, translated to uppercase). If the symbol has been assigned a value, it names a variable and its value is the value of that variable.

Further, when a stem is used as the target of an assignment, *all possible* compound variables whose names begin with that stem receive the new value, whether they previously had a value or not. Following the assignment, a reference to any compound symbol with that stem returns the new value until another value is assigned to the stem or or that stem or individual variable is dropped.

For example:

```
hole.  = "empty"
hole.9 = "full"
hole.rat = "full"
rat = "cheese"
drop hole.rat
say hole.1 hole.mouse hole.9 hole.rat
/* says "empty empty full HOLE.cheese" */
```

Thus, you can give a whole collection of variables the same value. For example:

```
total. = 0
do forever
   say "Enter an amount and a name:"
   pull amount name
   if datatype(amount)='CHAR' then leave
   total.name = total.name + amount
   end
```

You can always obtain the value that has been assigned to the whole collection of variables by using the stem. However, this is not the same as using a compound variable whose derived name is the same as the stem. For example:

```
total. = 0
null = ""
total.null = total.null + 5
say total. total.null              /* says "0 5" */
```

You can manipulate collections of variables, referred to by their stem, with the DROP and PROCEDURE instructions. `DROP FRED.` drops all variables with that stem (see page “DROP” on page 50), and `PROCEDURE EXPOSE FRED.` exposes *all possible* variables with that stem (see page “PROCEDURE” on page 60).

Note:

1. When the ARG, PARSE, or PULL instruction or the VALUE built-in function or the Variable Access Routine (IRXEXCOM) changes a variable, the effect is identical with an assignment. Anywhere a value can be assigned, using a stem sets an entire collection of variables.
2. Because an expression can include the operator =, and an instruction can consist purely of an expression (see “Commands to external environments” on page 22), a possible ambiguity is resolved by the following rule: any clause that starts with a symbol and whose second token is (or starts with) an equal sign (=) is an **assignment**, rather than an expression (or a keyword instruction). This is not a restriction, because you can ensure the clause is processed as a command in several ways, such as by putting a null string before the first name, or by enclosing the first part of the expression in parentheses.

   Similarly, if you unintentionally use a REXX keyword as the variable name in an assignment, this should not cause confusion. For example, the clause:

   ```
   Address='10 Downing Street';
   ```

   is an assignment, not an ADDRESS instruction.

3. You can use the SYMBOL function (see "SYMBOL" on page 100) to test whether a symbol has been assigned a value. In addition, you can set SIGNAL ON NOVALUE to trap the use of any uninitialized variables (except when they are tails in compound variables—see page Chapter 7, "Conditions and condition traps," on page 181).

Appendix E, "REXX symbol and hexadecimal code cross-reference," on page 463 shows the REXX symbols and their hexadecimal values as found in the U.S. Code Page (037).

Commands to external environments

Issuing commands to the surrounding environment is an integral part of REXX.

Environment

The system under which REXX programs run is assumed to include at least one host command environment for processing commands. An environment is selected by default on entry to a REXX program. In TSO/E REXX, the environment for processing host commands is known as the *host command environment*. TSO/E provides different environments for TSO/E and non-TSO/E address spaces. You can change the environment by using the ADDRESS instruction. You can find out the name of the current environment by using the ADDRESS built-in function. The underlying operating system defines environments external to the REXX program.

The host command environment selected depends on the caller. For example, if you call a REXX program from a TSO/E address space, the default host command environment that TSO/E provides for processing host commands is TSO. If you call an exec from a non-TSO/E address space, the default host command environment that TSO/E provides is MVS.

TSO/E provides several host command environments for a TSO/E address space (TSO/E and ISPF) and for non-TSO/E address spaces. "Host commands and host command environments" on page 23 explains the different types of host commands you can use in a REXX exec and the different host command environments TSO/E provides for the processing of host commands.

The environments are provided in the *host command environment table*, which specifies the host command environment name and the routine that is called to handle the command processing for that host command environment. You can provide your own host command environment and corresponding routine and define them to the host command environment table. "Host command environment table" on page 331 describes the table in more detail. "Changing the default values for initializing an environment" on page 344 describes how to change the defaults TSO/E provides in order to define your own host command environments. You can also use the IRXSUBCM routine to maintain entries in the host command environment table (see "Entry specifications" on page 280).

Commands

To send a command to the currently addressed host command environment, use a clause of the form:

```
expression;
```

The expression is evaluated, resulting in a character string (which can be the null string), which is then prepared as appropriate and submitted to the host command environment. Any part of the expression not to be evaluated should be enclosed in quotation marks.

The environment then processes the command, which might have side-effects. It eventually returns control to the language processor, after setting a return code. A **return code** is a string, typically a number, that returns some information about the command that has been processed. A return code usually indicates if a command was successful or not but can also represent other information. The language processor places this return code in the REXX special variable RC. See "Special variables" on page 184.

In addition to setting a return code, the underlying system might also indicate to the language processor if an error or failure occurred. An **error** is a condition raised by a command for which a program that uses that command would usually be expected to be prepared. (For example, a locate command to an editing system might report `requested string not found` as an error.) A **failure** is a condition raised by a command for which a program that uses that command would *not* usually be expected to recover (for example, a command that is not executable or cannot be found).

Errors and failures in commands can affect REXX processing if a condition trap for ERROR or FAILURE is ON (see Chapter 7, "Conditions and condition traps," on page 181). They might also cause the command to be traced if `TRACE E` or `TRACE F` is set. `TRACE Normal` is the same as `TRACE F` and is the default—see "TRACE" on page 67.

Here is an example of submitting a command. If the host command environment were TSO/E, the sequence:

```
mydata = "PROGA.LOAD"
"FREE DATASET("mydata")"
```

would result in the string `FREE DATASET(PROGA.LOAD)` being submitted to TSO/E. Of course, the simpler expression:

```
"FREE DATASET(PROGA.LOAD)"
```

would have the same effect in this case.

Recommendation: Whenever you use a host command in a REXX program, enclose the command in double quotation marks. See *z/OS TSO/E REXX User's Guide* for a description of using single and double quotation marks in commands.

On return, the return code from the FREE command is placed in the REXX special variable RC. The return code in RC is '0' if the FREE command processor successfully freed the data set or '12' if it did not. Whenever a host command is processed, the return code from the command is placed in the REXX special variable RC.

Remember that the expression is evaluated before it is passed to the environment. Enclose in quotation marks any part of the expression that is not to be evaluated.

Host commands and host command environments

You can issue host commands from a REXX program. When the language processor processes a clause that it does not recognize as a REXX instruction or an assignment instruction, the language processor considers the clause to be a host command and routes the command to the current host command environment. The host command environment processes the command and then returns control to the language processor.

For example, in REXX processing, a host command can be:

- A TSO/E command processor, such as ALLOCATE, FREE, or EXEC
- A TSO/E REXX command, such as NEWSTACK or QBUF
- A program that you link to or attach
- An MVS system or subsystem command that you invoke during an extended MCS console session
- An ISPF command or service
- An SAA CPI Communications call or APPC/MVS call

If a REXX exec contains

```
FRED var1 var2
```

the language processor considers the clause to be a command and passes the clause to the current host command environment for processing. The host command environment processes the command, sets a return code in the REXX special variable RC, and returns control to the language processor. The return code set in RC is the return code from the host command you specified. For example, the value in RC may be the return code from a TSO/E command processor, an ISPF command or service, or a program you attached. The return code may also be a -3, which indicates that the host command environment could not locate the specified host command (TSO/E command, CLIST, exec, attached or linked routine, ISPF command or service, and so on). A return code of -3 is always returned if you issue a *host command* in an exec and the host command environment cannot locate the command. A return code of -3 is also returned if a TSO/E host command is in the AUTHCMD list of the IKJTSOxx parmlib member and the host command is found in a non-APF authorized library. This is different from what happens in a CLIST or line-mode TSO/E when a command from the AUTHCMD list is found in a non-APF authorized library. In those cases, the command is called and runs unauthorized. However, in a REXX exec if a command from the AUTHCMD list is found in a non-APF authorized library, it fails.

Tip: If you issue a host command from a REXX exec that is running in an authorized or isolated environment, a -3 return code may be returned.

If a system abend occurs during a host command, the REXX special variable RC is set to the negative of the decimal value of the abend code. If a user abend occurs during a host command, the REXX special variable RC is set to the decimal value of the abend code. If no abend occurs during a host command, the REXX special variable RC is set to the decimal value of the return code from the command.

Certain conditions may be raised depending on the value of the special variable RC:

- If the RC value is negative, the FAILURE condition is raised.
- If the RC value is positive, the ERROR condition is raised.
- If the RC value is zero, neither the ERROR nor FAILURE conditions are raised.

See Chapter 7, “Conditions and condition traps,” on page 181 for more information.

Recommendation: If you issue a host command in a REXX exec, you should enclose the entire command in double quotation marks, for example:

```
"routine-name var1 var2"
```

TSO/E provides several host command environments that process different types of host commands. The following topics describe the different host command environments TSO/E provides for non-TSO/E address spaces and for the TSO/E address space (TSO/E and ISPF).

The TSO host command environment

The TSO host command environment is available only to REXX execs that run in the TSO/E address space. Use the TSO host command environment to invoke TSO/E commands and services. You can also invoke all of the TSO/E REXX commands, such as MAKEBUF and NEWSTACK, and invoke other REXX execs and CLISTs. When you invoke a REXX exec in the TSO/E address space, the default initial host command environment is TSO.

Note that the value that can be set in the REXX special variable RC for the TSO environment is a signed 24-bit number in the range -8,388,608 to +8,388,607.

The CONSOLE host command environment

The CONSOLE host command environment is available only to REXX execs that run in the TSO/E address space. Use the CONSOLE environment to invoke MVS system and subsystem commands during an extended MCS console session. To use the CONSOLE environment, you must have CONSOLE command authority.

Before you can use the CONSOLE environment, you must first activate an extended MCS console session using the TSO/E CONSOLE command. After the console session is active, use ADDRESS CONSOLE to issue MVS system and subsystem commands. The CONSOLE environment lets you issue MVS commands from a REXX exec without having to repeatedly issue the CONSOLE command with the SYSCMD keyword. For more information about the CONSOLE environment and related TSO/E services, see Appendix C, “Writing REXX Execs to perform MVS operator activities,” on page 449.

If you use ADDRESS CONSOLE and issue an MVS system or subsystem command before activating a console session, the CONSOLE environment will not be able to locate the command you issued. In this case, the REXX special variable RC is set to -3 and the FAILURE condition is raised. The -3 return code indicates that the host command environment could not locate the command you issued. In this case, the command could not be found because a console session is not active.

Note that the value that can be set in the REXX special variable RC for the CONSOLE environment is a signed 31-bit number in the range -2,147,483,648 to +2,147,483,647.

The ISPEXEC and ISREDIT host command environments

The ISPEXEC and ISREDIT host command environments are available only to REXX execs that run in ISPF. Use the environments to invoke ISPF commands and services, and ISPF edit macros.

When you invoke a REXX exec from ISPF, the default initial host command environment is TSO. You can use the ADDRESS instruction to use an ISPF service. For example, to use the ISPF SELECT service, use the following instruction:

```
ADDRESS ISPEXEC 'SELECT service'
```

The ISREDIT environment lets you issue ISPF edit macros. To use ISREDIT, you must be in an edit session.

Note that the value that can be set in the REXX special variable RC for the ISPEXEC and ISREDIT environments is a signed 24-bit number in the range -8,388,608 to +8,388,607.

The CPICOMM, LU62, and APPCMVS host command environments

The CPICOMM, LU62, and APPCMVS host command environments are available to REXX execs that run in any MVS address space. The CPICOMM environment lets you use the SAA common programming interface (CPI) Communications calls. The LU62 environment lets you use the APPC/MVS calls that are based on the SNA LU 6.2 architecture. The APPCMVS environment allows APPC transaction programs to invoke APPC/MVS server facilities callable services and callable services related to the testing of transaction programs (TPs). Using these environments, you can write APPC/MVS transaction programs (TPs) in the REXX programming language. Using CPICOMM, you can write transaction programs (TPs) in REXX that can be used in different SAA environments.

The CPICOMM environment supports the starter set and advanced function set of the following SAA CPI Communications calls. For more information about each call and its parameters, see *SAA Common Programming Interface Communications Reference*.

- CMACCP (Accept_Conversation)
- CMALLC (Allocate)
- CMCFM (Confirm)
- CMCFMD (Confirmed)
- CMDEAL (Deallocate)
- CMECS (Extract_Conversation_State)
- CMECT (Extract_Conversation_Type)
- CMEMN (Extract_Mode_Name)
- CMEPLN (Extract_Partner_LU_Name)
- CMESL (Extract_Sync_Level)
- CMFLUS (Flush)
- CMINIT (Initialize_Conversation)
- CMPTR (Prepare_To_Receive)
- CMRCV (Receive)
- CMRTS (Request_To_Send)
- CMSCT (Set_Conversation_Type)
- CMSDT (Set_Deallocate_Type)
- CMSED (Set_Error_Direction)
- CMSEND (Send_Data)

- CMSERR (Send_Error)
- CMSF (Set_Fill)
- CMSLD (Set_Log_Data)
- CMSMN (Set_Mode_Name)
- CMSPLN (Set_Partner_LU_Name)
- CMSPTR (Set_Prepare_To_Receive_Type)
- CMSRC (Set_Return_Control)
- CMSRT (Set_Receive_Type)
- CMSSL (Set_Sync_Level)
- CMSST (Set_Send_Type)
- CMSTPN (Set_TP_Name)
- CMTRTS (Test_Request_To_Send_Received)

The LU62 environment supports the following APPC/MVS calls. These calls are based on the SNA LU 6.2 architecture and are referred to as APPC/MVS calls in this book. For more information about the calls and their parameters, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

- ATBALLC (Allocate)
- ATBCFM (Confirm)
- ATBCFMD (Confirmed)
- ATBDEAL (Deallocate)
- ATBFLUS (Flush)
- ATBGETA (Get_Attributes)
- ATBGETC (Get_Conversation)
- ATBGETP (Get_TP_Properties)
- ATBGETT (Get_Type)
- ATBPTR (Prepare_to_Receive)
- ATBRCVI (Receive_Immediate)
- ATBRCVW (Receive_and_Wait)
- ATBRTS (Request_to_Send)
- ATBSEND (Send_Data)
- ATBSERR (Send_Error)

The LU62 host command environment supports the following callable services that existed before MVS/SP 4.3.0, but have been updated:

- ATBALC2 (Allocate)
- ATBGTA2 (Get_Attributes)

With TSO/E APAR OA08990, the REXX LU62 host command environment supports the following updated APPC version 3, 4, 5, and 6 callable services. (The APPC version is that which is returned by ATBVERS, the APPC_Version_Service.)

Version 3:

- ATBEES3 (Error_Extract)

Version 4:

- ATBGTP4 (Get_TP_Properties)
- ATBSSO4 (Set_Syncpt_Options)

Version 5:

- ATBALC5 (Allocate)
- ATBSTO5 (Set_TimeOut_Value)

Version 6:

- ATBALC6 (Allocate)
- ATBSTO6 (Set_TimeOut_Value)
- ATBGTA6 (Get_Attributes)

For more information about the new LU62 service calls and their parameters, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

Note: In order to use the new version 6 services, the user must have the PTF for APAR OA08990 installed and must be running z/OS V1R7 or later.

The APPCMVS host command environment supports the following advanced callable services:

- ATBRFA2 (Register_for_Allocates)
- ATBRAL2 (Receive_Allocate)
- ATBRJC2 (Reject_Conversation)
- ATBSTE2 (Set_Event_Notification)
- ATBGTE2 (Get_Event)
- ATBQAQ2 (Query_Allocate_Queue)
- ATBSAQ2 (Set_Allocate_Queue_Attributes)
- ATBSCA2 (Set_Conversation_Accounting_Information)
- ATBURA2 (Unregister_for_Allocates)
- ATBPOR2 (Post_on_Receipt)

The APPCMVS host command environment supports the following advanced callable services:

- ATBRFA2 (Register_for_Allocates)
- ATBRAL2 (Receive_Allocate)
- ATBTER1 (Register_Test)
- ATBTEA1 (Accept_Test)
- ATBTEU1 (Unregister_Test)
- ATBCUC1 (Cleanup_TP (Unauthorized))
- ATBVERS (APPC_Version_Service)

Restriction: If you use the APPC/MVS calls, be aware that TSO/E REXX does not support data spaces. In addition, the buffer length limit for ATBRCVI, ATBRCVW, and ATBSEND is 16 MB. You can request callable service asynchronous processing on services that provide it. This is specified on the parameter.

To use either an SAA CPI Communications call or an APPC/MVS call, specify the name of the call followed by variable names for each of the parameters. Separate each variable name by one or more blanks. For example:

```
ADDRESS LU62 'ATBCFMD conversation_ID notify_type return_code'
```

You must enclose the entire call in single or double quotation marks. You must also pass a variable name for each parameter. Do not pass actual values for the parameters. By enclosing the call in quotation marks, the language processor does not evaluate any variables and simply passes the expression to the host command environment for processing. The CPICOMM or LU62 environment itself evaluates the variables and performs variable substitution. If you do not specify a variable for each parameter and enclose the call in quotation marks, you may have problems with variable substitution and receive unexpected results.

As an example, the SAA CPI Communications call, CMINIT, has three parameters; `conversation_id`, `sym_dest_name`, and `return_code`. When you use CMINIT, specify three variables for the three parameters; for example, *convid* for the conversation_id parameter, *symdest* for the sym_dest_name parameter, and *retcode* for the return_code parameter. Before you use CMINIT, you can assign the value you want to use for the sym_dest_name parameter, such as CPINY17.

```
/*  REXX transaction program ...   */
⋮
symdest = 'CPINY17'
⋮
ADDRESS CPICOMM "CMINIT convid symdest retcode"
IF retcode ¬= CM_OK THEN
⋮
ADDRESS CPICOMM "CMALLC convid retcode"
IF retcode = CM_OK THEN
⋮
EXIT
```

In the example, you assign the variable *symdest* the value CPINY17. On the CMINIT call, you use the variable names for the parameters. The CPICOMM host command environment evaluates the variables and uses the value CPINY17 for the sym_dest_name parameter.

When the call returns control to the language processor, the output variables whose names were specified on the call contain the returned values. In this example, the variable "convid" contains the value for the conversation_id parameter and "retcode" contains the value for the return_code parameter.

On return, the REXX special variable RC is also set to one of the following:

- A zero if the service is invoked successfully.
- A -3 if the parameter list was incorrect or if the APPC/MVS call could not be found.

Note that the value that can be set in the REXX special variable RC for the CPICOMM and LU62 environments is a signed 31-bit number in the range -2,147,483,648 to +2,147,483,647.

Pseudonym files

Both the SAA CPI Communications calls and the APPC/MVS calls use pseudonyms for actual calls, characteristics, variables, and so on. For example, the return_code parameter for SAA CPI Communications calls can be the pseudonym CM_OK. The integer value for the CM_OK pseudonym is 0.

APPC/MVS provides pseudonym files in the header file data set SYS1.SIEAHDR.H that define the pseudonyms and corresponding integer values. The pseudonym files APPC/MVS provides are:

- ATBPBREX for APPC/MVS calls
- ATBCMREX for SAA CPI Communications calls.
- ATBCTREX for APPCMVS host command environment pseudonyms.

The sample pseudonym files contain REXX assignments that simplify writing transaction programs (TPs) in REXX. You can copy either the entire pseudonym file or parts of the file into your transaction program.

Alternatively, you can use EXECIO to read each assignment from the pseudonym file and then interpret the line to cause the pseudonym to be defined to your REXX Program.

Transaction program profiles

If you write a transaction program in REXX and you plan to run the program as an inbound TP, you have to create a transaction program (TP) profile for the exec. The profile is required for inbound or attached TPs. The transaction program (TP) profile consists of a set of JCL statements that you store in a TP profile data set on MVS. The following figures provide example JCL for transaction program (TP) profiles. For more information about TP profiles, see *z/OS MVS Planning: APPC/MVS Management*.

Figure 2 on page 29 shows example JCL for an exec that you write for non-TSO/E address spaces.

```
//JOBNAME   JOB  parameters
//STEPNAME EXEC PGM=IRXJCL,PARM='exec_member_name argument'
//SYSPRINT DD   SYSOUT=A
//SYSEXEC  DD   DSN=exec_data_set_name,DISP=SHR
//SYSTSIN  DD   DSN=input_data_set_name,DISP=SHR
//SYSTSPRT DD   DSN=output_data_set_name,DISP=SHR
```

Figure 2. Example JCL for TP profile for a Non-TSO/E REXX exec

Figure 3 on page 29 shows example JCL for an exec that you write for a TSO/E address space.

```
//JOBNAME   JOB  parameters
//STEPNAME EXEC PGM=IKJEFT01,PARM='exec_member_name argument'
//SYSPRINT DD   SYSOUT=A
//SYSEXEC  DD   DSN=exec_data_set_name,DISP=SHR
//SYSTSPRT DD   DSN=output_data_set_name,DISP=SHR
//SYSTSIN  DD   DUMMY
```

Figure 3. Example JCL for TP profile for a TSO/E REXX exec

Sample Transaction programs

APPC/MVS provides sample transaction programs (TPs) written in REXX and related information in SYS1.SAMPLIB. Table 2 on page 29 lists the member names of the samples and their descriptions. For information about using the sample TPs, see the comments at the beginning of the outbound transaction program (TPs) for the particular sample. For the SAA CPI Communications sample, the outbound TP is in member ATBCAO. For the APPC/MVS sample (based on the SNA LU 6.2 architecture), the outbound TP is in member ATBLAO.

Table 2. Sample APPC/MVS transaction programs in SYS1.SAMPLIB

Samplib member	Description
ATBCAJ	JCL to run REXX SAA CPI Communications sample program A
ATBCAP	JCL to add a TP profile for REXX SAA CPI Communications sample program A
ATBCAS	JCL to add side information for REXX SAA CPI Communications sample program A
ATBLAJ	JCL to run REXX APPC/MVS sample program A
ATBLAP	JCL to add a TP profile for REXX APPC/MVS sample program A
ATBCAI	REXX SAA CPI Communications sample program A; inbound REXX transaction program (TP)
ATBCAO	REXX SAA CPI Communications sample program A; outbound REXX transaction program (TP)
ATBCKRC	REXX subroutine to check return codes; used by sample REXX transaction programs (TPs)
ATBLAI	REXX APPC/MVS sample program A; inbound REXX transaction program (TP)
ATBLAO	REXX APPC/MVS sample program A; outbound REXX transaction program (TP)

The MVS host command environment

The MVS host command environment is available in any MVS address space. When you run a REXX exec in a non-TSO/E address space, the default initial host command environment is MVS. When you invoke an exec in a z/OS UNIX address space, SH is the initial host command environment and the programmer has control over the host command environment by setting the value of INITIAL in IRXSUBCT.

Note: When you invoke an exec in a TSO/E address space, TSO is the initial host command environment.

In ADDRESS MVS, you can use a subset of the TSO/E REXX commands as follows:

- DELSTACK
- NEWSTACK
- QSTACK
- QBUF
- QELEM
- EXECIO
- MAKEBUF
- DROPBUF
- SUBCOM
- TS
- TE

Chapter 10, “TSO/E REXX commands,” on page 201 describes the commands.

In ADDRESS MVS, you can also invoke another REXX exec using the ADDRESS MVS EXEC command. Note that this command is not the same as the TSO/E EXEC command processor. You can use one of the following instructions to invoke an exec. The instructions in the following example assume the current host command environment is *not* MVS.

```
ADDRESS MVS "execname p1 p2 ..."

ADDRESS MVS "EX execname p1 p2 ..."

ADDRESS MVS "EXEC execname p1 p2 ..."
```

If you use the ADDRESS MVS EXEC command to invoke another REXX exec, the system searches only the DD from which the calling exec was loaded. If the exec is not found in that DD, the search for the exec ends and the REXX special variable RC is set to -3. Note that the value that can be set in the REXX special variable RC for the MVS environment is a signed 31-bit number in the range -2,147,483,648 to +2,147,483,647.

To invoke an unauthorized program from an exec, use one of the link or attach host command environments that are described in “Host command environments for linking to and attaching programs” on page 30.

All of the services that are available in ADDRESS MVS are also available in ADDRESS TSO. For example, if you run a REXX exec in TSO/E, you can use the TSO/E REXX commands (for example, MAKEBUF, NEWSTACK, QSTACK) in ADDRESS TSO.

Host command environments for linking to and attaching programs

TSO/E provides the LINK, LINKMVS, and LINKPGM host command environments that let you link to unauthorized programs on the same task level.

TSO/E also provides the ATTACH, ATTCHMVS, and ATTCHPGM host command environments that let you attach unauthorized programs on a different task level.

When REXX attaches a program under any of these host command environments, the REXX exec task waits until the attached program completes. (That is, the attach is synchronous, not asynchronous.) Once the attached program returns, the REXX exec resumes at the point after the attach. The REXX exec can examine the REXX special variable RC for the results of the attach, as discussed below.

These link and attach environments are available to REXX execs that run in any address space.

To link to or attach a program, specify the name of the program followed by any parameters you want to pass to the program. For example:

```
ADDRESS LINKMVS "program p1 p2 ... pn"

ADDRESS ATTCHPGM "program p1 p2 ... pn"
```

Enclose the name of the program and any parameters in either single or double quotation marks.

The host command environment routines for the environments use the following search order to locate the program:

- Job pack area
- ISPLLIB. If the user issued *LIBDEF ISPLLIB ...*, the system searches the new alternate library defined by LIBDEF followed by the ISPLLIB library.
- Task library and all preceding task libraries
- Step library. If there is no step library, the job library is searched, if one exists.
- Link pack area (LPA)
- Link library.

The differences between the environments are the format of the parameter list that the program receives, the capability of passing multiple parameters, variable substitution for the parameters, and the ability of the invoked program to update the parameters.

- For the LINK and ATTACH environments, you can specify only a single character string that gets passed to the program. The LINK and ATTACH environments do not evaluate the character string and do not perform variable substitution. The environments simply pass the string to the invoked program. The program can use the character string it receives. However, the program cannot return an updated string to the exec.
- For the LINKMVS, LINKPGM, ATTCHMVS, and ATTCHPGM environments, you can pass multiple parameters to the program. The environments evaluate the parameters you specify and perform variable substitution. That is, the environment determines the value of each variable. When the environment invokes the program, the environment passes the value of each variable to the program. The program can update the parameters it receives and return the updated values to the exec.

After you link to or attach the program, the host command environment sets a return code in the REXX special variable RC. For all of the link and attach environments, the return code may be:

- A -3 if the host command environment could not locate the program you specified
- The return code that the linked or attached program set in register 15

Additionally, for the LINKMVS, ATTCHMVS, LINKPGM, and ATTCHPGM environments, the return code set in RC may be -2, which indicates that processing of the variables was not successful. Variable processing may have been unsuccessful because the host command environment could not:

- Perform variable substitution before linking to or attaching the program
- Update the variables after the program completed

For LINKMVS and ATTCHMVS, you can also receive an RC value of -2 if the length of the value of the variable was larger than the length that could be specified in the signed halfword length field in the parameter list. The maximum value of the halfword length field is 32,767.

Note that the value that can be set in the RC special variable for the LINK, LINKMVS, and LINKPGM environments is a signed 31-bit number in the range -2,147,483,648 to +2,147,483,647. The value that can be set in RC for the ATTACH, ATTCHMVS, and ATTCHPGM environments is a signed 24-bit number in the range -8,388,608 to +8,388,607.

The following topics describe how to link to and attach programs using the different host command environments.

The LINK and ATTACH host command environments

For the LINK and ATTACH environments, you can pass only a single character string to the program. Enclose the name of the program and the character string in either single or double quotation marks to prevent the language processor from performing variable substitution. For example:

```
ADDRESS ATTACH 'TESTPGMA varid'
```

If you want to pass the value of a variable, then it should not be enclosed in quotation marks. In this case the interpreter will perform the variable substitution before passing the string to the host command environment. The following excerpt from a REXX program would have identical results as the previous example:

```
parm_value = 'varid'
ADDRESS ATTACH 'TESTPGMA' parm_value
```

The host command environment routines for LINK and ATTACH do not evaluate the character string you specify. The routine simply passes the character string to the program that it links to or attaches. The program can use the character string it receives. However, the program cannot return an updated string to the exec.

Figure 4 on page 32 shows how the LINK or ATTACH host command environment routine passes a character string to a program. Register 0 points to the ENVBLOCK under which the REXX exec issuing the ADDRESS LINK or ADDRESS ATTACH is running. Register 1 points to a list that consists of two addresses. The first address points to a fullword that contains the address of the character string. The second address points to a fullword that contains the length of the character string. The high- order bit of the last address in the parameter list is set to 1 to indicate the end of the parameter list.

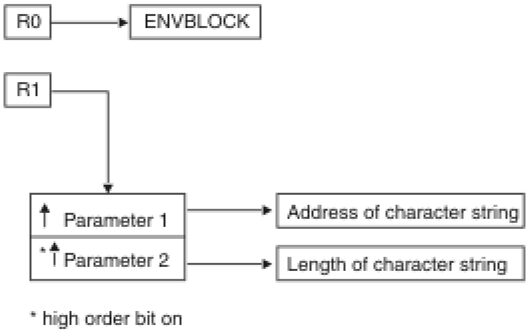


Figure 4. Parameters for LINK and ATTACH environments

For example, suppose you use the following instruction:

```
ADDRESS LINK 'TESMODA numberid payid'
```

When the LINK host command environment routine links to the TESMODA program, the address of the character string points to the string:

```
numberid payid
```

The length of the character string is 14. In this example, if *numberid* and *payid* were REXX variables, no substitution is performed by the LINK host command environment.

You can use the LINK or ATTACH environments and not specify a character string. For example:

```
ADDRESS ATTACH "proga"
```

In this case, the address of the character string is 0 and the length of the string is 0.

The LINKMVS and ATTCHMVS host command environments

For the LINKMVS and ATTCHMVS environments, you can pass multiple parameters to the program. Specify the name of the program followed by variable names for each of the parameters. Separate each variable name by one or more blanks. For example:

```
ADDRESS ATTCHMVS 'TESTPGMA var1 var2 var3'
```

For the parameters, specify variable names instead of the actual values. Enclose the name of the program and the variable names in either single or double quotation marks. By using the quotation marks, the language processor does not evaluate any variables. The language processor simply passes the expression to the host command environment for processing. The LINKMVS or ATTCHMVS environment itself evaluates the variables and performs variable substitution. If you do not use a variable for each parameter and enclose the expression in quotation marks, you may have problems with variable substitution and receive unexpected results.

After the LINKMVS or ATTCHMVS environment routine evaluates the value of each variable, it builds a parameter list pointing to the values. The routine then links to or attaches the program and passes the parameter list to the program.

Figure 5 on page 33 shows how the LINKMVS or ATTCHMVS host command environment routine passes the parameters to the program. Register 0 points to the ENVBLOCK under which the REXX exec issuing the ADDRESS LINKMVS or ADDRESS ATTCHMVS is running. Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list is set to 1 to indicate the end of the parameter list.

Each parameter consists of a halfword length field followed by the parameter, which is the value of the variable you specified on the LINKMVS or ATTCHMVS instruction. The halfword length field contains the length of the parameter, which is the length of the value of the variable. The maximum value of the halfword length field is 32,767.

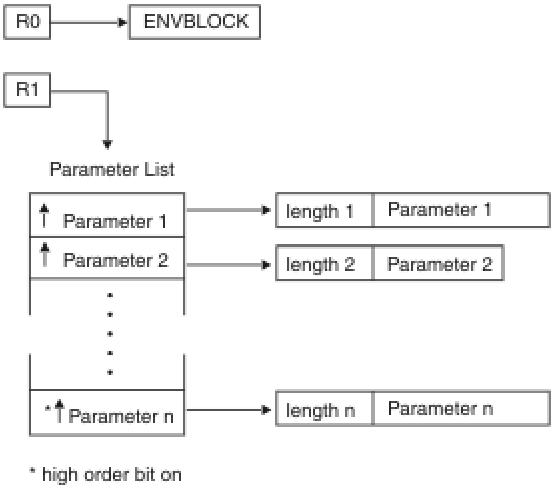


Figure 5. Parameters for LINKMVS and ATTCHMVS environments

As an example, suppose you want to attach the RTNWORK program and you want to pass two parameters; an order number (43176) and a code (CDETT76). When you use the ADDRESS ATTCHMVS instruction, specify variable names for the two parameters; for example, *ordernum* for the order number, 43176, and *codenum* for the code, CDETT76. Before you use ADDRESS ATTCHMVS, assign the values to the variable names.

```
⋮
ordernum = 43176
codenum = "CDETT76"
⋮
ADDRESS ATTCHMVS "RTNWORK ordernum codenum"
```

```
⋮
EXIT
```

In the example, you assign to the variable *ordernum* the value 43176 and you assign to the variable *codenum* the value CDETT76. On the ADDRESS ATTCHMVS instruction, you use the variable names for the two parameters. The ATTCHMVS host command environment evaluates the variables and passes the values of the variables to the RTNWORK program. In the parameter list, the length field for the first parameter (variable *ordernum*) is 5, followed by the character string 43176. The length field for the second parameter (variable *codenum*) is 7, followed by the character string CDETT76.

On entry to the linked or attached program, the halfword length fields contain the actual length of the parameters. The linked or attached program can update the values of the parameters before it completes processing. The value that the program returns in the halfword length field determines the type of processing that LINKMVS or ATTCHMVS performs.

When the LINKMVS or ATTCHMVS environment routine regains control, it determines whether to update the values of the REXX variables before returning to the REXX exec. To determine whether to update the value of a variable for a specific parameter, the LINKMVS or ATTCHMVS environment checks the value in the halfword length field. Depending on the value in the length field, LINKMVS or ATTCHMVS updates the variable, does not update the variable, or sets the variable to the null string.

- If the value in the length field is less than 0, the LINKMVS or ATTCHMVS environment does not update the variable for that parameter.
- If the value in the length field is 0, the LINKMVS or ATTCHMVS environment sets the variable for that parameter to the null string.
- If the value in the length field is greater than 0, the LINKMVS or ATTCHMVS environment updates the variable for that parameter with the value the program returned in the parameter list. If the length field is a positive number, LINKMVS or ATTCHMVS simply updates the variable using the length in the length field.

  If the original length of the value is less than 500 bytes, TSO/E provides a storage area of 500 bytes regardless of the length of the value of the variable. For example, if the length of the value of the variable on entry to the program were 8 bytes, the halfword length field would contain the value 8. However, there are 500 bytes of storage available for the parameter itself. This allows the program to increase the length of the variable without having to obtain storage. If the invoked program changes the length of the variable, it must also update the length field.

  If the original length of the value is 500 or more bytes, there is no additional space allotted. For example, suppose you specify a variable whose value has a length of 620 bytes. The invoked program can return a value with a maximum length of 620 bytes. TSO/E does not provide an additional buffer area. In this case, if you expect that the linked or attached program may want to return a larger value, pad the original value to the right with blanks.

As an example, suppose you link to a program called PGMCODES and pass a variable *pcode* that has the value PC7177. The LINKMVS environment evaluates the value of the variable *pcode* (PC7177) and builds a parameter list pointing to the value. The halfword length field contains the length of the value, which is 6, followed by the value itself. Suppose the PGMCODES program updates the PC7177 value to the value PC7177ADC3. When the PGMCODES program returns control to the LINKMVS environment, the program must update the length value in the halfword length field to 10 to indicate the actual length of the value it is returning to the exec.

You can use the LINKMVS or ATTCHMVS environments and not specify any parameters. For example:

```
ADDRESS ATTCHMVS 'workpgm'
```

If you do not specify any parameters, register 1 contains an address that points to a parameter list. The high-order bit is on in the first parameter address. The parameter address points to a parameter that has a length of 0.

An example using LINKMVS to specify user-defined Ddnames

In this example the user had the need to specify user-defined ddnames, instead of using SYSUT1 and SYSUT2, for an invocation of IEBGENER, an MVS data set utility program.

```
/* Rexx - Invoke IEBGENER with alternate ddnames.   */
prog   = 'IEBGENER'
parm   = ''                      /* Standard PARM, as from JCL    */
ddlist = copies('00'x,8) ||,      /* DDname  1 override: SYSLIN    */
         copies('00'x,8) ||,      /* DDname  2 override:  n/a      */
         copies('00'x,8) ||,      /* DDname  3 override: SYSLMOD   */
         copies('00'x,8) ||,      /* DDname  4 override: SYSLIB    */
         left('CTL',  8) ||,      /* DDname  5 override: SYSIN     */
         left('REP',  8) ||,      /* DDname  6 override: SYSPRINT  */
         copies('00'x,8) ||,      /* DDname  7 override: SYSPUNCH  */
         left('INP',  8) ||,      /* DDname  8 override: SYSUT1    */
         left('OUT',  8) ||,      /* DDname  9 override: SYSUT2    */
         copies('00'x,8) ||,      /* DDname 10 override: SYSUT3    */
         copies('00'x,8) ||,      /* DDname 11 override: SYSUT4    */
         copies('00'x,8) ||,      /* DDname 12 override: SYSTERM   */
         copies('00'x,8) ||,      /* DDname 13 override:  n/a      */
         copies('00'x,8)          /* DDname 14 override: SYSCIN    */

address 'LINKMVS' prog 'PARM DDLIST'
exit rc
```

The program to be invoked is specified in variable PROG as IEBGENER, the parameters for the IEBGENER program are specified in variables PARM and DDLIST. CTL, REP, INP, and OUT are the replaced ddnames.

Examples of using LINKMVS, ATTCHMVS, and CALL to invoke a program

The LINKMVS and ATTCHMVS address environments can be used to invoke a program normally invoked as a batch job, as can the TSO CALL command. Table 3 on page 35 shows various examples of invoking program MYWTO while passing a single parameter or while passing two parameters to the program. Note the varying effects of specifying the *msg* variable, either within quotation marks or not within quotation marks, on the LINKMVS (or ATTCHMVS) invocation.

Examples 1, 2, 3, 4, and 6 show several different ways of invoking MYWTO while passing a single 9-character parameter string: **A MESSAGE**. Example 5 is similar to examples 1–4, and 6, however, an 11-character string **'A MESSAGE'** is passed. The leading and trailing single quotation markers part of the string that is passed.

Example 7 shows the passing of two parameters to the program, the value of variable *A* and the value of the variable *MESSAGE*. However, because these two variables were not initialized before the invocation, each variable has a value equal to its own name, in uppercase. The end result is that two parameter strings are passed to the program. The first parameter is the 1-character string **A**, and the second parameter is the 7-character string **MESSAGE**. In contrast, in example 9, two parameter strings are passed to the program, the value of variable *A* (namely, **Value1**) and the value of variable *MESSAGE* (namely, **Value 2**).

Example 8 is similar to example 6, in that both pass the same string **A MESSAGE**. In example 8, the fact that variables *A* and *MESSAGE* have values before the invocation is irrelevant, since the parameter string **A MESSAGE** is passed, not the values of variables *A* and *MESSAGE*.

Table 3. Examples of using LINKMVS, ATTCHMVS, and CALL to invoke a program

Example Number	Invocation from	Invocation Method Used	Example Invocation Implementation
1	JCL	EXEC PGM=	//* Call MYWTO, passing **A MESSAGE***/ /stepname EXEC PGM=MYWTO,PARM='A MESSAGE'
2	TSO CLIST	CALL command	/* Call MYWTO, passing **A MESSAGE** */ CALL *(MYWTO) 'A MESSAGE'

Table 3. Examples of using LINKMVS, ATTCHMVS, and CALL to invoke a program (continued)

Example Number	Invocation from	Invocation Method Used	Example Invocation Implementation
3	TSO REXX	CALL command	/* Call MYWTO, passing **A MESSAGE** */ Address TSO "CALL *(MYWTO) 'A MESSAGE'"
4	TSO REXX	CALL command	/* REXX */ /* Call MYWTO, passing **A MESSAGE** */ *msg* = "'A MESSAGE'" Address TSO "CALL *(MYWTO)" *msg*
5	TSO REXX	LINKMVS or ATTCHMVS host command environment	/* REXX */ /* Call MYWTO, passing **'A MESSAGE'** */ *msg* = "'A MESSAGE'" /* single quotes part of value passed */ Address LINKMVS "MYWTO *msg*" **Guideline**: The variable name *msg* must be inside the quotation marks in order to pass a single parameter with the value **'A MESSAGE'** to the called program.
6	TSO REXX	LINKMVS or ATTCHMVS host command environment	/* REXX */ /* Call MYWTO, passing **A MESSAGE** */ *msg* = "A MESSAGE" Address LINKMVS "MYWTO *msg*" /* msg inside quotation marks */ **Guideline**: The variable name *msg* must be inside the quotation marks in order to pass a single parameter with the value **A MESSAGE** to the called program.
7	TSO REXX	LINKMVS or ATTCHMVS host command environment	/* REXX */ /* Call MYWTO, passing two parameters */ /* - value of the variable *A* */ /* - value of the variable *MESSAGE* */ *msg* = "A MESSAGE" Address LINKMVS "MYWTO" *msg* /* msg outside quotes */ **Guideline**: The variable name *msg* is outside the double quotation marks, so the statement is equivalent to: Address LINKMVS "MYWTO A MESSAGE" The values of variables A and MESSAGE are passed to the called routine. Since these variables are uninitialized, their respective values are **A** and **MESSAGE**. Program MYWTO therefore is passed two parameters, the first with the value **A** and the second with the value **MESSAGE**.

Table 3. Examples of using LINKMVS, ATTCHMVS, and CALL to invoke a program (continued)

Example Number	Invocation from	Invocation Method Used	Example Invocation Implementation
8	TSO REXX	LINKMVS or ATTCHMVS host command environment	/* REXX */ /* Call MYWTO, passing **A MESSAGE** */ *A* = "Value1" *MESSAGE* = "Value2" *msg* = "A MESSAGE" Address LINKMVS "MYWTO *msg*" /* msg inside quotation marks */ **Guideline**: The variable name *msg* must be inside the quotation marks in order to pass a single parameter with the value **A MESSAGE** to the called program; the values of variables *A* and *MESSAGE* are not used in this example.
9	TSO REXX	LINKMVS or ATTCHMVS host command environment	/* REXX */ /* Call MYWTO, passing two parameters */ /* - value of the variable *A* */ /* - value of the variable *MESSAGE* */ *A* = "Value1" *MESSAGE* = "Value2" *msg* = "A MESSAGE" Address LINKMVS "MYWTO" *msg* /* msg outside quotation marks */ **Guideline**: The variable name *msg* is outside the double quotation marks, so the statement is equivalent to: Address LINKMVS "MYWTO A MESSAGE" The values of variables *A* and *MESSAGE* are passed to the called routine. Program MYWTO therefore is passed two parameters, the first with the value **Value1** and the second with the value **Value 2**.

The LINKPGM and ATTCHPGM host command environments

For the LINKPGM and ATTCHPGM environments, you can pass multiple parameters to the program. Specify the name of the program followed by variable names for each of the parameters. Separate each variable name by one or more blanks. For example:

```
ADDRESS LINKPGM "WKSTATS var1 var2"
```

For the parameters, specify variable names instead of the actual values. Enclose the name of the program and the variable names in either single or double quotation marks. By using the quotation marks, the language processor does not evaluate any variables and simply passes the expression to the host command environment for processing. The LINKPGM or ATTCHPGM environment itself evaluates the variables and performs variable substitution. If you do not use a variable for each parameter and enclose the expression in quotation marks, you may have problems with variable substitution and receive unexpected results.

After the LINKPGM or ATTCHPGM environment routine evaluates the value of each variable, it builds a parameter list pointing to the values. The routine then links to or attaches the program and passes the parameter list to the program.

Figure 6 on page 38 shows how the LINKPGM or ATTCHPGM host command environment routine passes the parameters to the program. Register 0 points to the ENVBLOCK under which the REXX exec issuing the ADDRESS LINKPGM or ADDRESS ATTCHPGM is running. Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list is set to 1 to indicate the end of the parameter list.

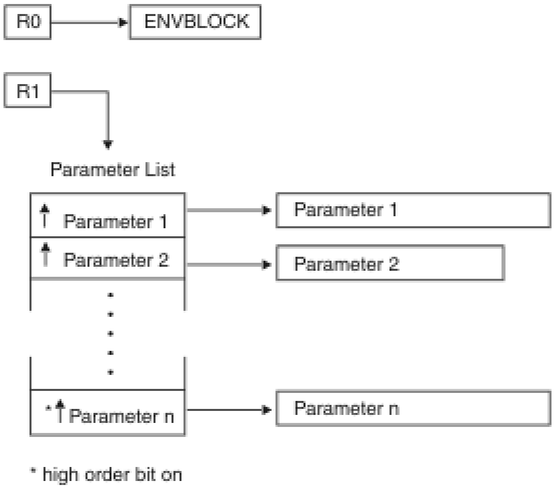


Figure 6. Parameters for LINKPGM and ATTCHPGM environments

Unlike the LINKMVS and ATTCHMVS host command environments, the parameters for the LINKPGM and ATTCHPGM environments do not have a length field. On output from the linked or attached routine, the value of the parameter is updated and the length of each parameter is considered to be the same as when the parameter list was created. The linked or attached routine cannot increase the length of the value of a variable that it receives. However, you can pad the length of the value of a variable with blanks to increase its length before you link to or attach a program.

As an example, suppose you want to link to the RESLINE program and you want to pass one parameter, a reservation code of WK007816. When you use the ADDRESS LINKPGM instruction, specify a variable name for the parameter; for example, *revcode* for the reservation code, WK007816. Before you use ADDRESS LINKPGM, assign the value to the variable name.

```
/*  REXX program that links ...    */
⋮
revcode = 'WK007816'
⋮
ADDRESS LINKPGM 'RESLINE revcode'
⋮
EXIT
```

In the example, you assign the variable *revcode* the value WK007816. On the ADDRESS LINKPGM instruction, you use the variable name for the parameter. The LINKPGM host command environment evaluates the variable and passes the value of the variable to the RESLINE program. The length of the parameter (variable *revcode*) is 8. If the RESLINE program wanted to update the value of the variable and return the updated value to the REXX exec, the program could not return a value that is greater than 8 bytes. To allow the linked program to return a larger value, you could pad the value of the original variable to the right with blanks. For example, in the exec you could add seven blanks and assign the value "WK007816 " to the *revcode* variable. The length would then be 15 and the linked program could return an updated value that was up to 15 bytes.

You can use the LINKPGM or ATTCHPGM environments and not specify any parameters. For example:

```
ADDRESS ATTCHPGM "monbill"
```

If you do not specify any parameters, register 1 contains an address that points to a parameter list. The high-order bit is on in the first parameter address, but the address itself is 0.

BCPii host command environment

The Base Control Program internal interface (BCPii) host command environment is available to REXX execs that run in the TSO/E address space. Use the BCPii host command environment to invoke the BCPii callable services. BCPii is an authorized z/OS application which can retrieve, change, and perform operational procedures against the installed System z® hardware. Since the BCPii API suite is limited to program-authorized callers, and TSO/E runs in non-program-authorized environment, an installation must choose whether to allow TSO users access to the BCPii host command environment. Even if this authority is granted, the TSO REXX exec issuing the BCPii host commands still require the proper SAF authorization for it to perform BCPii functions. For information about TSO/E setup with BCPii, see the section "BCPii setup and installation" in the chapter for Base Control Program internal interface service (BCPii) in *z/OS MVS Programming: Callable Services for High-Level Languages*.

When a REXX exec is invoked from TSO, the default initial host command environment is TSO. You can use the ADDRESS instruction to use a BCPii service. For example, to use the BCPii HWICONN service, use the following instruction:

```
ADDRESS BCPii "HWICONN parm1 parm2 ..."
```

For more information about the specific BCPii calls supported and their parameters, see the chapter for Base Control Program internal interface service (BCPii) in *z/OS MVS Programming: Callable Services for High-Level Languages*

OUTTRAP versus MSG function when trapping or suppressing output

TPUT and WTOs are not affected by MSG and OUTTRAP. MSG and OUTTRAP only affect messages that are issued by PUTLINE / PUTGET. Some PUTLINE messages might be suppressed by MSG('OFF') or might be trapped by OUTTRAP. PUTGET mode messages are generally not affected by the MSG setting, but some might be trapped by OUTTRAP, such as those for ACCOUNT or EDIT, and so on.

MSG('ON'|'OFF') affects only the display of Informational messages. Informational Message refers to a message that is issued by PUTLINE with the INFO keyword indicating that the message is an informational message. It is not always obvious from the message itself whether the message qualifies because you might not know how it was issued. But generally, TSO Command error messages are issued as INFO messages. However, most REXX messages, output from SAY statements, and TRACE output is usually issued as PUTLINE DATA output, therefore, MSG('OFF') would have no effect.

OUTTRAP can trap both INFO and DATA type of PUTLINE output. For example, the output from the "PROFILE LIST" is IKJ56688I, which can be suppressed by MSG('OFF') or can be trapped by OUTTRAP.

OUTTRAP cannot be used to trap REXX error message output that occurs during interpretation of the exec, or TRACE output or any IRXxxx message output that is issued by the interpreter for the executing exec that issued OUTTRAP. Also, OUTTRAP cannot trap SAY output from within the exec that issued OUTTRAP. However, if exec1 invokes exec2 (as a cmd of subroutine) and exec 2 issues SAY output, or exec2 generates TRACE output, the calling exec can trap that information. In addition, if an error occurs in exec2 causing the exec to issue an IRXxxx error message, that error can usually be trapped by the calling exec if TRAPMSG('ON') was set by exec1 before exec2 was invoked. Also, when a REXX exec issues a REXX command like EXECIO, or invokes another REXX function, IRXxxx messages that are issued from the command of function can usually be trapped if TRAPMSG('ON') was activated in the calling exec.

PUTLINE DATA output from some commands, such as "LISTA ST HI" can be trapped by OUTTRAP, but they cannot be suppressed by MSG('OFF'), since these output lines are usually issued as PUTLINE DATA lines.

In some cases, data can either be suppressed by MSG('OFF') or trapped by OUTTRAP. For example, the output IKJ56688I from "PROFILE LIST" can be either suppressed or trapped. However, if MSG('OFF') and OUTTRAP are both active, the message is seen by MSG('OFF') processing and suppressed before it can be seen and trapped by OUTTRAP.

Certain other output might be issued as WTO/WTP messages to the terminal. PROFILE NOWTPMSG can be used to suppress certain WTP/WTO type output.

Generally speaking, REXX OUTTRAP corresponds to CLIST &SYSOUTTRAP, and MSG('ON'/'OFF') corresponds to CLIST &SYSMSG (for example, CONTROL MSG/NOMSG). For more information about &SYSOUTTRAP and &SYSMSG usage, see *z/OS TSO/E CLISTs*.

If you have an exec that invokes an unknown command, IKJ56500I COMMAND XXXXXXXX NOT FOUND is issued. REXX also issues RC(-3) indicating that the command was not found. The IKJ56500I can be suppressed by MSG('OFF') but cannot be trapped by OUTTRAP.

Chapter 3. Keyword instructions

A **keyword instruction** is one or more clauses, the first of which starts with a keyword that identifies the instruction. Some keyword instructions affect the flow of control, while others provide services to the programmer. Some keyword instructions, like DO, can include nested instructions.

In the syntax diagrams on the following pages, symbols (words) in capitals denote keywords or subkeywords; other words (such as *expression*) denote a collection of tokens as defined previously. Note, however, that the keywords and subkeywords are not case-dependent; the symbols `if`, `If`, and `iF` all have the same effect. Note also that you can usually omit most of the clause delimiters (`;`) shown because they are implied by the end of a line.

A keyword instruction is recognized *only* if its keyword is the first token in a clause, and if the second token does not start with an = character (implying an assignment) or a colon (implying a label). The keywords ELSE, END, OTHERWISE, THEN, and WHEN are recognized in the same situation. Note that any clause that starts with a keyword defined by REXX cannot be a command. Therefore,

```
arg(fred) rest
```

is an ARG keyword instruction, not a command that starts with a call to the ARG built-in function. A syntax error results if the keywords are not in their correct positions in a DO, IF, or SELECT instruction. (The keyword THEN is also recognized in the body of an IF or WHEN clause.) In other contexts, keywords are not reserved and can be used as labels or as the names of variables (although this is generally not suggested).

Certain other keywords, known as subkeywords, are reserved within the clauses of individual instructions. See “Keyword instructions” on page 18. For example, the symbols VALUE and WITH are subkeywords in the ADDRESS and PARSE instructions, respectively. For details, see the description of each instruction. For a general discussion on reserved keywords, see “Reserved keywords” on page 197.

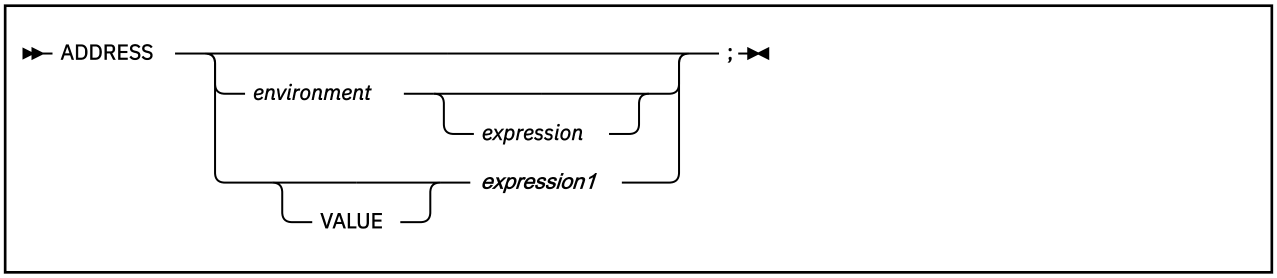
Blanks adjacent to keywords have no effect other than to separate the keyword from the subsequent token. One or more blanks following VALUE are required to separate the *expression* from the subkeyword in the example following:

```
ADDRESS VALUE expression
```

However, no blank is required after the VALUE subkeyword in the following example, although it would add to the readability:

```
ADDRESS VALUE'ENVIR'||number
```

ADDRESS



ADDRESS temporarily or permanently changes the destination of commands. Commands are strings sent to an external environment. You can send commands by specifying clauses consisting of only an expression or by using the ADDRESS instruction.

How to enter commands to the host and the different host command environments TSO/E provides are described in "Commands to external environments" on page 22.

To send a single command to a specified environment, code an *environment*, a literal string or a single symbol, which is taken to be a constant, followed by an *expression*. (The environment name is the name of an external procedure or process that can process commands.) The *expression* is evaluated, and the resulting string is routed to the *environment* to be processed as a command. (Enclose in quotation marks any part of the expression you do not want to be evaluated.) After execution of the command, *environment* is set back to whatever it was before, thus temporarily changing the destination for a single command. The special variable RC is set, just as it would be for other commands. (See "Commands" on page 22.) Errors and failures in commands processed in this way are trapped or traced as usual.

Example:

```
ADDRESS LINK "routine p1 p2"       /*   TSO/E   */
```

If you specify only *environment*, a lasting change of destination occurs: all commands that follow (clauses that are neither REXX instructions nor assignment instructions) are routed to the specified command environment, until the next ADDRESS instruction is processed. The previously selected environment is saved.

Example:

```
Address MVS
"QBUF"
"MAKEBUF"
```

Similarly, you can use the VALUE form to make a lasting change to the environment. Here *expression1* (which may be simply a variable name) is evaluated, and the result forms the name of the environment. You can omit the subkeyword VALUE if *expression1* does not begin with a literal string or symbol (that is, if it starts with a special character, such as an operator character or parenthesis).

Example:

```
ADDRESS ('ENVIR'||number)  /* Same as ADDRESS VALUE 'ENVIR'||number */
```

With no arguments, commands are routed back to the environment that was selected before the previous lasting change of environment was made, and the current environment name is saved. After changing the environment, repeated execution of ADDRESS alone, therefore, switches the command destination between two environments alternately.

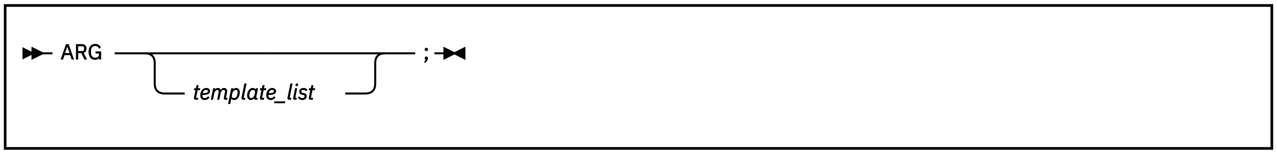
The two environment names are automatically saved across internal and external subroutine and function calls. See "CALL" on page 43) for more details.

The address setting is the currently selected environment name. You can retrieve the current address setting by using the ADDRESS built-in function (see "ADDRESS" on page 79).

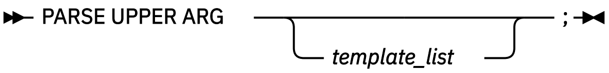
TSO/E REXX provides several host command environments that you can use with the ADDRESS instruction. The environments allow you to use different TSO/E, MVS, and ISPF services. See "Host commands and host command environments" on page 23.

You can provide your own environments or routines that handle command processing in each environment. For more information, see "Host command environment table" on page 331.

ARG



ARG retrieves the argument strings provided to a program or internal routine and assigns them to variables. It is a short form of the instruction:



The *template_list* is often a single template but can be several templates separated by commas. If specified, each template is a list of symbols separated by blanks or patterns or both.

Unless a subroutine or internal function is being processed, the strings passed as parameters to the program are parsed into variables according to the rules described in Chapter 5, “Parsing,” on page 157.

If a subroutine or internal function is being processed, the data used will be the argument strings that the caller passes to the routine.

In either case, the language processor translates the passed strings to uppercase (that is, lowercase a–z to uppercase A–Z) before processing them. Use the PARSE ARG instruction if you do not want uppercase translation.

You can use the ARG and PARSE ARG instructions repeatedly on the same source string or strings (typically with different templates). The source string does not change. The only restrictions on the length or content of the data parsed are those the caller imposes.

Example:

```
/* String passed is "Easy Rider"  */

Arg adjective noun .

/* Now:  ADJECTIVE  contains 'EASY'           */
/*       NOUN       contains 'RIDER'          */
```

If you expect more than one string to be available to the program or routine, you can use a comma in the parsing *template_list* so each template is selected in turn.

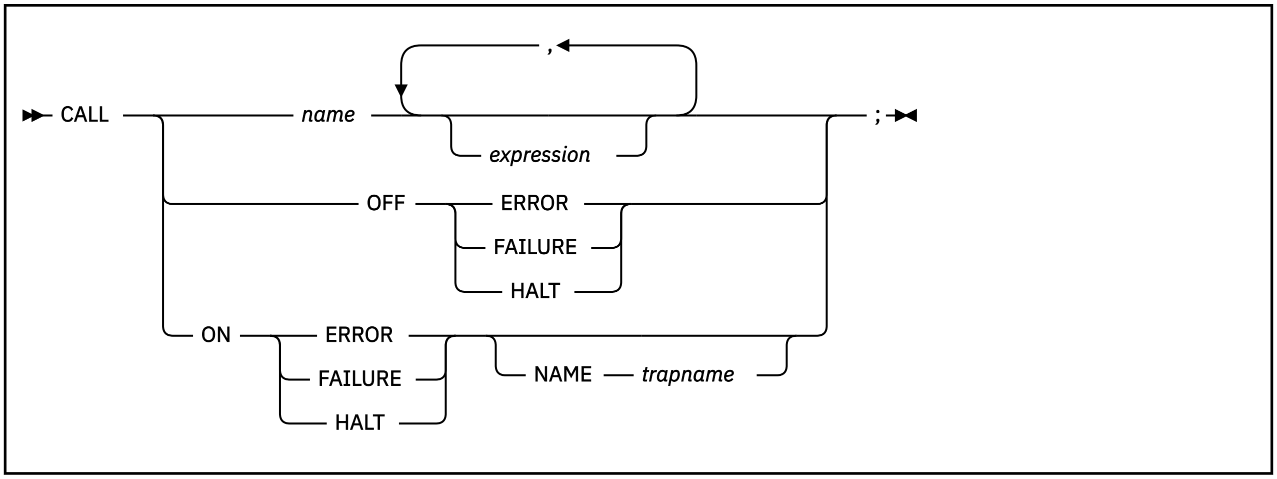
Example:

```
/* Function is called by  FRED('data X',1,5)    */

Fred:  Arg string, num1, num2

/* Now:   STRING  contains 'DATA X'             */
/*        NUM1    contains '1'                  */
/*        NUM2    contains '5'                  */
```

Note:

1. The ARG built-in function can also retrieve or check the argument strings to a REXX program or internal routine. See “ARG (Argument)” on page 79.
2. The source of the data being processed is also made available on entry to the program. See “PARSE” on page 57 for details.

CALL



CALL calls a routine (if you specify *name*) or controls the trapping of certain conditions (if you specify ON or OFF).

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information about condition traps is contained in Chapter 7, “Conditions and condition traps,” on page 181.

To call a routine, specify *name*, a literal string or symbol that is taken as a constant. The *name* must be a symbol, which is treated literally, or a literal string. The routine called can be:

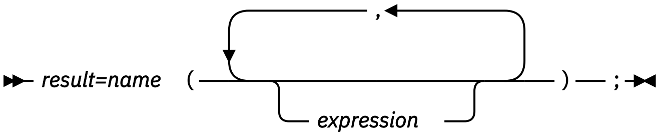
An internal routine
A function or subroutine that is in the same program as the CALL instruction or function call that calls it.

A built-in routine
A function (which may be called as a subroutine) that is defined as part of the REXX language.

An external routine
A function or subroutine that is neither built-in nor in the same program as the CALL instruction or function call that calls it.

If *name* is a string (that is, you specify *name* in quotation marks), the search for internal routines is bypassed, and only a built-in function or an external routine is called. Note that the names of built-in functions (and generally the names of external routines, too) are in uppercase; therefore, you should uppercase the name in the literal string.

The called routine can optionally return a result, and when it does, the CALL instruction is functionally identical with the clause:



If the called routine does not return a result, then you will get an error if you call it as a function (as previously shown).

If the subroutine returns a result, the result is stored in the REXX special variable RESULT, not the special variable RC. The REXX special variable RC is set when you enter host commands from a REXX program (see “Host commands and host command environments” on page 23), but RC is not set when you use the CALL instruction. See Chapter 9, “Reserved keywords, special variables, and command names,” on page 197 for descriptions of the three REXX special variables RESULT, RC, and SIGL.

TSO/E supports specifying up to 20 expressions, separated by commas. The *expression*s are evaluated in order from left to right and form the argument strings during execution of the routine. Any ARG or PARSE ARG instruction or ARG built-in function in the called routine accesses these strings rather than any previously active in the calling program, until control returns to the CALL instruction. You can omit expressions, if appropriate, by including extra commas.

The CALL then causes a branch to the routine called *name*, using exactly the same mechanism as function calls. (See Chapter 4, “Functions,” on page 73.) The search order is in the section on functions (see “Search order” on page 74) but briefly is as follows:

Internal routines:

These are sequences of instructions inside the same program, starting at the label that matches *name* in the CALL instruction. If you specify the routine name in quotation marks, then an internal routine is not considered for that search order. You can use SIGNAL and CALL together to call an internal routine whose name is determined at the time of execution; this is known as a multi-way call (see “SIGNAL” on page 66). The RETURN instruction completes the execution of an internal routine.

Built-in routines:

These are routines built into the language processor for providing various functions. They always return a string that is the result of the routine (see “Built-in functions” on page 77).

External routines:

Users can write or use routines that are external to the language processor and the calling program. You can code an external routine in REXX or in any language that supports the system-dependent interfaces. For information about using the system-dependent interfaces, see “External functions and subroutines, and function packages” on page 263. For information about the search order the system uses to locate external routines, see “Search order” on page 74. If the CALL instruction calls an external routine written in REXX as a subroutine, you can retrieve any argument strings with the ARG or PARSE ARG instructions or the ARG built-in function.

During execution of an internal routine, all variables previously known are generally accessible. However, the PROCEDURE instruction can set up a local variables environment to protect the subroutine and caller from each other. The EXPOSE option on the PROCEDURE instruction can expose selected variables to a routine.

Calling an external program as a subroutine is similar to calling an internal routine. The external routine, however, is an implicit PROCEDURE in that all the caller's variables are always hidden. The status of internal values (NUMERIC settings, and so forth) start with their defaults (rather than inheriting those of the caller). In addition, you can use EXIT to return from the routine.

When control reaches an internal routine the line number of the CALL instruction is available in the variable SIGL (in the caller's variable environment). This may be used as a debug aid, as it is, therefore, possible to find out how control reached a routine. Note that if the internal routine uses the PROCEDURE instruction, then it needs to EXPOSE SIGL to get access to the line number of the CALL.

Eventually the subroutine should process a RETURN instruction, and at that point control returns to the clause following the original CALL. If the RETURN instruction specified an expression, the variable RESULT is set to the value of that expression. Otherwise, the variable RESULT is dropped (becomes uninitialized).

An internal routine can include calls to other internal routines, as well as recursive calls to itself.

Example:

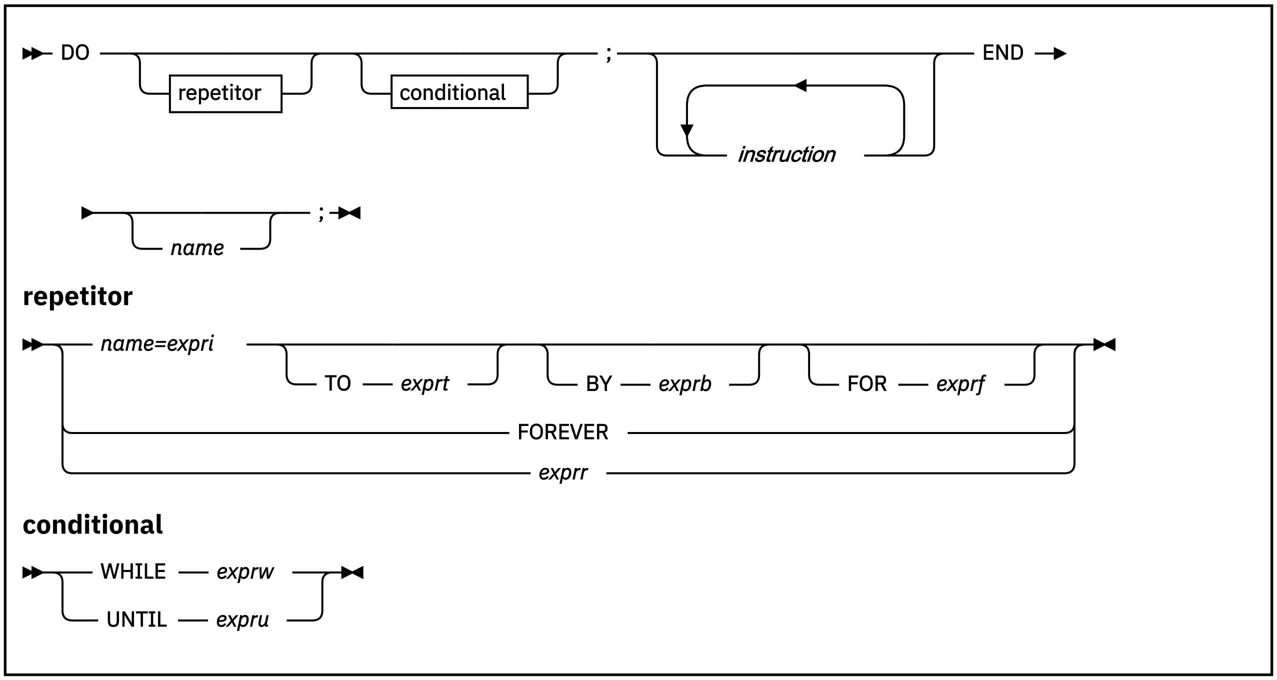
```
/* Recursive subroutine execution... */
arg z
call factorial z
say z'! =' result
exit

factorial: procedure     /* Calculate factorial by  */
  arg n                  /*  recursive invocation.  */
  if n=0 then return 1
  call factorial n-1
  return  result * n
```

During internal subroutine (and function) execution, all important pieces of information are automatically saved and are then restored upon return from the routine. These are:

- **The status of DO loops and other structures**: Executing a SIGNAL while within a subroutine is safe because DO loops, and so forth, that were active when the subroutine was called are not ended. (But those currently active within the subroutine are ended.)
- **Trace action**: After a subroutine is debugged, you can insert a TRACE Off at the beginning of it, and this does not affect the tracing of the caller. Conversely, if you simply wish to debug a subroutine, you can insert a TRACE Results at the start and tracing is automatically restored to the conditions at entry (for example, Off) upon return. Similarly, ? (interactive debug) and ! (command inhibition) are saved across routines.
- **NUMERIC settings**: The DIGITS, FUZZ, and FORM of arithmetic operations (in “NUMERIC” on page 55) are saved and are then restored on return. A subroutine can, therefore, set the precision, and so forth, that it needs to use without affecting the caller.
- **ADDRESS settings**: The current and previous destinations for commands (see “ADDRESS” on page 41) are saved and are then restored on return.
- **Condition traps**: (CALL ON and SIGNAL ON) are saved and then restored on return. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions the caller set up.
- **Condition information**: This information describes the state and origin of the current trapped condition. The CONDITION built-in function returns this information. See “CONDITION” on page 82.
- **Elapsed-time clocks**: A subroutine inherits the elapsed-time clock from its caller (see “TIME” on page 101), but because the time clock is saved across routine calls, a subroutine or internal function can independently restart and use the clock without affecting its caller. For the same reason, a clock started within an internal routine is not available to the caller.
- **OPTIONS settings**: ETMODE and EXMODE are saved and are then restored on return. For more information, see “OPTIONS” on page 56.

Implementation maximum: The total nesting of control structures, which includes internal routine calls, may not exceed a depth of 250.

DO



DO groups instructions together and optionally processes them repetitively. During repetitive execution, a control variable (*name*) can be stepped through some range of values.

Syntax Notes:

- The *exprr*, *expri*, *exprb*, *exprt*, and *exprf* options (if present) are any expressions that evaluate to a number. The *exprr* and *exprf* options are further restricted to result in a positive whole number or zero. If necessary, the numbers are rounded according to the setting of NUMERIC DIGITS.
- The *exprw* or *expru* options (if present) can be any expression that evaluates to 1 or 0.
- The TO, BY, and FOR phrases can be in any order, if used, and are evaluated in the order in which they are written.
- The *instruction* can be any instruction, including assignments, commands, and keyword instructions (including any of the more complex constructs such as IF, SELECT, and the DO instruction itself).
- The subkeywords WHILE and UNTIL are reserved within a DO instruction, in that they cannot be used as symbols in any of the expressions. Similarly, TO, BY, and FOR cannot be used in *expri*, *exprt*, *exprb*, or *exprf*. FOREVER is also reserved, but only if it immediately follows the keyword DO and an equal sign does not follow it.
- The *exprb* option defaults to 1, if relevant.

Simple DO group

If you specify neither *repetitor* nor *conditional*, the construct merely groups a number of instructions together. These are processed one time.

In the following example, the instructions are processed one time.

Example:

```
/* The two instructions between DO and END are both  */
/* processed if A has the value "3".                 */
If a=3 then Do
              a=a+2
              Say 'Smile!'
            End
```

Repetitive DO loops

If a DO instruction has a repetitor phrase or a conditional phrase or both, the group of instructions forms a **repetitive DO loop**. The instructions are processed according to the repetitor phrase, optionally modified by the conditional phrase. (See “Conditional phrases (WHILE and UNTIL)” on page 49).

Simple repetitive loops

A simple repetitive loop is a repetitive DO loop in which the repetitor phrase is an expression that evaluates to a count of the iterations.

If *repetitor* is omitted but there is a *conditional* or if the *repetitor* is FOREVER, the group of instructions is nominally processed "forever", that is, until the condition is satisfied or a REXX instruction is processed that ends the loop (for example, LEAVE).

For a discussion on conditional phrases, see “Conditional phrases (WHILE and UNTIL)” on page 49.

In the simple form of a repetitive loop, *exprr* is evaluated immediately (and must result in a positive whole number or zero), and the loop is then processed that many times.

Example:

```
/* This displays "Hello" five times */
Do 5
  say 'Hello'
  end
```

Note that, similar to the distinction between a command and an assignment, if the first token of *exprr* is a symbol and the second token is (or starts with) =, the controlled form of *repetitor* is expected.

Controlled repetitive loops

The controlled form specifies *name*, a **control variable** that is assigned an initial value (the result of *expri*, formatted as though 0 had been added) before the first execution of the instruction list. The variable is then stepped (by adding the result of *exprb*) before the second and subsequent times that the instruction list is processed.

The instruction list is processed repeatedly while the end condition (determined by the result of *exprt*) is not met. If *exprb* is positive or 0, the loop is ended when *name* is greater than *exprt*. If negative, the loop is ended when *name* is less than *exprt*.

The *expri*, *exprt*, and *exprb* options must result in numbers. They are evaluated only one time, before the loop begins and before the control variable is set to its initial value. The default value for *exprb* is 1. If *exprt* is omitted, the loop runs indefinitely unless some other condition stops it.

Example:

```
Do I=3 to -2 by -1        /* Displays:   */
  say i                   /*      3      */
  end                     /*      2      */
                          /*      1      */
                          /*      0      */
                          /*      -1     */
                          /*      -2     */
```

The numbers do not have to be whole numbers:

Example:

```
I=0.3                     /* Displays:   */
Do Y=I to I+4 by 0.7      /*     0.3     */
  say Y                   /*     1.0     */
  end                     /*     1.7     */
                          /*     2.4     */
                          /*     3.1     */
                          /*     3.8     */
```

The control variable can be altered within the loop, and this may affect the iteration of the loop. Altering the value of the control variable is not usually considered good programming practice, though it may be appropriate in certain circumstances.

Note that the end condition is tested at the start of each iteration (and after the control variable is stepped, on the second and subsequent iterations). Therefore, if the end condition is met immediately, the group of instructions can be skipped entirely. Note also that the control variable is referred to by name. If (for example) the compound name A.I is used for the control variable, altering I within the loop causes a change in the control variable.

The execution of a controlled loop can be bounded further by a FOR phrase. In this case, you must specify *exprf*, and it must evaluate to a positive whole number or zero. This acts just like the repetition count in a simple repetitive loop, and sets a limit to the number of iterations around the loop if no other condition stops it. Like the TO and BY expressions, it is evaluated only one time—when the DO instruction is first processed and before the control variable receives its initial value. Like the TO condition, the FOR condition is checked at the start of each iteration.

Example:

```
Do Y=0.3 to 4.3 by 0.7 for 3  /* Displays:    */
  say Y                       /*     0.3      */
  end                         /*     1.0      */
                              /*     1.7      */
```

In a controlled loop, the *name* describing the control variable can be specified on the END clause. This *name* must match *name* in the DO clause in all respects except case (note that no substitution for compound variables is carried out); a syntax error results if it does not. This enables the nesting of loops to be checked automatically, with minimal overhead.

Example:

```
Do K=1 to 10
  ...
  ...
  End k  /* Checks that this is the END for K loop */
```

The NUMERIC settings may affect the successive values of the control variable, because REXX arithmetic rules apply to the computation of stepping the control variable.

Conditional phrases (WHILE and UNTIL)

A conditional phrase can modify the iteration of a repetitive DO loop. It may cause the termination of a loop. It can follow any of the forms of *repetitor* (none, FOREVER, simple, or controlled). If you specify WHILE or UNTIL, *exprw* or *expru*, respectively, is evaluated each time around the loop using the latest values of all variables (and must evaluate to either `0` or `1`), and the loop is ended if *exprw* evaluates to `0` or *expru* evaluates to `1`.

For a WHILE loop, the condition is evaluated at the top of the group of instructions. For an UNTIL loop, the condition is evaluated at the bottom—before the control variable has been stepped.

Example:

```
Do I=1 to 10 by 2 until i>6
  say i
  end
/* Displays: "1" "3" "5" "7" */
```

Tip: You can also modify the execution of repetitive loops by using the LEAVE or ITERATE instructions.

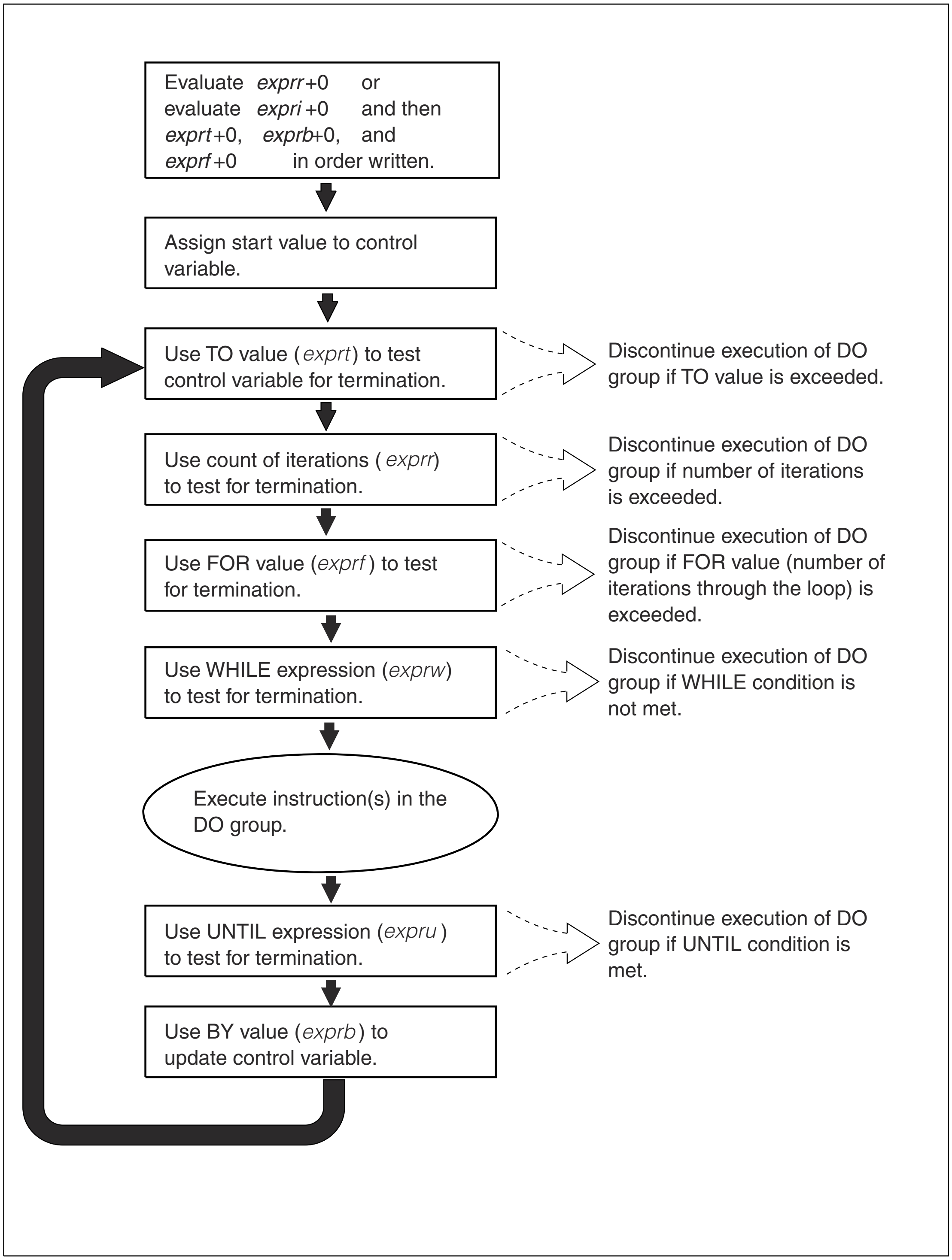
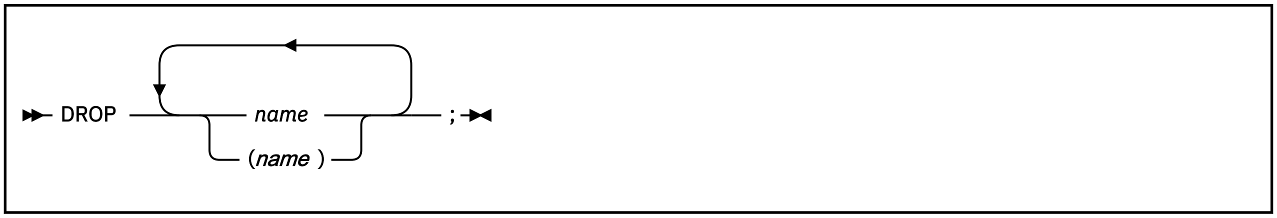

Evaluate *exprr*+0 or
evaluate *expri* +0 and then
exprt+0, *exprb*+0, and
exprf +0 in order written.
Assign start value to control variable.
Use TO value (*exprt*) to test control variable for termination.
Discontinue execution of DO group if TO value is exceeded.
Use count of iterations (*exprr*) to test for termination.
Discontinue execution of DO group if number of iterations is exceeded.
Use FOR value (*exprf*) to test for termination.
Discontinue execution of DO group if FOR value (number of iterations through the loop) is exceeded.
Use WHILE expression (*exprw*) to test for termination.
Discontinue execution of DO group if WHILE condition is not met.
Execute instruction(s) in the DO group.
Use UNTIL expression (*expru*) to test for termination.
Discontinue execution of DO group if UNTIL condition is met.
Use BY value (*exprb*) to update control variable.

Figure 7. Concept of a DO loop

DROP



DROP "unassigns" variables, that is, restores them to their original uninitialized state. If *name* is not enclosed in parentheses, it identifies a variable you want to drop and must be a symbol that is a valid variable name, separated from any other *name* by one or more blanks or comments.

If parentheses enclose a single *name*, then its value is used as a subsidiary list of variables to drop. (Blanks are not necessary either inside or outside the parentheses, but you can add them if desired.) This subsidiary list must follow the same rules as the original list (that is, be valid variable names, separated by blanks) except that no parentheses are allowed.

Variables are dropped in sequence from left to right. It is not an error to specify a name more than one time or to DROP a variable that is not known. If an exposed variable is named, (see "PROCEDURE" on page 60), the variable in the older generation is dropped.

Example:

```
j=4
Drop  a z.3 z.j
/* Drops the variables: A, Z.3, and Z.4              */
/* so that reference to them returns their names.    */
```

Here, a variable name in parentheses is used as a subsidiary list.

Example:

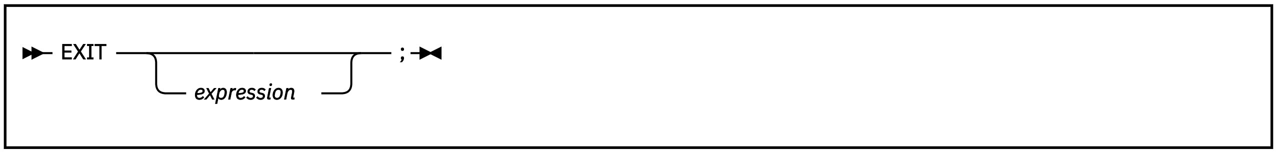
```
mylist='c d e'
drop (mylist) f
/* Drops the variables C, D, E, and F          */
/* Does not drop MYLIST                        */
```

Specifying a stem (that is, a symbol that contains only one period, as the last character), drops all variables starting with that stem.

Example:

```
Drop  z.
/* Drops all variables with names starting with Z. */
```

EXIT



EXIT leaves a program unconditionally. Optionally EXIT returns a character string to the caller. The program is stopped immediately, even if an internal routine is currently being run. If no internal routine is active, RETURN (see "RETURN" on page 64) and EXIT are identical in their effect on the program that is being run.

If you specify *expression*, it is evaluated and the string resulting from the evaluation is passed back to the caller when the program stops.

Example:

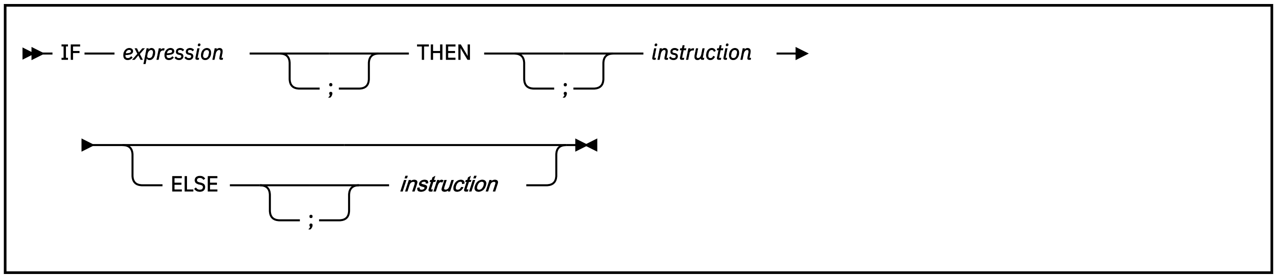
```
j=3
Exit j*4
/* Would exit with the string '12' */
```

If you do not specify *expression*, no data is passed back to the caller. If the program was called as an external function, this is detected as an error—either immediately (if RETURN was used), or on return to the caller (if EXIT was used).

"Running off the end" of the program is always equivalent to the instruction EXIT, in that it stops the whole program and returns no result string.

Note: If the program was called through a command interface, an attempt is made to convert the returned value to a return code acceptable by the host. If the conversion fails, it is deemed to be a failure of the host interface and thus is not subject to trapping with SIGNAL ON SYNTAX. The returned string must be a whole number whose value fits in a general register (that is, must be in the range -2**31 through 2**31-1).

IF



IF conditionally processes an instruction or group of instructions depending on the evaluation of the *expression*. The *expression* is evaluated and must result in 0 or 1.

The instruction after the THEN is processed only if the result is 1 (true). If you specify an ELSE, the instruction after the ELSE is processed only if the result of the evaluation is 0 (false).

Example:

```
if answer='YES' then say 'OK!'
                else say 'Why not?'
```

Remember that if the ELSE clause is on the same line as the last clause of the THEN part, you need a semicolon before the ELSE.

Example:

```
if answer='YES' then say 'OK!';  else say 'Why not?'
```

The ELSE binds to the nearest IF at the same level. You can use the NOP instruction to eliminate errors and possible confusion when IF constructs are nested, as in the following example.

Example:

```
If answer = 'YES' Then
   If name = 'FRED' Then
      say 'OK, Fred.'
   Else
      nop
Else
   say 'Why not?'
```

Note:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, SELECT, or the IF instruction itself. A null clause is not an instruction, so putting an extra semicolon (or label) after the THEN or ELSE is not equivalent to putting a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.
2. The symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the IF clause to be ended by the THEN, without a `;` being required. If this were not so, people who are accustomed to other computer languages would experience considerable difficulties.

INTERPRET

```
▶▶─ INTERPRET ── expression ;─▶◀
```

INTERPRET processes instructions that have been built dynamically by evaluating *expression*.

The *expression* is evaluated and is then processed (interpreted) just as though the resulting string were a line inserted into the program (and bracketed by a DO; and an END;).

Any instructions (including INTERPRET instructions) are allowed, but note that constructions such as DO...END and SELECT...END must be complete. For example, a string of instructions being interpreted cannot contain a LEAVE or ITERATE instruction (valid only within a repetitive DO loop) unless it also contains the whole repetitive DO...END construct.

A semicolon is implied at the end of the expression during execution, if one was not supplied.

Example:

```
data='FRED'
interpret data '= 4'
/* Builds the string  "FRED = 4" and        */
/* Processes:  FRED = 4;                    */
/* Thus the variable FRED is set to "4"     */
```

Example:

```
data='do 3; say "Hello there!"; end'
interpret data        /* Displays:          */
                      /*  Hello there!      */
                      /*  Hello there!      */
                      /*  Hello there!      */
```

Note:

1. Label clauses are not permitted in an interpreted character string.
2. If you are new to the concept of the INTERPRET instruction and are getting results that you do not understand, you may find that executing it with `TRACE  R` or `TRACE  I` in effect is helpful.

   Example:

   ```
   /* Here is a small REXX program. */
   Trace Int
   name='Kitty'
   indirect='name'
   interpret 'say "Hello"' indirect'"!"'
   ```

   When this is run, it gives the trace:

   ```
   kitty
        3 *-* name='Kitty'
          >L>   "Kitty"
        4 *-* indirect='name'
          >L>   "name"
        5 *-* interpret 'say "Hello"' indirect'"!"'
   ```

```
       >L>   "say "Hello""
       >V>   "name"
       >O>   "say "Hello" name"
       >L>   ""!""
       >O>   "say "Hello" name"!""
       *-*  say "Hello" name"!"
       >L>    "Hello"
       >V>    "Kitty"
       >O>    "Hello Kitty"
       >L>    "!"
       >O>    "Hello Kitty!"
Hello Kitty!
```

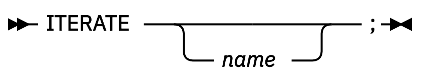
Here, lines 3 and 4 set the variables used in line 5. Execution of line 5 then proceeds in two stages. First the string to be interpreted is built up, using a literal string, a variable (`INDIRECT`), and another literal string. The resulting pure character string is then interpreted, just as though it were actually part of the original program. Because it is a new clause, it is traced as such (the second `*-*` trace flag under line 5) and is then processed. Again a literal string is concatenated to the value of a variable (`NAME`) and another literal, and the final result (`Hello Kitty!`) is then displayed.

3. For many purposes, you can use the VALUE function (see "VALUE" on page 104) instead of the INTERPRET instruction. The following line could, therefore, have replaced line 5 in the last example:

```
say "Hello" value(indirect)"!"
```

INTERPRET is usually required only in special cases, such as when two or more statements are to be interpreted together, or when an expression is to be evaluated dynamically.

ITERATE



ITERATE alters the flow within a repetitive DO loop (that is, any DO construct other than that with a simple DO).

Execution of the group of instructions stops, and control is passed to the DO instruction. The control variable (if any) is incremented and tested, as usual, and the group of instructions is processed again, unless the DO instruction ends the loop.

The *name* is a symbol, taken as a constant. If *name* is not specified, ITERATE steps the innermost active repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop (which may be the innermost), and this is the loop that is stepped. Any active loops inside the one selected for iteration are ended (as though by a LEAVE instruction).

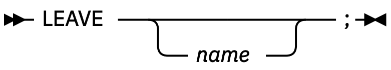
Example:

```
do i=1 to 4
  if i=2 then iterate
  say i
  end
/* Displays the numbers:  "1" "3" "4" */
```

Note:

1. If specified, *name* must match the symbol naming the control variable in the DO clause in all respects except case. No substitution for compound variables is carried out when the comparison is made.
2. A loop is active if it is currently being processed. If a subroutine is called (or an INTERPRET instruction is processed) during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. ITERATE cannot be used to step an inactive loop.
3. If more than one active loop uses the same control variable, ITERATE selects the innermost loop.

LEAVE



LEAVE causes an immediate exit from one or more repetitive DO loops (that is, any DO construct other than a simple DO).

Processing of the group of instructions is ended, and control is passed to the instruction following the END clause. The control variable (if any) will contain the value it had when the LEAVE instruction was processed.

The *name* is a symbol, taken as a constant. If *name* is not specified, LEAVE ends the innermost active repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop (which may be the innermost), and that loop (and any active loops inside it) is then ended. Control then passes to the clause following the END that matches the DO clause of the selected loop.

Example:

```
do i=1 to 5
  say i
  if i=3 then leave
  end
/* Displays the numbers:  "1" "2" "3" */
```

Note:

1. If specified, *name* must match the symbol naming the control variable in the DO clause in all respects except case. No substitution for compound variables is carried out when the comparison is made.
2. A loop is active if it is currently being processed. If a subroutine is called (or an INTERPRET instruction is processed) during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. LEAVE cannot be used to end an inactive loop.
3. If more than one active loop uses the same control variable, LEAVE selects the innermost loop.

NOP

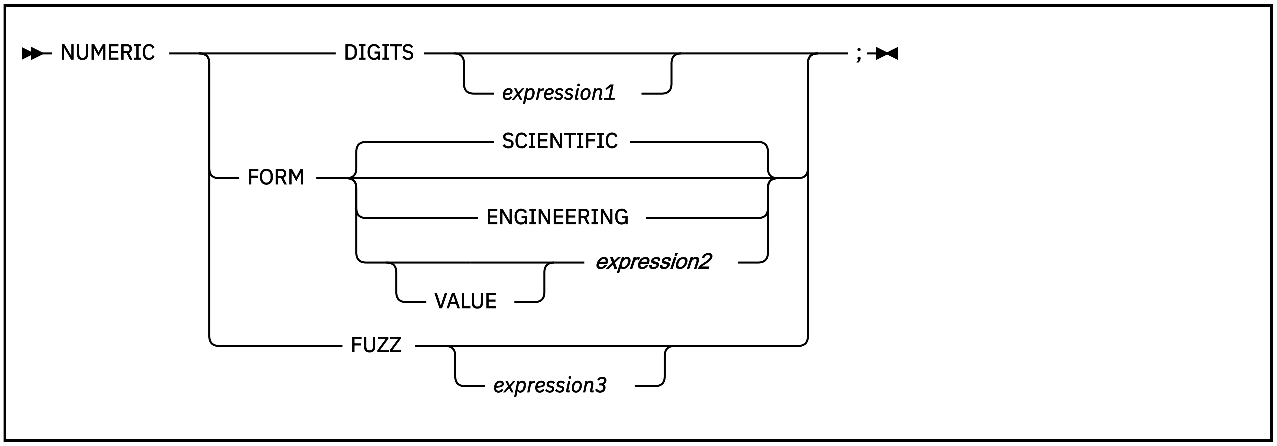
NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause:

Example:

```
Select
   when a=c then nop           /* Do nothing */
   when a>c then say 'A > C'
   otherwise     say 'A < C'
end
```

Tip: Putting an extra semicolon instead of the NOP would merely insert a null clause, which would be ignored. The second WHEN clause would be seen as the first instruction expected after the THEN, and would, therefore, be treated as a syntax error. NOP is a true instruction, however, and is, therefore, a valid target for the THEN clause.

NUMERIC



NUMERIC changes the way in which a program carries out arithmetic operations. See Chapter 6, "Numbers and arithmetic," on page 171-"Errors" on page 178 for the options of this instruction, but in summary:

NUMERIC DIGITS

controls the precision to which arithmetic operations and arithmetic built-in functions are evaluated. If you omit *expression1*, the precision defaults to 9 digits. Otherwise, *expression1* must evaluate to a positive whole number and must be larger than the current NUMERIC FUZZ setting.

There is no limit to the value for DIGITS (except the amount of storage available), but note that high precisions are likely to require a good deal of processing time. It is suggested that you use the default value wherever possible.

You can retrieve the current NUMERIC DIGITS setting with the DIGITS built-in function. See "DIGITS" on page 88.

NUMERIC FORM

controls which form of exponential notation REXX uses for the result of arithmetic operations and arithmetic built-in functions. This may be either SCIENTIFIC (in which case only one, nonzero digit appears before the decimal point) or ENGINEERING (in which case the power of 10 is always a multiple of 3). The default is SCIENTIFIC. The subkeywords SCIENTIFIC or ENGINEERING set the FORM directly, or it is taken from the result of evaluating the expression (*expression2*) that follows VALUE. The result in this case must be either `SCIENTIFIC` or `ENGINEERING`. You can omit the subkeyword VALUE if *expression2* does not begin with a symbol or a literal string (that is, if it starts with a special character, such as an operator character or parenthesis).

You can retrieve the current NUMERIC FORM setting with the FORM built-in function. See "FORM" on page 90.

NUMERIC FUZZ

controls how many digits, at full precision, are ignored during a numeric comparison operation. (See "Numeric comparisons" on page 176.) If you omit *expression3*, the default is `0` digits. Otherwise, *expression3* must evaluate to `0` or a positive whole number, rounded if necessary according to the current NUMERIC DIGITS setting, and must be smaller than the current NUMERIC DIGITS setting.

NUMERIC FUZZ temporarily reduces the value of NUMERIC DIGITS by the NUMERIC FUZZ value during every numeric comparison. The numbers are subtracted under a precision of DIGITS minus FUZZ digits during the comparison and are then compared with 0.

You can retrieve the current NUMERIC FUZZ setting with the FUZZ built-in function. See "FUZZ" on page 91.

The three numeric settings are automatically saved across internal and external subroutine and function calls. See "CALL" on page 43 for more details about the CALL instruction.

OPTIONS

```
▶▶─ OPTIONS ── expression ;─▶◀
```

OPTIONS passes special requests or parameters to the language processor. For example, these may be language processor options or perhaps define a special character set.

The *expression* is evaluated, and the result is examined one word at a time. The language processor converts the words to uppercase. If the language processor recognizes the words, then they are obeyed. Words that are not recognized are ignored and assumed to be instructions to a different processor.

The language processor recognizes the following words:

ETMODE
: specifies that literal strings and symbols and comments containing DBCS characters are checked for being valid DBCS strings. If you use this option, it must be the first instruction of the program.

If the *expression* is an external function call, for example `OPTIONS 'GETETMOD'()`, and the program contains DBCS literal strings, enclose the name of the function in quotation marks to ensure that the entire program is not scanned before the option takes effect. It is not suggested to use internal function calls to set ETMODE because of the possibility of errors in interpreting DBCS literal strings in the program.

NOETMODE
: specifies that literal strings and symbols and comments containing DBCS characters are not checked for being valid DBCS strings. NOETMODE is the default. The language processor ignores this option unless it is the first instruction in a program.

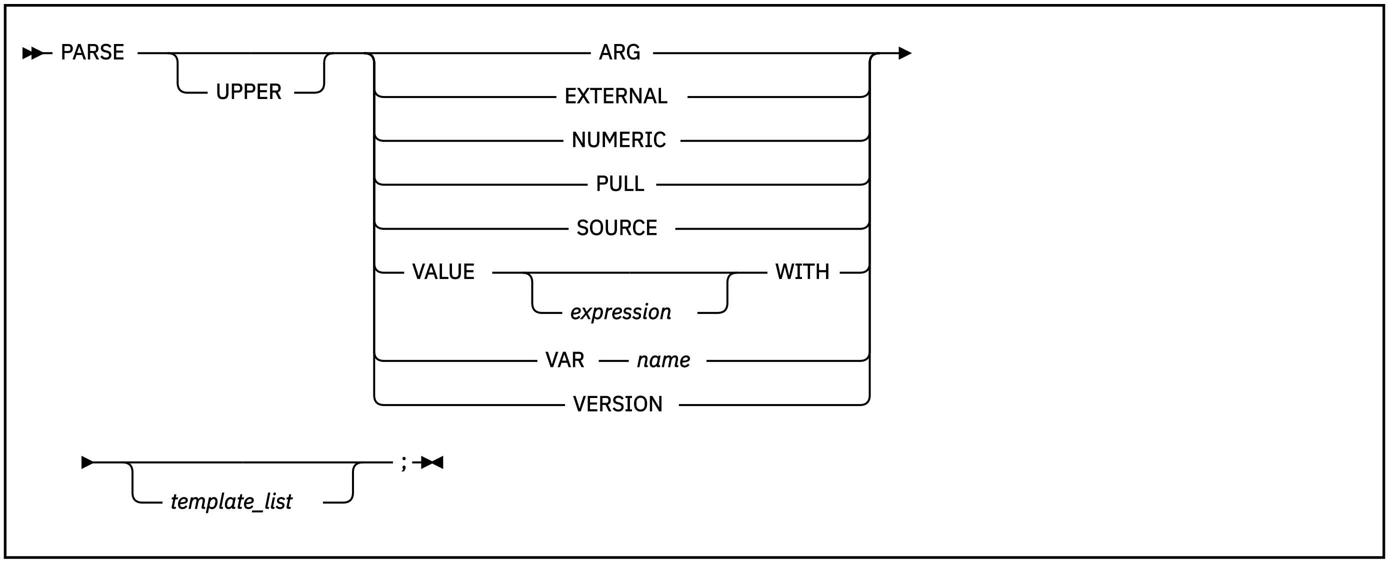
EXMODE
: specifies that instructions, operators, and functions handle DBCS data in mixed strings on a logical character basis. DBCS data integrity is maintained.

NOEXMODE
: specifies that any data in strings is handled on a byte basis. The integrity of DBCS characters, if any, may be lost. NOEXMODE is the default.

Note:

1. Because of the language processor's scanning procedures, you must place an `OPTIONS 'ETMODE'` instruction as the first instruction in a program containing DBCS characters in literal strings, symbols, or comments. If you do not place `OPTIONS 'ETMODE'` as the first instruction and you use it later in the program, you receive error message IRX0033I. If you do place it as the first instruction of your program, all subsequent uses are ignored. If the expression contains anything that would start a label search, all clauses tokenized during the label search process are tokenized within the current setting of ETMODE. Therefore, if this is the first statement in the program, the default is NOETMODE.
2. To ensure proper scanning of a program containing DBCS literals and DBCS comments, enter the words ETMODE, NOETMODE, EXMODE, and NOEXMODE as literal strings (that is, enclosed in quotation marks) in the OPTIONS instruction.
3. The EXMODE setting is saved and restored across subroutine and function calls.
4. To distinguish DBCS characters from 1-byte EBCDIC characters, sequences of DBCS characters are enclosed with a shift-out (SO) character and a shift-in (SI) character. The hexadecimal values of the SO and SI characters are X'0E' and X'0F', respectively.
5. When you specify `OPTIONS 'ETMODE'`, DBCS characters within a literal string are excluded from the search for a closing quotation mark in literal strings.
6. The words ETMODE, NOETMODE, EXMODE, and NOEXMODE can appear several times within the result. The one that takes effect is determined by the last valid one specified between the pairs ETMODE-NOETMODE and EXMODE-NOEXMODE.

PARSE



PARSE assigns data (from various sources) to one or more variables according to the rules of parsing (see Chapter 5, "Parsing," on page 157).

The *template_list* is often a single template but may be several templates separated by commas. If specified, each template is a list of symbols separated by blanks or patterns or both.

Each template is applied to a single source string. Specifying multiple templates is never a syntax error, but only the PARSE ARG variant can supply more than one non-null source string. See "Parsing multiple strings" on page 166 for information about parsing multiple source strings.

If you do not specify a template, no variables are set but action is taken to prepare the data for parsing, if necessary. Thus for PARSE PULL, a data string is removed from the queue, and for PARSE VALUE, *expression* is evaluated. For PARSE VAR, the specified variable is accessed. If it does not have a value, the NOVALUE condition is raised, if it is enabled.

If you specify the UPPER option, the data to be parsed is first translated to uppercase (that is, lowercase a–z to uppercase A–Z). Otherwise, no uppercase translation takes place during the parsing.

The following list describes the data for each variant of the PARSE instruction.

PARSE ARG

parses the string or strings passed to a program or internal routine as input arguments. See "ARG" on page 42 for details and examples of the ARG instruction.

Tip: You can also retrieve or check the argument strings to a REXX program or internal routine with the ARG built-in function (see "ARG (Argument)" on page 79).

PARSE EXTERNAL

In TSO/E, PARSE EXTERNAL reads from the:

- Terminal (TSO/E foreground)
- Input stream, which is SYSTSIN (TSO/E background).

In non-TSO/E address spaces, PARSE EXTERNAL reads from the input stream as defined by the file name in the INDD field in the module name table (see "Module name table" on page 326). The system default is SYSTSIN. PARSE EXTERNAL returns a field based on the record that is read from the INDD file. If SYSTSIN has no data, the PARSE EXTERNAL instruction returns a null string.

PARSE NUMERIC

The current numeric controls (as set by the NUMERIC instruction) are available (see "NUMERIC" on page 55). These controls are in the order DIGITS FUZZ FORM.

Example:

```
Parse Numeric Var1
```

After this instruction, `Var1` would be equal to: 9 0 SCIENTIFIC. See "NUMERIC" on page 55 and the built-in functions "DIGITS" on page 88, "FORM" on page 90, and "FUZZ" on page 91.

PARSE PULL

parses the next string from the external data queue. If the external data queue is empty, PARSE PULL reads a line from the default input stream (the user's terminal), and the program pauses, if necessary, until a line is complete. You can add data to the head or tail of the queue by using the PUSH and QUEUE instructions, respectively. You can find the number of lines currently in the queue with the QUEUED built-in function. (See "QUEUED" on page 96.) Other programs in the system can alter the queue and use it as a means of communication with programs written in REXX. See also "PULL" on page 62 for the PULL instruction.

PULL and PARSE PULL read from the data stack. In TSO/E, if the data stack is empty, PULL and PARSE PULL read from the:

- Terminal (TSO/E foreground)
- Input stream, which is SYSTSIN (TSO/E background).

In non-TSO/E address spaces, if the data stack is empty, PULL and PARSE PULL read from the input stream as defined by the file name in the INDD field in the module name table (see "Module name table" on page 326). The system default is SYSTSIN. If SYSTSIN has no data, the PULL and PARSE PULL instructions return a null string.

PARSE SOURCE

parses data describing the source of the program running. The language processor returns a string that is fixed (does not change) while the program is running.

The source string contains the following tokens:

1. The characters TSO
2. The string COMMAND, FUNCTION, or SUBROUTINE depending on whether the program was called as some kind of host command (for example, as an exec from TSO/E READY mode), or from a function call in an expression, or using the CALL instruction.
3. Usually, name of the exec in uppercase. If the name is not known, this token is a question mark (?). If the name is an extended execname, this field is the full name, possibly greater than 8-characters and not converted to uppercase. If the execname contains any blank characters (which is possible in an execname of file system), the blank characters are replaced by null characters (X'00') within this token. See "Format of the exec block" on page 397 and "Format of the in-storage control block" on page 400 for more information about how the exec load routine can load an extended execname.
4. Name of the DD from which the exec was loaded. If the name is not known, this token is a question mark (?).
5. Name of the data set from which the exec was loaded. If the name is not known, this token is a question mark (?). If the name contains any blank characters (which is possible in a z/OS UNIX directory), the blank characters are replaced by null characters (X'00') within this token.
6. Name of the exec as it was called, that is, the name is not folded to uppercase. If the name is not known, this token is a question mark (?).

   If the name returned in token3 is an extended name, this token is a question mark (?).
7. Initial (default) host command environment in uppercase. For example, this token may be TSO or MVS.
8. Name of the address space in uppercase. For example, the value may be MVS (non-TSO/E) or TSO/E or ISPF. If the exec was called from ISPF, the address space name is ISPF.

   The value is taken from the parameter block (see "Characteristics of a Language Processor Environment" on page 317). Note that the initialization exit routines may change the name specified in the parameters module. If the name of the address space is not known, this token is a question mark (?).

9. Eight character user token. This is the token that is specified in the PARSETOK field in the parameters module (see "Characteristics of a Language Processor Environment" on page 317).

For example, the string parsed might look like one of the following:

```
TSO COMMAND PROGA SYSXR07 EGGERS.ECE.EXEC ? TSO TSO/E ?

TSO SUBROUTINE PROGSUB SYSEXEC ? ? TSO ISPF ?

TSO SUBROUTINE /u/cmddir/pgm.cmd PATH /u/cmddir/pgm.cmd
? SH OMVS OpenMVS
```

PARSE VALUE

parses the data that is the result of evaluating *expression*. If you specify no *expression*, then the null string is used. Note that WITH is a subkeyword in this context and cannot be used as a symbol within *expression*.

Thus, for example:

```
PARSE VALUE time() WITH  hours ':' mins ':' secs
```

gets the current time and splits it into its constituent parts.

PARSE VAR *name*

parses the value of the variable *name*. The *name* must be a symbol that is valid as a variable name (that is, it cannot start with a period or a digit). Note that the variable *name* is not changed unless it appears in the template, so that for example:

```
PARSE VAR string word1 string
```

removes the first word from *string*, puts it in the variable *word1*, and assigns the remainder back to *string*. Similarly

```
PARSE UPPER VAR string word1 string
```

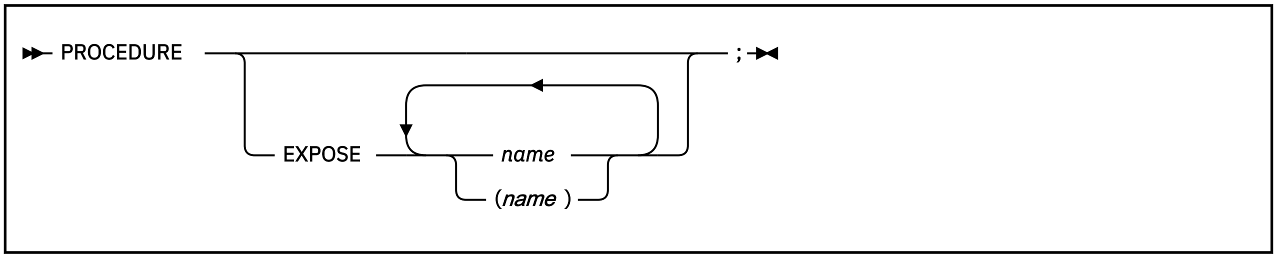
in addition translates the data from *string* to uppercase before it is parsed.

PARSE VERSION

parses information describing the language level and the date of the language processor. This information consists of five blank-delimited words:

1. A word describing the language, which is the string "REXX370"
2. The language level description, for example, "3.46"
3. Three tokens describing the language processor release date, for example, "31 May 2001".

PROCEDURE



PROCEDURE, within an internal routine (subroutine or function), protects variables by making them unknown to the instructions that follow it. After a RETURN instruction is processed, the original variables environment is restored and any variables used in the routine (that were not exposed) are dropped. (An exposed variable is one belonging to a caller of a routine that the PROCEDURE instruction has exposed.

When the routine refers to or alters the variable, the original (caller's) copy of the variable is used.) An internal routine need not include a PROCEDURE instruction; in this case the variables it is manipulating are those the caller "owns." If used, the PROCEDURE instruction must be the first instruction processed after the CALL or function invocation; that is, it must be the first instruction following the label.

If you use the EXPOSE option, any variable specified by *name* is exposed. Any reference to it (including setting and dropping) refers to the variables environment the caller owns. Hence, the values of existing variables are accessible, and any changes are persistent even on RETURN from the routine. If *name* is not enclosed in parentheses, it identifies a variable you want to expose and must be a symbol that is a valid variable name, separated from any other *name* with one or more blanks.

If parentheses enclose a single *name*, then, after the variable *name* is exposed, the value of *name* is immediately used as a subsidiary list of variables. (Blanks are not necessary either inside or outside the parentheses, but you can add them if desired.) This subsidiary list must follow the same rules as the original list (that is, valid variable names, separated by blanks) except that no parentheses are allowed.

Variables are exposed in sequence from left to right. It is not an error to specify a name more than one time, or to specify a name that the caller has not used as a variable.

Any variables in the main program that are not exposed are still protected. Therefore, some limited set of the caller's variables can be made accessible, and these variables can be changed (or new variables in this set can be created). All these changes are visible to the caller upon RETURN from the routine.

Example:

```
/* This is the main REXX program */
j=1; z.1='a'
call toft
say j k m       /* Displays "1 7 M"          */
exit

/* This is a subroutine     */
toft: procedure expose j k z.j
   say j k z.j  /* Displays "1 K a"          */
   k=7; m=3     /* Note: M is not exposed    */
   return
```

Note that if `Z.J` in the EXPOSE list had been placed before `J`, the caller's value of `J` would not have been visible at that time, so `Z.1` would not have been exposed.

The variables in a subsidiary list are also exposed from left to right.

Example:

```
/* This is the main REXX program */
j=1;k=6;m=9
a ='j k m'
call test
exit

/* This is a subroutine     */
test: procedure expose (a)   /* Exposes A, J, K, and M     */
  say a j k m                /* Displays "j k m 1 6 9"     */
  return
```

You can use subsidiary lists to more easily expose a number of variables at one time or, with the VALUE built-in function, to manipulate dynamically named variables.

Example:

```
/* This is the main REXX program */
c=11; d=12; e=13
Showlist='c d'     /* but not E              */
call Playvars
say c d e f        /* Displays "11 New 13 9" */
exit

/* This is a subroutine     */
Playvars: procedure expose (showlist) f
 say word(showlist,2)              /* Displays "d"                     */
 say value(word(showlist,2),'New') /* Displays "12" and sets new value */
```

```
  say value(word(showlist,2))        /* Displays "New"                   */
  e=8                                /* E is not exposed                 */
  f=9                                /* F was explicitly exposed         */
  return
```

Specifying a **stem** as *name* exposes this stem and *all possible* compound variables whose names begin with that stem. (See page "Stems" on page 20 for information about stems.)

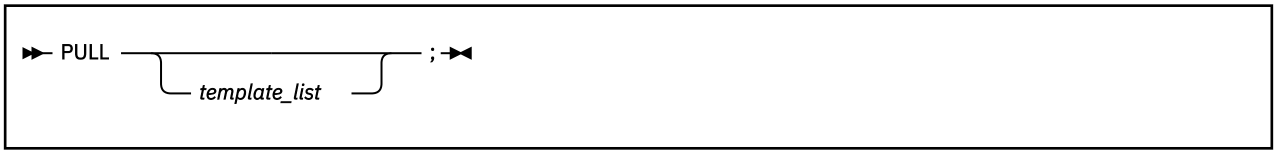
Example:

```
/* This is the main REXX program */
a.=11; i=13; j=15
i = i + 1
C.5 = 'FRED'
call lucky7
say a. a.1 i j c. c.5
say 'You should see 11 7 14 15 C. FRED'
exit
lucky7:Procedure Expose i j a. c.
/* This exposes I, J, and all variables whose     */
/* names start with A. or C.                      */
A.1='7'  /* This sets A.1 in the caller's         */
         /* environment, even if it did not       */
         /* previously exist.                     */
return
```

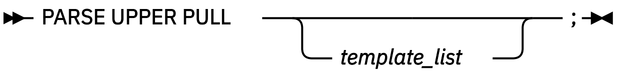
Variables may be exposed through several generations of routines, if desired, by ensuring that they are included on all intermediate PROCEDURE instructions.

See page "CALL" on page 43 and page Chapter 4, "Functions," on page 73 for details and examples of how routines are called.

PULL



PULL reads a string from the head of the external data queue. It is just a short form of the instruction:



The current head-of-queue is read as one string. Without a *template_list* specified, no further action is taken (and the string is thus effectively discarded). If specified, a *template_list* is usually a single template, which is a list of symbols separated by blanks or patterns or both. (The *template_list* can be several templates separated by commas, but PULL parses only one source string; if you specify several comma-separated templates, variables in templates other than the first one are assigned the null string.) The string is translated to uppercase (that is, lowercase a–z to uppercase A–Z) and then parsed into variables according to the rules described in the section on parsing (see Chapter 5, "Parsing," on page 157). Use the PARSE PULL instruction if you do not desire uppercase translation.

The TSO/E implementation of the external data queue is the data stack. REXX execs that run in TSO/E and non-TSO/E address spaces can use the data stack. In TSO/E, if the data stack is empty, PULL reads from the:

- Terminal (TSO/E foreground)
- Input stream, which is SYSTSIN (TSO/E background).

In non-TSO/E address spaces, if the data stack is empty, PULL reads from the input stream as defined by the file name in the INDD field in the module name table (see "Module name table" on page 326). The system default is SYSTSIN. If SYSTSIN has no data, the PULL instruction returns a null string.

The length of each element you can place onto the data stack can be up to one byte less than 16 megabytes.

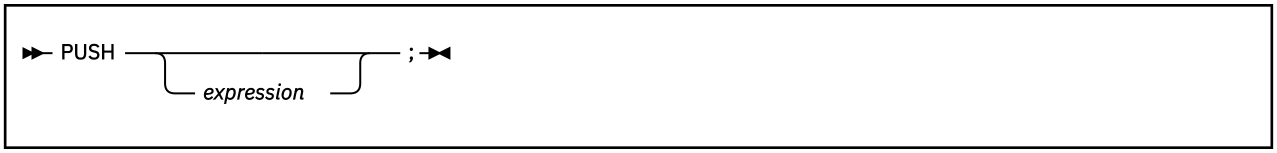
Example:

```
Say 'Do you want to erase the file?  Answer Yes or No:'
Pull answer .
if answer='NO' then say 'The file will not be erased.'
```

Here the dummy placeholder, a period (.), is used on the template to isolate the first word the user enters.

The QUEUED built-in function (see “QUEUED” on page 96) returns the number of lines currently in the external data queue.

PUSH



PUSH stacks the string resulting from the evaluation of *expression* LIFO (Last In, First Out) onto the external data queue.

If you do not specify *expression*, a null string is stacked.

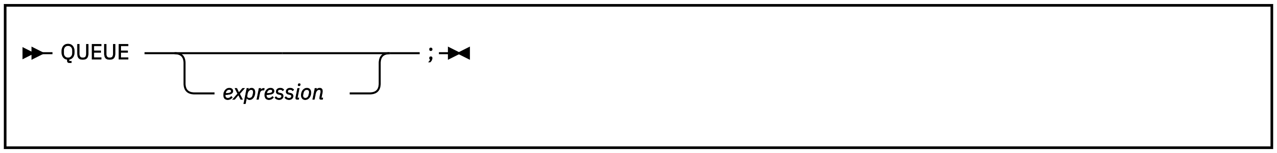
Note: The TSO/E implementation of the external data queue is the data stack. The length of an element in the data stack can be up to one byte less than 16 megabytes. The data stack contains one buffer initially, but you can create additional buffers using the TSO/E REXX command MAKEBUF.

Example:

```
a='Fred'
push        /* Puts a null line onto the queue */
push a 2    /* Puts "Fred 2"    onto the queue */
```

See “QUEUED” on page 96 for the QUEUED built-in function that returns the number of lines currently in the external data queue.

QUEUE



QUEUE appends the string resulting from *expression* to the tail of the external data queue. That is, it is added FIFO (First In, First Out).

If you do not specify *expression*, a null string is queued.

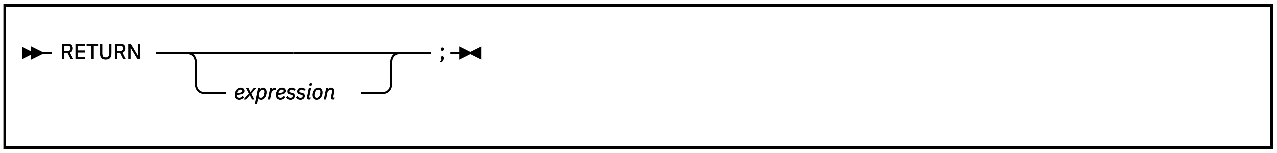
Note: The TSO/E implementation of the external data queue is the data stack. The length of an element in the data stack can be up to one byte less than 16 megabytes. The data stack contains one buffer initially, but you can create additional buffers using the TSO/E REXX command MAKEBUF.

Example:

```
a='Toft'
queue a 2   /* Enqueues "Toft 2" */
queue       /* Enqueues a null line behind the last */
```

See “QUEUED” on page 96 for the QUEUED built-in function that returns the number of lines currently in the external data queue.

RETURN



RETURN returns control (and possibly a result) from a REXX program or internal routine to the point of its invocation.

If no internal routine (subroutine or function) is active, RETURN and EXIT are identical in their effect on the program that is being run, (See “EXIT” on page 51.) in that both would cause an immediate exit (or return) from the program.

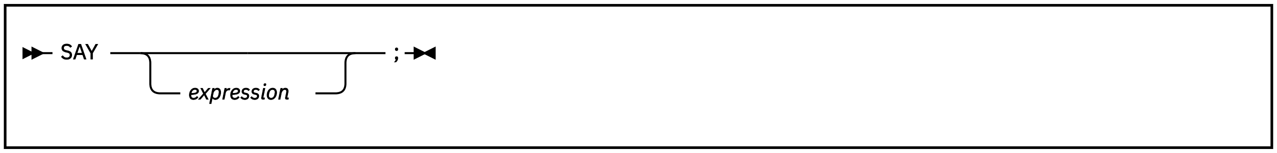
If a *subroutine* is being run (see the CALL instruction), *expression* (if any) is evaluated, control passes back to the caller, and the REXX special variable RESULT is set to the value of *expression*. If *expression* is omitted, the special variable RESULT is dropped (becomes uninitialized). The various settings saved at the time of the CALL (tracing, addresses, and so forth) are also restored. (See “CALL” on page 43.)

If a *function* is being processed, the action taken is identical to that for *subroutine* processing, except that *expression* **must** be specified on the RETURN instruction that returns from a function. The result of *expression* is then used in the original expression at the point where the function was called. See Chapter 4, “Functions,” on page 73 for more details. Failure to set an *expression* when returning from a function call would raise an error condition in the invoking function call.

If a PROCEDURE instruction was processed within the routine (subroutine or internal function), all variables of the current generation are dropped (and those of the previous generation are exposed) after *expression* is evaluated and before the result is used or assigned to RESULT.

Note: The variables within a PROCEDURE form a new generation of variables that are known locally within the PROCEDURE and unknown to the caller of that PROCEDURE. Variables defined outside of the PROCEDURE constitute a prior generation that are hidden from the PROCEDURE. However variables that are exposed on the PROCEDURE statement are known to both the PROCEDURE and the caller, and can be used as global variables shared between the PROCEDURE and its caller.

SAY



SAY writes a line to the output stream. This typically displays it to the user, but the output destination can depend on the implementation. The result of *expression* may be of any length. If you omit *expression*, the null string is written.

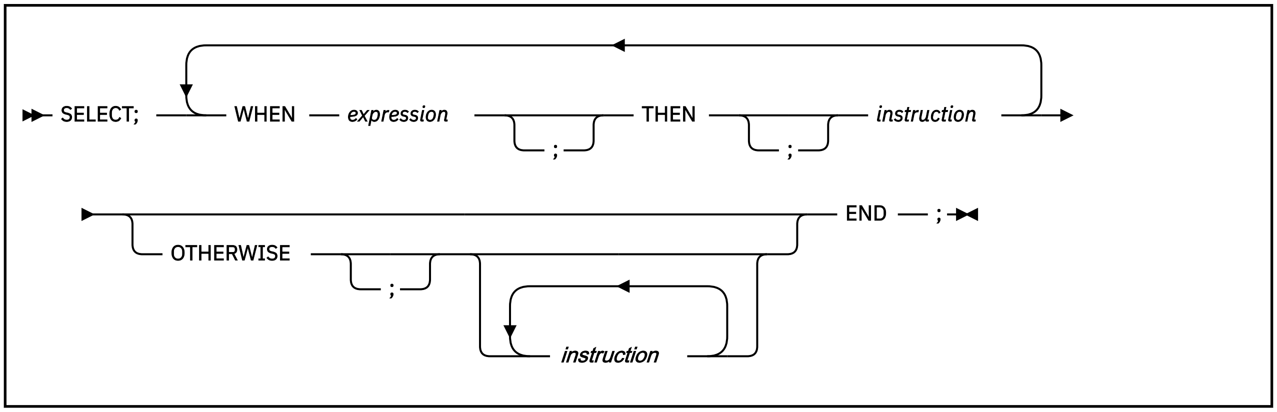
If a REXX exec runs in TSO/E foreground, SAY displays the expression on the terminal. The result from the SAY instruction is formatted to the current terminal line width (as defined by the TSO/E TERMINAL command) minus 1 character. In TSO/E background, SAY writes the expression to the output stream, which is SYSTSPRT. In either case, when the length is undefined (LINESIZE() returns 0), SAY uses a default line size of 80.

If an exec runs in a non-TSO/E address space, SAY writes the expression to the output stream as defined by the OUTDD field in the module name table (see “Module name table” on page 326). The system default is SYSTSPRT. The ddname may be changed on an application basis or on a system basis.

Example:

```
data=100
Say data 'divided by 4 =>' data/4
/* Displays: "100 divided by 4 => 25"  */
```

SELECT



SELECT conditionally calls one of several alternative instructions.

Each *expression* after a WHEN is evaluated in turn and must result in 0 or 1. If the result is 1, the instruction following the associated THEN (which may be a complex instruction such as IF, DO, or SELECT) is processed and control then passes to the END. If the result is 0, control passes to the next WHEN clause.

If none of the WHEN expressions evaluates to 1, control passes to the instructions, if any, after OTHERWISE. In this situation, the absence of an OTHERWISE causes an error (but note that you can omit the instruction list that follows OTHERWISE).

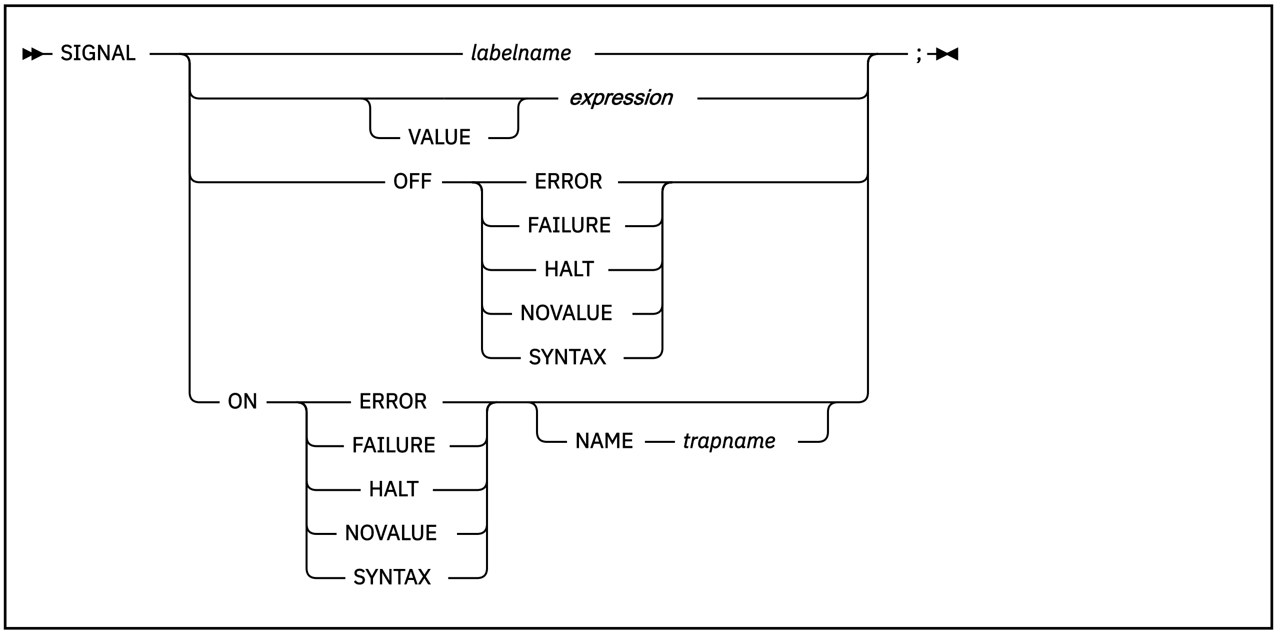
Example:

```
balance=100
check=50
balance = balance - check
Select
  when balance > 0 then
       say 'Congratulations! You still have' balance 'dollars left.'
  when balance = 0 then do
       say 'Attention, Balance is now zero!  STOP all spending.'
       say "You cut it close this month! Hope you do not have any"
       say "checks left outstanding."
       end
  Otherwise
       say "You have just overdrawn your account."
       say "Your balance now shows" balance "dollars."
       say "Oops!  Hope the bank does not close your account."
end  /* Select */
```

Note:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, IF, or the SELECT instruction itself.
2. A null clause is not an instruction, so putting an extra semicolon (or label) after a THEN clause is not equivalent to putting a dummy instruction. The NOP instruction is provided for this purpose.
3. The symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the WHEN clause to be ended by the THEN without a `;` (delimiter) being required.

SIGNAL



SIGNAL causes an *unusual* change in the flow of control (if you specify *labelname* or VALUE *expression*), or controls the trapping of certain conditions (if you specify ON or OFF).

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information about condition traps is contained in Chapter 7, “Conditions and condition traps,” on page 181.

To change the flow of control, a label name is derived from *labelname* or taken from the result of evaluating the *expression* after VALUE. The *labelname* you specify must be a literal string or symbol that is taken as a constant. If you use a symbol for *labelname*, the search is independent of alphabetic case. If you use a literal string, the characters should be in uppercase. This is because the language processor translates all labels to uppercase, regardless of how you enter them in the program. Similarly, for SIGNAL VALUE, the *expression* must evaluate to a string in uppercase or the language processor does not find the label. You can omit the subkeyword VALUE if *expression* does not begin with a symbol or literal string (that is, if it starts with a special character, such as an operator character or parenthesis). All active pending DO, IF, SELECT, and INTERPRET instructions in the current routine are then ended (that is, they cannot be resumed). Control then passes to the first label in the program that matches the given name, as though the search had started from the top of the program.

Example:

```
Signal fred;  /* Transfer control to label FRED below */
  ....
  ....
Fred: say 'Hi!'
```

Because the search effectively starts at the top of the program, if duplicates are present, control always passes to the first occurrence of the label in the program.

When control reaches the specified label, the line number of the SIGNAL instruction is assigned to the special variable SIGL. This can aid debugging because you can use SIGL to determine the source of a transfer of control to a label.

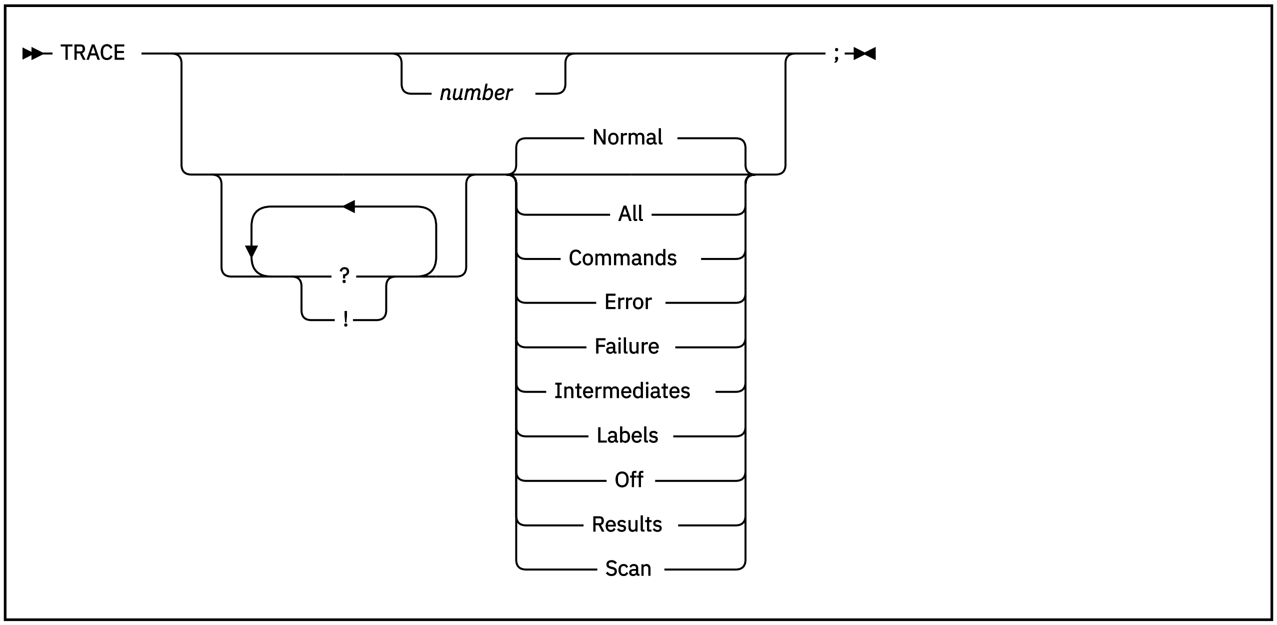
Using SIGNAL VALUE

The VALUE form of the SIGNAL instruction allows a branch to a label whose name is determined at the time of execution. This can safely effect a multi-way CALL (or function call) to internal routines because any DO loops, and so forth, in the calling routine are protected against termination by the call mechanism.

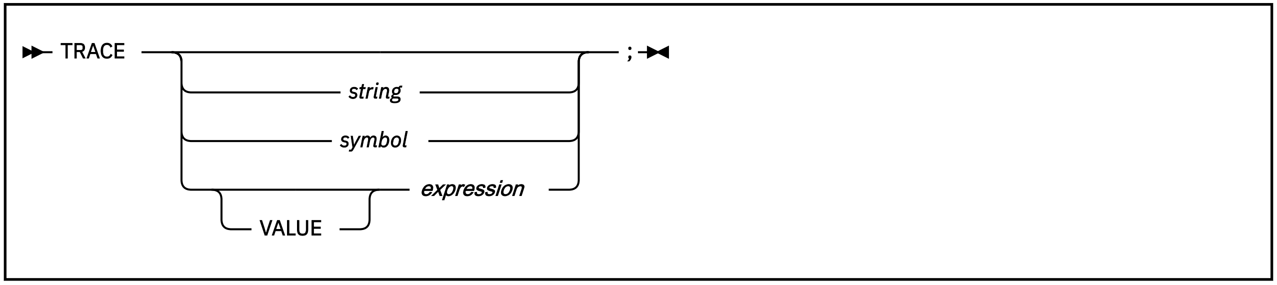
Example:

```
fred='PETE'
call multiway fred, 7
   ....
   ....
exit
Multiway: procedure
   arg label .              /* One word, uppercase                */
                            /* Can add checks for valid labels here */
   signal value label       /* Transfer control to wherever       */
   ....
Pete: say arg(1) '!' arg(2) /* Displays: "PETE ! 7"               */
   return
```

TRACE



Or, alternatively:



TRACE controls the tracing action (that is, how much is displayed to the user) during processing of a REXX program. (Tracing describes some or all of the clauses in a program, producing descriptions of clauses as they are processed.) TRACE is mainly used for debugging. Its syntax is more concise than that of other REXX instructions because TRACE is usually entered manually during interactive debugging. (This is a form of tracing in which the user can interact with the language processor while the program is running.) For this use, economy of key strokes is especially convenient.

If specified, the *number* must be a whole number.

The *string* or *expression* evaluates to:

- A numeric option
- One of the valid prefix or alphabetic character (word) options described later
- Null

The *symbol* is taken as a constant, and is, therefore:

- A numeric option
- One of the valid prefix or alphabetic character (word) options described later

The option that follows TRACE or the result of evaluating *expression* determines the tracing action. You can omit the subkeyword VALUE if *expression* does not begin with a symbol or a literal string (that is, if it starts with a special character, such as an operator or parenthesis).

Alphabetic character (word) options

Although you can enter the word in full, only the capitalized and highlighted letter is needed; all characters following it are ignored. That is why these are referred to as alphabetic character options.

TRACE actions correspond to the alphabetic character options as follows:

All
Traces (that is, displays) all clauses before execution.

Commands
Traces all commands before execution. If the command results in an error or failure, [3] then tracing also displays the return code from the command.

Error
Traces any command resulting in an error or failure [3] after execution, together with the return code from the command.

Failure
Traces any command resulting in a failure [3] after execution, together with the return code from the command. This is the same as the `Normal` option.

Intermediates
Traces all clauses before execution. Also traces intermediate results during evaluation of expressions and substituted names.

Labels
Traces only labels passed during execution. This is especially useful with debug mode, when the language processor pauses after each label. It also helps the user to note all internal subroutine calls and transfers of control because of the SIGNAL instruction.

Normal
Traces any command resulting in a negative return code after execution, together with the return code from the command. **This is the default setting**.

Off
Traces nothing and resets the special prefix options (described later) to OFF. Please consider the Note given with the description of the MSG function (see "MSG" on page 127.

Results
Traces all clauses before execution. Displays final results (contrast with `Intermediates`, preceding) of evaluating an expression. Also displays values assigned during PULL, ARG, and PARSE instructions. **This setting is suggested for general debugging.**

Scan
Traces all remaining clauses in the data without them being processed. Basic checking (for missing ENDs and so forth) is carried out, and the trace is formatted as usual. This is valid only if the TRACE S clause itself is not nested in any other instruction (including INTERPRET or interactive debug) or in an internal routine.

Prefix options

The prefixes `!` and `?` are valid either alone or with one of the alphabetic character options. You can specify both prefixes, in any order, on one TRACE instruction. You can specify a prefix more than one time, if

[3] See "Commands" on page 22 for definitions of error and failure.

desired. Each occurrence of a prefix on an instruction reverses the action of the previous prefix. The prefix(es) must immediately precede the option (no intervening blanks).

The prefixes `!` and `?` modify tracing and execution as follows:

!

Inhibits host command execution. During regular execution, a TRACE instruction with a prefix of `!` suspends execution of all subsequent host commands. For example, `TRACE !C` causes commands to be traced but not processed. As each command is bypassed, the REXX special variable RC is set to `0`. You can use this action for debugging potentially destructive programs. (Note that this does not inhibit any commands entered manually while in interactive debug. These are always processed.)

You can switch off command inhibition, when it is in effect, by issuing a TRACE instruction with a prefix `!`. Repeated use of the `!` prefix, therefore, switches you alternately in or out of command inhibition mode. Or, you can turn off command inhibition at any time by issuing `TRACE O` or TRACE with no options.

?

Controls interactive debug. During usual execution, a TRACE option with a prefix of `?` causes interactive debug to be switched on. (See "Interactive debugging of programs" on page 233 for full details of this facility.) While interactive debug is on, interpretation pauses after most clauses that are traced. For example, the instruction `TRACE ?E` makes the language processor pause for input after executing any command that returns an error (that is, a nonzero return code).

Any TRACE instructions in the program being traced are ignored. (This is so that you are not taken out of interactive debug unexpectedly.)

You can switch off interactive debug in several ways:

- Entering `TRACE O` turns off all tracing.
- Entering TRACE with no options restores the defaults—it turns off interactive debug but continues tracing with TRACE Normal (which traces any failing command after execution) in effect.
- Entering `TRACE ?` turns off interactive debug and continues tracing with the current option.
- Entering a TRACE instruction with a `?` prefix before the option turns off interactive debug and continues tracing with the new option.

Using the `?` prefix, therefore, switches you alternately in or out of interactive debug. (Because the language processor ignores any further TRACE statements in your program after you are in interactive debug, use `CALL TRACE '?'` to turn off interactive debug.)

Tip: The TSO/E REXX immediate command TS and the EXECUTIL TS command can also be used to enter interactive debug. See Chapter 10, "TSO/E REXX commands," on page 201.

Numeric options

If interactive debug is active *and* if the option specified is a positive whole number (or an expression that evaluates to a positive whole number), that number indicates the number of debug pauses to be skipped over. (See separate section in "Interactive debugging of programs" on page 233, for further information.) However, if the option is a negative whole number (or an expression that evaluates to a negative whole number), all tracing, including debug pauses, is temporarily inhibited for the specified number of clauses. For example, `TRACE -100` means that the next 100 clauses that would usually be traced are not, in fact, displayed. After that, tracing resumes as before.

Tracing tips

1. When a loop is being traced, the DO clause itself is traced on every iteration of the loop.
2. You can retrieve the trace actions currently in effect by using the TRACE built-in function (see "TRACE" on page 102).

3. If available at the time of execution, comments associated with a traced clause are included in the trace, as are comments in a null clause, if you specify TRACE A, R, I, or S.
4. Commands traced before execution always have the final value of the command (that is, the string passed to the environment), and the clause generating it produced in the traced output.
5. Trace actions are automatically saved across subroutine and function calls. See "CALL" on page 43 for more details.

A typical example

One of the most common traces you will use is:

```
TRACE ?R
/* Interactive debug is switched on if it was off, */
/*  and tracing Results of expressions begins.     */
```

Tracing may be switched on, without requiring modification to a program, by using the EXECUTIL TS command. Tracing may also be turned on or off asynchronously, (that is, while an exec is running) using the TS and TE immediate commands from attention mode. See "Interrupting exec processing" on page 235 for the description of these facilities.

Format of TRACE output

Every clause traced appears with automatic formatting (indentation) according to its logical depth of nesting and so forth. The language processor may replace any control codes in the encoding of data (for example, EBCDIC values less than '40'x) with a question mark (?) to avoid console interference. Results (if requested) are indented an extra two spaces and are enclosed in double quotation marks so that leading and trailing blanks are apparent.

A line number precedes the first clause traced on any line. If the line number is greater than 99999, the language processor truncates it on the left, and the ? prefix indicates the truncation. For example, the line number 100354 appears as ?00354. All lines displayed during tracing have a three-character prefix to identify the type of data being traced. These can be:

-
Identifies the source of a single clause, that is, the data actually in the program.

+++
Identifies a trace message. This may be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error when in interactive debug, or the traceback clauses after a syntax error in the program (see below).

>>>
Identifies the result of an expression (for TRACE R) or the value assigned to a variable during parsing, or the value returned from a subroutine call.

>.>
Identifies the value "assigned" to a placeholder during parsing (see "The period as a placeholder" on page 158).

The following prefixes are used only if TRACE Intermediates is in effect:

>C>
The data traced is the name of a compound variable, traced after substitution and before use, provided that the name had the value of a variable substituted into it.

>F>
The data traced is the result of a function call.

>L>
The data traced is a literal (string, uninitialized variable, or constant symbol).

>O>
The data traced is the result of an operation on two terms.

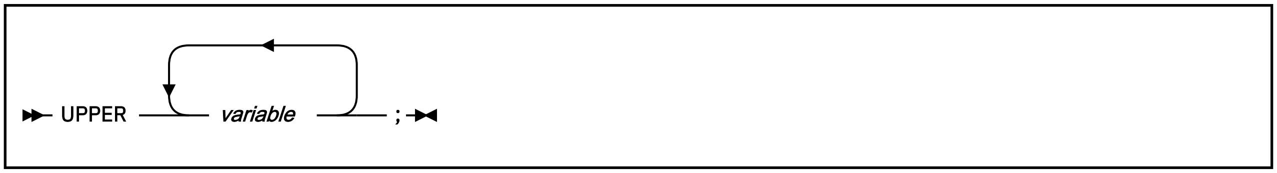
>P>
The data traced is the result of a prefix operation.

>V>
The data traced is the contents of a variable.

If no option is specified on a TRACE instruction, or if the result of evaluating the expression is null, the default tracing actions are restored. The defaults are TRACE N, command inhibition (!) off, and interactive debug (?) off.

Following a syntax error that SIGNAL ON SYNTAX does not trap, the clause in error is always traced. Any CALL or INTERPRET or function invocations active at the time of the error are also traced. If an attempt to transfer control to a label that could not be found caused the error, that label is also traced. The special trace prefix +++ identifies these traceback lines.

UPPER



UPPER translates the contents of one or more variables to uppercase. The variables are translated in sequence from left to right.

The *variable* is a symbol, separated from any other *variable*s by one or more blanks or comments. Specify only simple symbols and compound symbols. (See “Simple symbols” on page 19.)

Using this instruction is more convenient than repeatedly invoking the TRANSLATE built-in function.

Example:

```
a1='Hello';  b1='there'
Upper a1 b1
say a1 b1     /* Displays "HELLO THERE" */
```

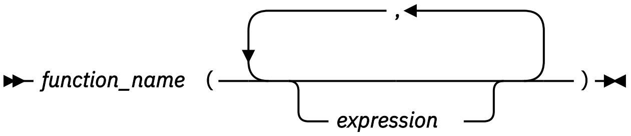
An error is signalled if a constant symbol or a stem is encountered. Using an uninitialized variable is *not* an error, and has no effect, except that it is trapped if the NOVALUE condition (SIGNAL ON NOVALUE) is enabled.

Chapter 4. Functions

A **function** is an internal, built-in, or external routine that returns a single result string. (A **subroutine** is a function that is an internal, built-in, or external routine that may or may not return a result and that is called with the CALL instruction.)

Syntax

A **function call** is a term in an expression that calls a routine that carries out some procedures and returns a string. This string replaces the function call in the continuing evaluation of the expression. You can include function calls to internal and external routines in an expression anywhere that a data term (such as a string) would be valid, using the notation:



The *function_name* is a literal string or a single symbol, which is taken to be a constant.

There can be up to an implementation-defined maximum number of expressions, separated by commas, between the parentheses. In TSO/E, the implementation maximum is up to 20 expressions. These expressions are called the **arguments** to the function. Each argument expression may include further function calls.

Note that the left parenthesis must be adjacent to the name of the function, with no blank in between, or the construct is not recognized as a function call. (A blank operator would be assumed at this point instead.) Only a comment (which has no effect) can appear between the name and the left parenthesis.

The arguments are evaluated in turn from left to right and the resulting strings are all then passed to the function. This then runs some operation (usually dependent on the argument strings passed, though arguments are not mandatory) and eventually returns a single character string. This string is then included in the original expression just as though the entire function reference had been replaced by the name of a variable whose value is that returned data.

For example, the function SUBSTR is built-in to the language processor (see "SUBSTR (Substring)" on page 99) and could be used as:

```
N1='abcdefghijk'
Z1='Part of N1 is: 'substr(N1,2,7)
/* Sets Z1 to 'Part of N1 is: bcdefgh' */
```

A function may have a variable number of arguments. You need to specify only those that are required. For example, `SUBSTR('ABCDEF',4)` would return DEF.

Functions and subroutines

The function calling mechanism is identical with that for subroutines. The only difference between functions and subroutines is that functions must return data, whereas subroutines need not.

The following types of routines can be called as functions:

Internal
If the routine name exists as a label in the program, the current processing status is saved, so that it is later possible to return to the point of invocation to resume execution. Control is then passed to the first label in the program that matches the name. As with a routine called by the CALL instruction, various other status information (TRACE and NUMERIC settings and so forth) is saved too. See "CALL" on page 43 for details about this. You can use SIGNAL and CALL together to call an internal routine

whose name is determined at the time of execution; this is known as a multi-way call (see "SIGNAL" on page 66).

If you are calling an internal routine as a function, you *must* specify an expression in any RETURN instruction to return from it. This is not necessary if it is called as a subroutine.

Example:

```
/* Recursive internal function execution... */
arg x
say x'! =' factorial(x)
exit

factorial: procedure   /* Calculate factorial by    */
  arg n                /*   recursive invocation.   */
  if n=0 then return 1
  return  factorial(n-1) * n
```

FACTORIAL is unusual in that it calls itself (this is recursive invocation). The PROCEDURE instruction ensures that a new variable n is created for each invocation.

Note: When there is a search for a routine, the language processor currently scans the statements in the REXX program to locate the internal label. During the search, the language processor may encounter a syntax error. As a result, a syntax error may be raised on a statement different from the original line being processed.

Built-in

These functions are always available and are defined in the next section of this manual. (See "Built-in functions" on page 77 to "X2D (Hexadecimal to Decimal)" on page 108.)

External

You can write or use functions that are external to your program and to the language processor. An external routine can be written in any language (including REXX) that supports the system-dependent interfaces the language processor uses to call it. You can call a REXX program as a function and, in this case, pass more than one argument string. The ARG or PARSE ARG instructions or the ARG built-in function can retrieve these argument strings. When called as a function, a program must return data to the caller. For information about writing external functions and subroutines and the system dependent interfaces, see "External functions and subroutines, and function packages" on page 263.

Note:

1. Calling an external REXX program as a function is similar to calling an internal routine. The external routine is, however, an implicit PROCEDURE in that all the caller's variables are always hidden and the status of internal values (NUMERIC settings and so forth) start with their defaults (rather than inheriting those of the caller).
2. Other REXX programs can be called as functions. You can use either EXIT or RETURN to leave the called REXX program, and in either case you must specify an expression.
3. With care, you can use the INTERPRET instruction to process a function with a variable function name. However, you should avoid this if possible because it reduces the clarity of the program.

Search order

The search order for functions is: internal routines take precedence, then built-in functions, and finally external functions.

Internal routines are *not* used if the function name is given as a literal string (that is, specified in quotation marks); in this case the function must be built-in or external. This lets you usurp the name of, say, a built-in function to extend its capabilities, yet still be able to call the built-in function when needed.

Example:

```
/* This internal DATE function modifies the          */
/* default for the DATE function to standard date.   */
date: procedure
      arg in
      if in='' then in='Standard'
      return 'DATE'(in)
```

Built-in functions have uppercase names, and so the name in the literal string must be in uppercase for the search to succeed, as in the example. The same is usually true of external functions. The search order for **external functions** and **subroutines** follows.

1. Check the following function packages defined for the language processor environment:
   - User function packages
   - Local function packages
   - System function packages
2. If a match to the function name is not found, the function search order flag (FUNCSOFL) is checked. The FUNCSOFL flag (see "Flags and corresponding masks" on page 322) indicates whether load libraries are searched before the search for a REXX exec.

   If the flag is off, check the load libraries. If a match to the function name is not found, search for a REXX program.

   If the flag is on, search for a REXX program. If a match to the function name is not found, check the load libraries.

   By default, the FUNCSOFL flag is off, which means that load libraries are searched before the search for a REXX exec.

   You can use TSO/E EXECUTIL RENAME to change functions in a function package directory. For more information, see EXECUTIL RENAME.
3. TSO/E uses the following order to search the load libraries:
   - Job pack area
   - ISPLLIB. If the user entered *LIBDEF ISPLLIB ...*, the system searches the new alternate library defined by LIBDEF followed by the ISPLLIB library.
   - Task library and all preceding task libraries
   - Step library. If there is no step library, the job library is searched, if one exists.
   - Link pack area (LPA)
   - Link library
4. The following list describes the steps used to search for a REXX exec for a function or subroutine call:

   Restriction: VLF is not searched for REXX execs called as functions or subroutines.

   a. Search the ddname from which the exec that is calling the function or subroutine was loaded. For example, if the calling exec was loaded from the DD MYAPPL, the system searches MYAPPL for the function or subroutine.

      Note: If the calling exec is running in a non-TSO/E address space and the exec (function or subroutine) being searched for was not found, the search for an exec ends. Note that depending on the setting of the FUNCSOFL flag, the load libraries may or may not have already been searched at this point.

   b. Search any exec libraries as defined by the TSO/E ALTLIB command

   c. Check the setting of the NOLOADDD flag (see "Flags and corresponding masks" on page 322).

      - If the NOLOADDD flag is off, search any data sets that are allocated to SYSEXEC. (SYSEXEC is the default system file in which you can store REXX execs; it is the default ddname specified in the LOADDD field in the module name table. See "Module name table" on page 326).

        If the function or subroutine is not found, search the data sets allocated to SYSPROC. If the function or subroutine is not found, the search for an exec ends. Note that depending on the setting of the FUNCSOFL flag, the load libraries may or may not have already been searched at this point.

- If the NOLOADDD flag is on, search any data sets that are allocated to SYSPROC. If the function or subroutine is not found, the search for an exec ends. Note that depending on the setting of the FUNCSOFL flag, the load libraries may or may not have already been searched at this point.

Note: With the defaults that TSO/E provides, the NOLOADDD flag is off. This means that SYSEXEC is searched before SYSPROC.

You can control the NOLOADDD flag using the TSO/E REXX EXECUTIL command. For more information, see "EXECUTIL" on page 216.

Figure 8 on page 76 illustrates how a call to an external function or subroutine is handled. After the user, local, and system function packages, and optionally, the load libraries are searched, if the function or subroutine was not found, the system searches for a REXX exec. The search for an exec is shown in part 2 of the figure.

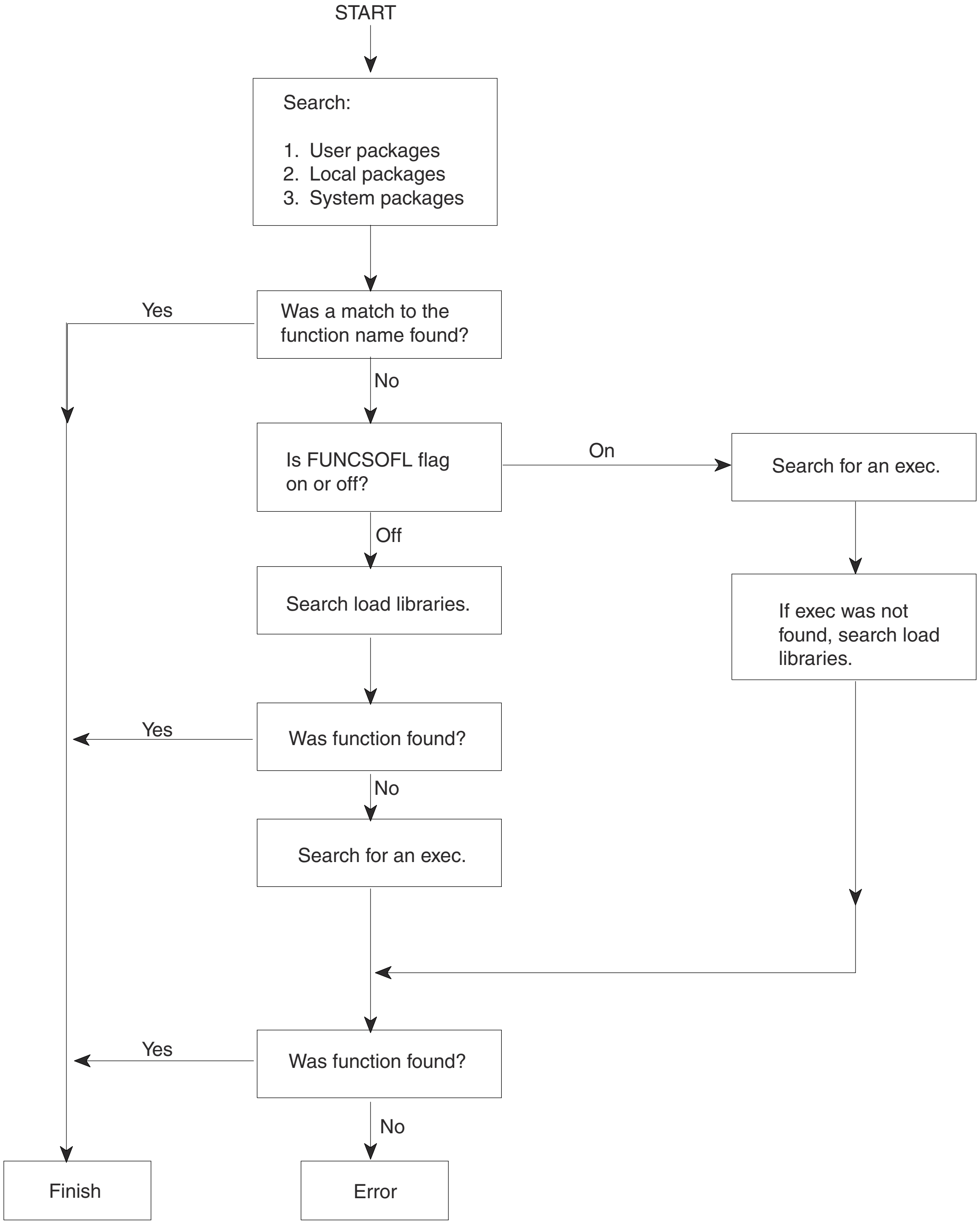


Figure 8. External routine resolution and execution Part 1

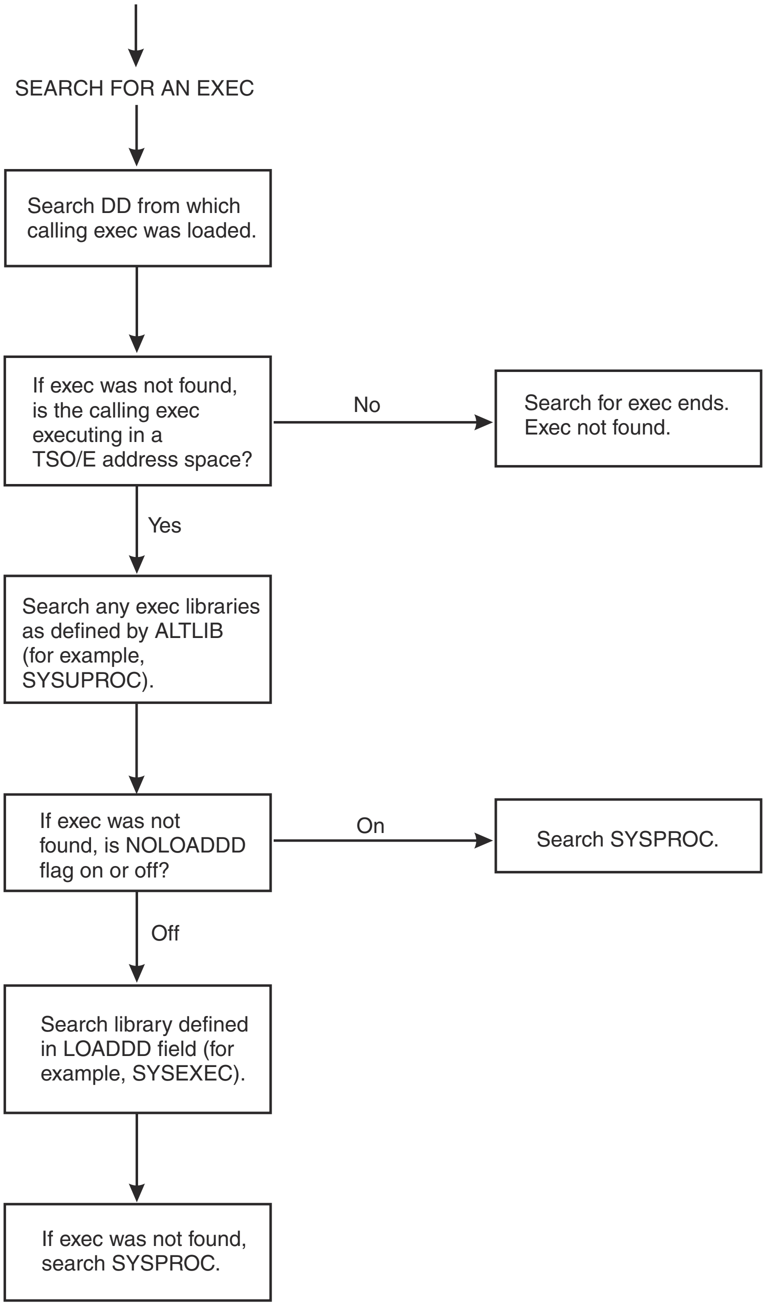

SEARCH FOR AN EXEC
Search DD from which calling exec was loaded.
If exec was not found, is the calling exec executing in a TSO/E address space?
No
Search for exec ends. Exec not found.
Yes
Search any exec libraries as defined by ALTLIB (for example, SYSUPROC).
If exec was not found, is NOLOADDD flag on or off?
On
Search SYSPROC.
Off
Search library defined in LOADDD field (for example, SYSEXEC).
If exec was not found, search SYSPROC.

Figure 9. External routine resolution and execution Part 2

Errors during execution

If an external or built-in function detects an error of any kind, the language processor is informed, and a syntax error results. Execution of the clause that included the function call is, therefore, ended. Similarly, if an external function fails to return data correctly, the language processor detects this and reports it as an error.

If a syntax error occurs during the execution of an internal function, it can be trapped (using SIGNAL ON SYNTAX) and recovery may then be possible. If the error is not trapped, the program is ended.

Built-in functions

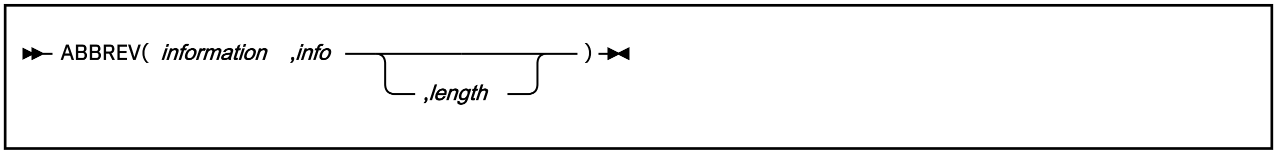
REXX provides a rich set of built-in functions, including character manipulation, conversion, and information functions.

There are six other built-in functions that TSO/E provides: EXTERNALS, FIND, INDEX, JUSTIFY, LINESIZE, and USERID. If you plan to write REXX programs that run on other SAA environments, note that these functions are not available to all the environments. In this section, these six built-in functions are identified as non-SAA functions.

In addition to the built-in functions, TSO/E also provides TSO/E external functions that you can use to perform different tasks. These functions are described in "TSO/E external functions" on page 109. The following are general notes on the built-in functions:

- The parentheses in a function are always needed, even if no arguments are required. The first parenthesis must follow the name of the function with no space in between.
- The built-in functions work internally with NUMERIC DIGITS 9 and NUMERIC FUZZ 0 and are unaffected by changes to the NUMERIC settings, except where stated. This is not true for RANDOM.
- Any argument named as a *string* may be a null string.
- If an argument specifies a *length*, it must be a positive whole number or zero. If it specifies a start character or word in a string, it must be a positive whole number, unless otherwise stated.
- Where the last argument is optional, you can always include a comma to indicate you have omitted it; for example, `DATATYPE(1,)`, like `DATATYPE(1)`, would return `NUM`.
- If you specify a *pad* character, it must be exactly one character long. (A pad character extends a string, usually on the right. For an example, see the LEFT built-in function on page "LEFT" on page 93.)
- If a function has an *option* you can select by specifying the first character of a string, that character can be in upper- or lowercase.
- A number of the functions described in this chapter support DBCS. A complete list and descriptions of these functions are in Appendix A, "Double-byte character set (DBCS) support," on page 429.

ABBREV (Abbreviation)



returns 1 if *info* is equal to the leading characters of *information* **and** the length of *info* is not less than *length*. Returns 0 if either of these conditions is not met.

If you specify *length*, it must be a positive whole number or zero. The default for *length* is the number of characters in *info*.

Here are some examples:

```
ABBREV('Print','Pri')      ->    1
ABBREV('PRINT','Pri')      ->    0
ABBREV('PRINT','PRI',4)    ->    0
ABBREV('PRINT','PRY')      ->    0
ABBREV('PRINT','')         ->    1
ABBREV('PRINT','',1)       ->    0
```

Example: A null string always matches if a length of 0 is used. This allows a default keyword to be selected automatically, when no value is specified, as shown in this example.

Here if option is the null string, keyword1 is selected.

```
say 'Enter option:';   pull option .
select  /* keyword1 is to be the default */
  when abbrev('keyword1',option) then ...
  when abbrev('keyword2',option) then ...
  when (abbrev("ONE",option))THEN CALL OPT1 /*default, if option is a null string*/
  OTHERWISE CALL OPTERROR    /* Invalid option found */

  ...
  otherwise nop;
end;
```

ABS (Absolute Value)

returns the absolute value of *number*. The result has no sign and is formatted according to the current NUMERIC settings.

Here are some examples:

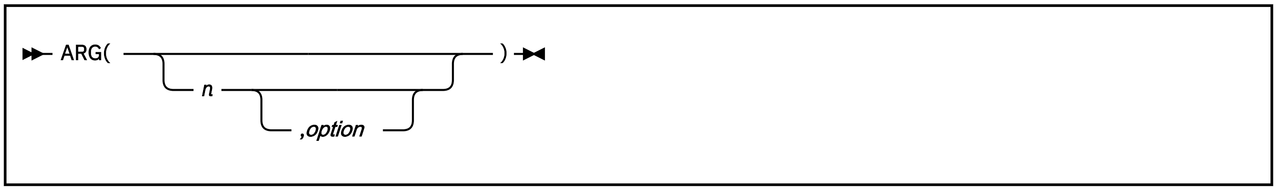
```
ABS('12.3')       ->    12.3
ABS(' -0.307')    ->    0.307
```

ADDRESS

returns the name of the environment to which commands are currently being submitted. See "ADDRESS" on page 41) for more information. Trailing blanks are removed from the result. Here are some examples:

```
ADDRESS()    ->    'TSO'        /* default under TSO/E  */
ADDRESS()    ->    'MVS'        /* default under MVS    */
```

ARG (Argument)



returns an argument string or information about the argument strings to a program or internal routine.

If you do not specify *n*, the number of arguments passed to the program or internal routine is returned.

If you specify only *n*, the *n*th argument string is returned. If the argument string does not exist, the null string is returned. The *n* must be a positive whole number.

If you specify *option*, ARG tests for the existence of the *n*th argument string. The following are valid *option*s. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Exists
returns 1 if the *n*th argument exists; that is, if it was explicitly specified when the routine was called. Returns 0 otherwise.

Omitted
returns 1 if the *n*th argument was omitted; that is, if it was *not* explicitly specified when the routine was called. Returns 0 otherwise.

Here are some examples:

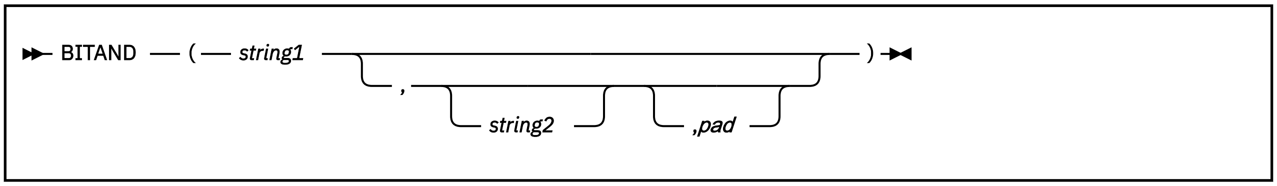
```
/*  following "Call name;" (no arguments) */
ARG()         ->    0
ARG(1)        ->    ''
ARG(2)        ->    ''
ARG(1,'e')    ->    0
ARG(1,'O')    ->    1
```

```
/*  following "Call name 'a',,'b';" */
ARG()         ->    3
ARG(1)        ->    'a'
ARG(2)        ->    ''
ARG(3)        ->    'b'
ARG(n)        ->    ''    /* for n>=4 */
ARG(1,'e')    ->    1
ARG(2,'E')    ->    0
ARG(2,'O')    ->    1
ARG(3,'o')    ->    0
ARG(4,'o')    ->    1
```

Note:

1. The number of argument strings is the largest number *n* for which `ARG(n,'e')` would return `1` or `0` if there are no explicit argument strings. That is, it is the position of the last explicitly specified argument string.
2. Programs called as commands can have only 0 or 1 argument strings. The program has 0 argument strings if it is called with the name only and has 1 argument string if anything else (including blanks) is included with the command.
3. You can retrieve and directly parse the argument strings to a program or internal routine with the ARG or PARSE ARG instructions. (See "ARG" on page 42, "PARSE" on page 57, and Chapter 5, "Parsing," on page 157.)

BITAND (Bit by Bit AND)

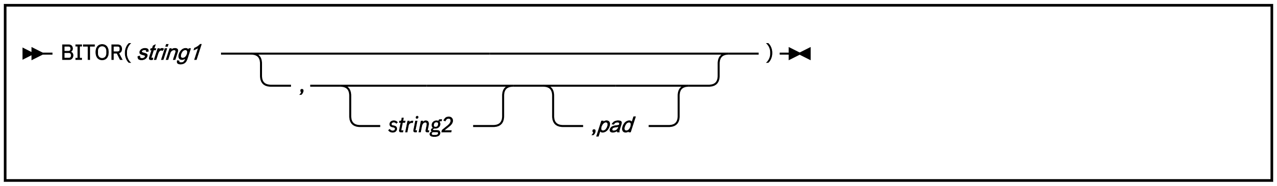


returns a string composed of the two input strings logically ANDed together, bit by bit. (The encoding of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the AND operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero length (null) string.

Here are some examples:

```
BITAND('12'x)                 ->    '12'x
BITAND('73'x,'27'x)           ->    '23'x
BITAND('13'x,'5555'x)         ->    '1155'x
BITAND('13'x,'5555'x,'74'x)   ->    '1154'x
BITAND('pQrS',,'BF'x)         ->    'pqrs'      /* EBCDIC  */
```

BITOR (Bit by Bit OR)

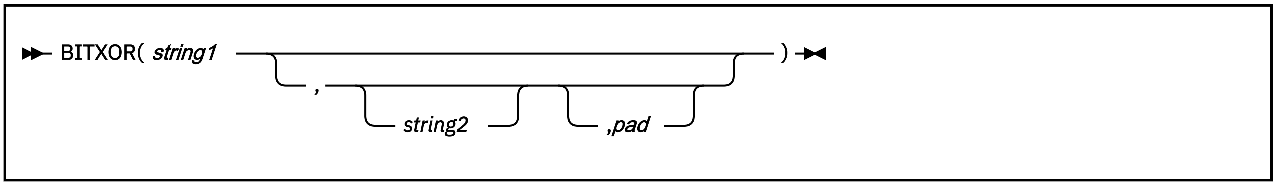


returns a string composed of the two input strings logically inclusive-ORed together, bit by bit. (The encoding of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the OR operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero length (null) string.

Here are some examples:

```
BITOR('12'x)                  ->    '12'x
BITOR('15'x,'24'x)            ->    '35'x
BITOR('15'x,'2456'x)          ->    '3556'x
BITOR('15'x,'2456'x,'F0'x)    ->    '35F6'x
BITOR('1111'x,,'4D'x)         ->    '5D5D'x
BITOR('Fred',,'40'x)          ->    'FRED'      /* EBCDIC     */
```

BITXOR (Bit by Bit Exclusive OR)

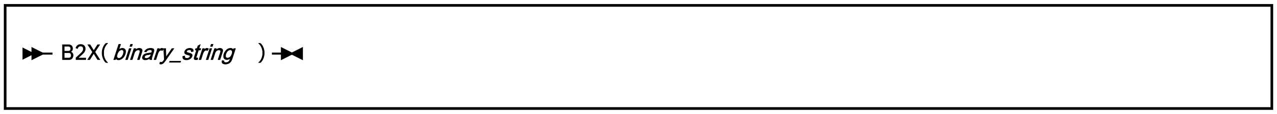


returns a string composed of the two input strings logically eXclusive-ORed together, bit by bit. (The encoding of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the XOR operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero length (null) string.

Here are some examples:

```
BITXOR('12'x)                     ->  '12'x
BITXOR('12'x,'22'x)               ->  '30'x
BITXOR('1211'x,'22'x)             ->  '3011'x
BITXOR('1111'x,'444444'x)         ->  '555544'x
BITXOR('1111'x,'444444'x,'40'x)   ->  '555504'x
BITXOR('1111'x,,'4D'x)            ->  '5C5C'x
BITXOR('C711'x,'222222'x,' ')     ->  'E53362'x  /* EBCDIC */
```

B2X (Binary to Hexadecimal)



returns a string, in character format, that represents *binary_string* converted to hexadecimal.

The *binary_string* is a string of binary (0 or 1) digits. It can be of any length. You can optionally include blanks in *binary_string* (at four-digit boundaries only, not leading or trailing) to aid readability; they are ignored.

The returned string uses uppercase alphabetics for the values A–F, and does not include blanks.

If *binary_string* is the null string, B2X returns a null string. If the number of binary digits in *binary_string* is not a multiple of four, then up to three 0 digits are added on the left before the conversion to make a total that is a multiple of four.

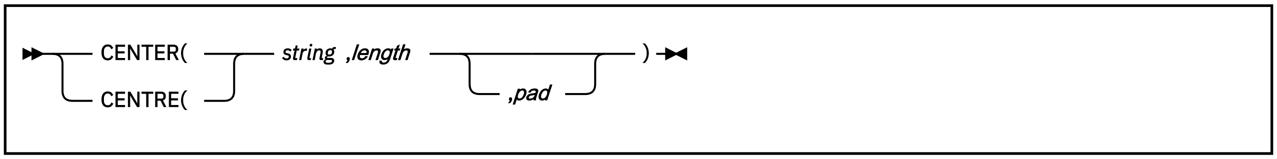
Here are some examples:

```
B2X('11000011')    ->   'C3'
B2X('10111')       ->   '17'
B2X('101')         ->   '5'
B2X('1 1111 0000') ->   '1F0'
```

You can combine B2X with the functions X2D and X2C to convert a binary number into other forms. For example:

```
X2D(B2X('10111'))  ->   '23'   /* decimal 23 */
```

CENTER/CENTRE



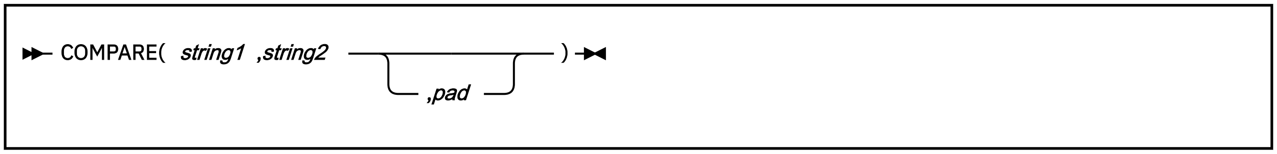
returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up length. The *length* must be a positive whole number or zero. The default *pad* character is blank. If the string is longer than *length*, it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand end loses or gains one more character than the left-hand end.

Here are some examples:

```
CENTER(abc,7)               ->    '  ABC  '
CENTER(abc,8,'-')           ->    '--ABC---'
CENTRE('The blue sky',8)    ->    'e blue s'
CENTRE('The blue sky',7)    ->    'e blue '
```

To avoid errors because of the difference between British and American spellings, this function can be called either CENTRE or CENTER.

COMPARE

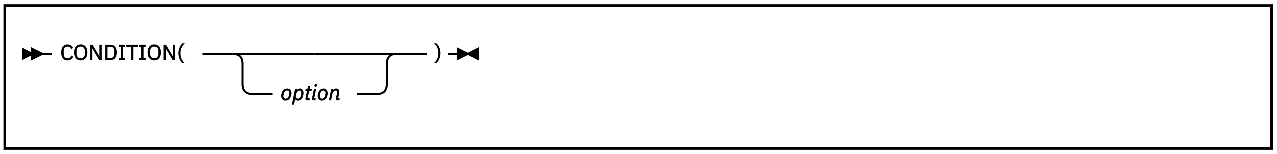


returns `0` if the strings, *string1* and *string2*, are identical. Otherwise, returns the position of the first character that does not match. The shorter string is padded on the right with *pad* if necessary. The default *pad* character is a blank.

Here are some examples:

```
COMPARE('abc','abc')         ->    0
COMPARE('abc','ak')          ->    2
COMPARE('ab ','ab')          ->    0
COMPARE('ab ','ab',' ')      ->    0
COMPARE('ab ','ab','x')      ->    3
COMPARE('ab-- ','ab','-')    ->    5
```

CONDITION



returns the condition information associated with the current trapped condition. (See Chapter 7, “Conditions and condition traps,” on page 181 for a description of condition traps.) You can request the following pieces of information:

- The name of the current trapped condition
- Any descriptive string associated with that condition
- The instruction processed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition.

To select the information to return, use the following *option*s. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Condition name
returns the name of the current trapped condition.

Description
returns any descriptive string associated with the current trapped condition. See "Descriptive strings" on page 184 for the list of possible strings. If no description is available, returns a null string.

Instruction
returns either `CALL` or `SIGNAL`, the keyword for the instruction processed when the current condition was trapped. This is the default if you omit *option*.

Status
returns the status of the current trapped condition. This can change during processing, and is either:

`ON` - the condition is enabled
`OFF` - the condition is disabled
`DELAY` - any new occurrence of the condition is delayed or ignored.

If no condition has been trapped, then the CONDITION function returns a null string in all four cases.

Here are some examples:

```
CONDITION()            ->    'CALL'        /* perhaps */
CONDITION('C')         ->    'FAILURE'
CONDITION('I')         ->    'CALL'
CONDITION('D')         ->    'FailureTest'
CONDITION('S')         ->    'OFF'        /* perhaps */
```

The CONDITION function returns condition information that is saved and restored across subroutine calls (including those a CALL ON condition trap causes). Therefore, after a subroutine called with CALL ON *trapname* has returned, the current trapped condition reverts to the condition that was current before the CALL took place (which may be none). CONDITION returns the values it returned before the condition was trapped.

COPIES

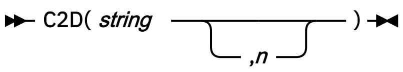
```
►►─ COPIES( string ,n) ─►◄
```

returns *n* concatenated copies of *string*. The *n* must be a positive whole number or zero.

Here are some examples:

```
COPIES('abc',3)    ->    'abcabcabc'
COPIES('abc',0)    ->    ''
```

C2D (Character to Decimal)



returns the decimal value of the binary representation of *string*. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS.

If *string* is null, returns `0`.

Here are some examples:

```
C2D('09'X)      ->       9
C2D('81'X)      ->     129
C2D('FF81'X)    ->   65409
```

```
C2D('')          ->        0
C2D('a')         ->      129     /*  EBCDIC  */
```

If you specify *n*, the string is taken as a signed number expressed in *n* characters. The number is positive if the leftmost bit is off, and negative, in two's complement notation, if the leftmost bit is on. In both cases, it is converted to a whole number, which may, therefore, be negative. The *string* is padded on the left with '00'x characters (note, not “sign-extended”), or truncated on the left to *n* characters. This padding or truncation is as though RIGHT(*string*,*n*,'00'x) had been processed. If *n* is 0, C2D always returns 0. If you do not specify *n*, *string* is processed as an unsigned binary number.

Here are some examples:

```
C2D('81'X,1)      ->     -127
C2D('81'X,2)      ->      129
C2D('FF81'X,2)    ->     -127
C2D('FF81'X,1)    ->     -127
C2D('FF7F'X,1)    ->      127
C2D('F081'X,2)    ->    -3967
C2D('F081'X,1)    ->     -127
C2D('0031'X,0)    ->        0
```

Implementation maximum: The input string cannot have more than 250 characters that are significant in forming the final result. Leading sign characters ('00'x and 'FF'x) do not count toward this total.

C2X (Character to Hexadecimal)

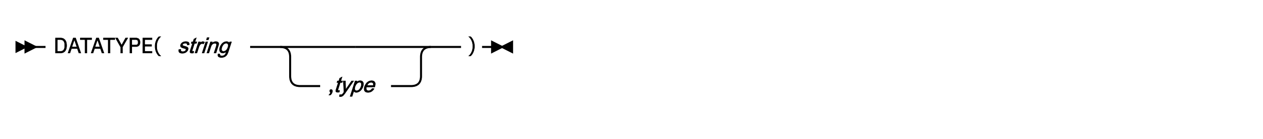
returns a string, in character format, that represents *string* converted to hexadecimal. The returned string contains twice as many bytes as the input string. For example, on an EBCDIC system, C2X(1) returns F1 because the EBCDIC representation of the character 1 is 'F1'X.

The string returned uses uppercase alphabetics for the values A–F and does not include blanks. The *string* can be of any length. If *string* is null, returns a null string.

Here are some examples:

```
C2X('72s')      ->    'F7F2A2' /* 'C6F7C6F2C1F2'X in EBCDIC */
C2X('0123'X)    ->    '0123'   /* 'F0F1F2F3'X     in EBCDIC */
```

DATATYPE



returns NUM if you specify only *string* and if *string* is a valid REXX number that can be added to 0 without error; returns CHAR if *string* is not a valid number.

If you specify *type*, returns 1 if *string* matches the type; otherwise returns 0. If *string* is null, the function returns 0 (except when *type* is X, which returns 1 for a null string). The following are valid *type*s. (Only the capitalized and highlighted letter is needed; all characters following it are ignored. Note that for the hexadecimal option, you must start your string specifying the name of the option with x rather than h.)

Alphanumeric
returns 1 if *string* contains only characters from the ranges a–z, A–Z, and 0–9.

Binary
returns 1 if *string* contains only the characters 0 or 1 or both.

C
returns 1 if *string* is a mixed SBCS/DBCS string.

Dbcs
returns 1 if *string* is a DBCS-only string enclosed by SO and SI bytes.

Lowercase
returns 1 if *string* contains only characters from the range `a`–`z`.

Mixed case
returns 1 if *string* contains only characters from the ranges `a`–`z` and `A`–`Z`.

Number
returns 1 if *string* is a valid REXX number.

Symbol
returns 1 if *string* contains only characters that are valid in REXX symbols. (See "Tokens" on page 8.) Note that both uppercase and lowercase alphabetics are permitted.

Uppercase
returns 1 if *string* contains only characters from the range `A`–`Z`.

Whole number
returns 1 if *string* is a REXX whole number under the current setting of NUMERIC DIGITS.

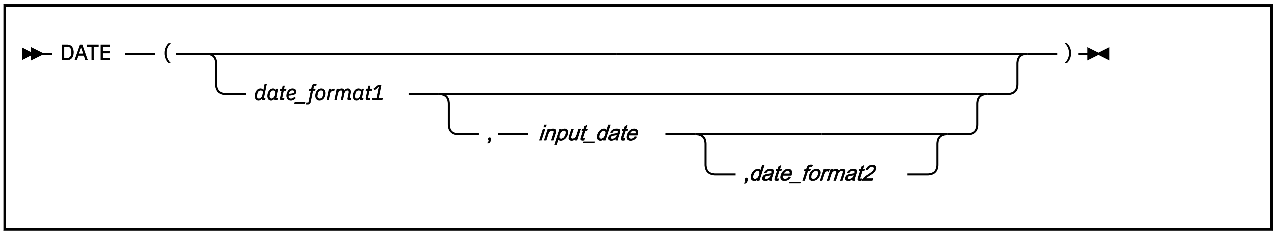
heXadecimal
returns 1 if *string* contains only characters from the ranges `a`–`f`, `A`–`F`, `0`–`9`, and blank (as long as blanks appear only between pairs of hexadecimal characters). Also returns 1 if *string* is a null string, which is a valid hexadecimal string.

Here are some examples:

```
DATATYPE(' 12 ')         ->   'NUM'
DATATYPE('')             ->   'CHAR'
DATATYPE('123*')         ->   'CHAR'
DATATYPE('12.3','N')     ->    1
DATATYPE('12.3','W')     ->    0
DATATYPE('Fred','M')     ->    1
DATATYPE('','M')         ->    0
DATATYPE('Fred','L')     ->    0
DATATYPE('?20K','s')     ->    1
DATATYPE('BCd3','X')     ->    1
DATATYPE('BC d3','X')    ->    1
```

The DATATYPE function tests the meaning or type of characters in a string, independent of the encoding of those characters (for example, ASCII or EBCDIC).

DATE



returns, by default, the local date in the format: *dd mon yyyy* (day, month, year—for example, `25 Dec 2001`), with no leading zero or blank on the day. Otherwise, the string *input_date* is converted to the format specified by *date_format1*. *date_format2* can be specified to define the current format of *input_date*. The default for *date_format1* and *date_format2* is **N**ormal. *input_date* must not have a leading zero or blank.

You can use the following options to obtain specific date formats. (Only the bold character is needed; all other characters are ignored.)

Base
the number of complete days (that is, not including the current day) since and including the base date, 1 January 0001, in the format: *dddddd* (no leading zeros or blanks). The expression `DATE('B')//7`

returns a number in the range 0–6 that corresponds to the current day of the week, where 0 is Monday and 6 is Sunday.

Thus, this function can be used to determine the day of the week independent of the national language in which you are working.

Note: The base date of 1 January 0001 is determined by extending the current Gregorian calendar backward (365 days each year, with an extra day every year that is divisible by 4 except century years that are not divisible by 400). It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

Century

the number of days, including the current day, since and including January 1 of the last year that is a multiple of 100 in the form: *ddddd* (no leading zeros). Example: A call to DATE(C) on March 13, 1992, returns 33675, the number of days from 1 January 1900 to 13 March 1992. Similarly, a call to DATE(C) on November 20, 2001, returns 690, the number of days from 1 January 2000 to 20 November 2001.

Note: When used for *date_format1*, this option is valid when *input_date* is not specified.

Days

the number of days, including the current day, so far in this year in the format: *ddd* (no leading zeros or blanks).

European

date in the format: *dd/mm/yy*

Julian

date in the format: *yyddd*.

Note: When used for *date_format1*, this option is valid only when *input_date* is not specified.

Month

full English name of the current month, in mixed case—for example, `August`. Only valid for *date_format1*.

Normal

date in the format: *dd mon yyyy*, in mixed case. **This is the default**. If the active language has an abbreviated form of the month name, then it is used—for example, `Jan`, `Feb`, and so on. If **N**ormal is specified (or allowed to default) for *date_format2*, the *input_date* must have the month (*mon*) specified in the English abbreviated form of the month name in mixed case.

Ordered

date in the format: *yy/mm/dd* (suitable for sorting, and so forth).

Standard

date in the format: *yyyymmdd* (suitable for sorting, and so forth).

Usa

date in the format: *mm/dd/yy*.

Weekday

the English name for the day of the week, in mixed case—for example, `Tuesday`. Only valid for *date_format1*.

Here are some examples, assuming today is `November 20, 2001`:

```
DATE()                      ->    '20 Nov 2001'
DATE(,'20020609','S')       ->    '9 Jun 2002'
DATE('B')                   ->    '730808'
DATE('B','25 Sep 2001')     ->    '730752'
DATE('C')                   ->    '690'
DATE('E')                   ->    '20/11/01'
DATE('J')                   ->    '01324'
DATE('M')                   ->    'November'
DATE('N')                   ->    '20 Nov 2001'
DATE('N','1438','C')        ->    '8 Dec 2003'
DATE('O')                   ->    '01/11/20'
DATE('S')                    ->    '20011120'
DATE('U')                    ->    '11/20/01'
DATE('U','25 May 2001')      ->    '05/25/01'
DATE('U','25 MAY 2001')      ->    ERROR,month not in mixed case
DATE('W')                    ->    'Tuesday'
```

Note:

1. The first call to DATE or TIME in one clause causes a time stamp to be made that is then used for *all* calls to these functions in that clause. Therefore, multiple calls to any of the DATE or TIME functions or both in a single expression or clause are guaranteed to be consistent with each other.
2. Input dates given in 2-digit year formats (i.e. **E**uropean, **J**ulian, **O**rdered, **U**sa) are interpreted as being within a 100 year window as calculated by:

   (current_year - 50) = low end of window
   (current_year + 49) = high end of window

DATE conversion requires that the input_date conforms exactly to one of the syntax forms that could have been output by the DATE function. For example, the following invocations of DATE would fail. The incorrect specification of *input_date* in each case would result in the same error message, IRX0040I.

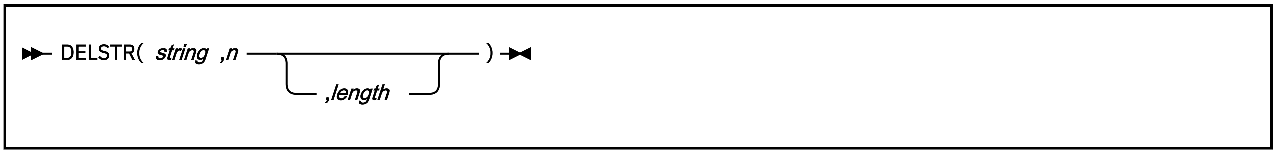
```
DATE('B','7 MAY 2001')   ->  IRX0040I  (input is not mixed case)
DATE('B',' 7 May 2001')  ->  IRX0040I  (input has a leading blank)
DATE('B','07 May 2001')  ->  IRX0040I  (input has a leading zero)
```

DBCS (double-byte character set functions)

The following are all part of DBCS processing functions. See Appendix A, “Double-byte character set (DBCS) support,” on page 429.

```
DBADJUST                    DBRIGHT                    DBUNBRACKET
DBBRACKET                   DBRLEFT                    DBVALIDATE
DBCENTER                    DBRRIGHT                   DBWIDTH
DBCJUSTIFY                  DBTODBCS
DBLEFT                      DBTOSBCS
```

DELSTR (Delete String)

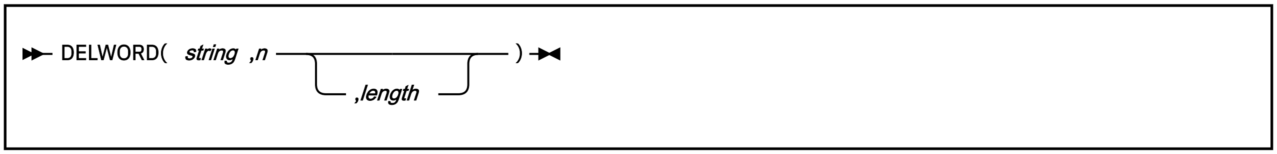


returns *string* after deleting the substring that begins at the *n*th character and is of *length* characters. If you omit *length*, or if *length* is greater than the number of characters from *n* to the end of *string*, the function deletes the rest of *string* (including the *n*th character). The *length* must be a positive whole number or zero. The *n* must be a positive whole number. If *n* is greater than the length of *string*, the function returns *string* unchanged.

Here are some examples:

```
DELSTR('abcd',3)       ->    'ab'
DELSTR('abcde',3,2)    ->    'abe'
DELSTR('abcde',6)      ->    'abcde'
```

DELWORD (Delete Word)

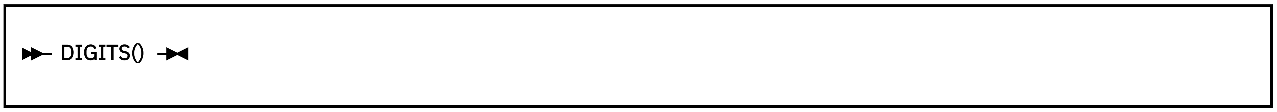


returns *string* after deleting the substring that starts at the *n*th word and is of *length* blank-delimited words. If you omit *length*, or if *length* is greater than the number of words from *n* to the end of *string*, the function deletes the remaining words in *string* (including the *n*th word). The *length* must be a positive whole number or zero. The *n* must be a positive whole number. If *n* is greater than the number of words in *string*, the function returns *string* unchanged. The string deleted includes any blanks following the final word involved but none of the blanks preceding the first word involved.

Here are some examples:

```
DELWORD('Now is the  time',2,2)  ->  'Now time'
DELWORD('Now is the time ',3)    ->  'Now is '
DELWORD('Now is the  time',5)    ->  'Now is the  time'
DELWORD('Now is   the time',3,1) ->  'Now is   time'
```

DIGITS

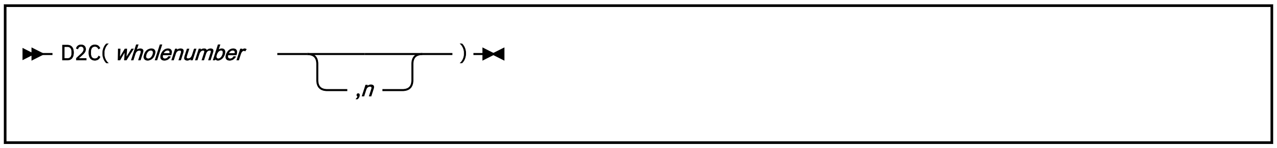


returns the current setting of NUMERIC DIGITS. See “NUMERIC” on page 55 for more information.

Here is an example:

```
DIGITS()    ->    9      /* by default */
```

D2C (Decimal to Character)



returns a string, in character format, that represents *wholenumber*, a decimal number, converted to binary. If you specify *n*, it is the length of the final result in characters; after conversion, the input string is sign-extended to the required length. If the number is too big to fit into *n* characters, then the result is truncated on the left. The *n* must be a positive whole number or zero.

If you omit *n*, *wholenumber* must be a positive whole number or zero, and the result length is as needed. Therefore, the returned result has no leading '00'x characters.

Here are some examples:

```
D2C(9)         ->   ' '    /* '09'x is unprintable in EBCDIC       */
D2C(129)       ->   'a'    /* '81'x is an EBCDIC 'a'               */
D2C(129,1)     ->   'a'    /* '81'x is an EBCDIC 'a'               */
D2C(129,2)     ->   ' a'   /* '0081'x is EBCDIC ' a'               */
D2C(257,1)     ->   ' '    /* '01'x is unprintable in EBCDIC       */
D2C(-127,1)    ->   'a'    /* '81'x is EBCDIC 'a'                  */
D2C(-127,2)    ->   ' a'   /* 'FF'x is unprintable EBCDIC;         */
                           /* '81'x is EBCDIC 'a'                  */
D2C(-1,4)      ->   '    ' /* 'FFFFFFFF'x is unprintable in EBCDIC */
D2C(12,0)      ->   ''     /* '' is a null string                  */
```

Implementation maximum: The output string may not have more than 250 significant characters, though a longer result is possible if it has additional leading sign characters ('00'x and 'FF'x).

D2X (Decimal to Hexadecimal)

```
►► D2X( wholenumber ─┬──────┬─ ) ►◄
                     └─ ,n ─┘
```

returns a string, in character format, that represents *wholenumber*, a decimal number, converted to hexadecimal. The returned string uses uppercase alphabetics for the values A–F and does not include blanks.

If you specify *n*, it is the length of the final result in characters; after conversion the input string is sign-extended to the required length. If the number is too big to fit into *n* characters, it is truncated on the left. The *n* must be a positive whole number or zero.

If you omit *n*, *wholenumber* must be a positive whole number or zero, and the returned result has no leading zeros.

Here are some examples:

```
D2X(9)          ->    '9'
D2X(129)        ->    '81'
D2X(129,1)      ->    '1'
D2X(129,2)      ->    '81'
D2X(129,4)      ->    '0081'
D2X(257,2)      ->    '01'
D2X(-127,2)     ->    '81'
D2X(-127,4)     ->    'FF81'
D2X(12,0)       ->    ''
```

Implementation maximum: The output string may not have more than 500 significant hexadecimal characters, though a longer result is possible if it has additional leading sign characters (0 and F).

ERRORTEXT

```
►► ERRORTEXT( n) ►◄
```

returns the REXX error message associated with error number *n*. The *n* must be in the range 0–99, and any other value is an error. Returns the null string if *n* is in the allowed range but is not a defined REXX error number.

Error numbers produced by syntax errors during processing of REXX execs correspond to TSO/E REXX messages (described in *z/OS TSO/E Messages*.) For example, error 26 corresponds to message number IRX0026I. The error number is also the value that is placed in the REXX special variable RC when SIGNAL ON SYNTAX event is trapped.

Here are some examples:

```
ERRORTEXT(16)    ->    'Label not found'
ERRORTEXT(60)    ->    ''
```

EXTERNALS

(Non-SAA Function)

EXTERNALS is a non-SAA built-in function provided only by TSO/E and VM.

```
►► EXTERNALS() ►◄
```

always returns a 0. For example:

```
EXTERNALS()    ->    0    /* Always */
```

The EXTERNALS function returns the number of elements in the terminal input buffer (system external event queue). In TSO/E, there is no equivalent buffer. Therefore, in the TSO/E implementation of REXX, the EXTERNALS function always returns a 0.

FIND

(Non-SAA Function)

FIND is a non-SAA built-in function provided only by TSO/E and VM.

WORDPOS is the preferred built-in function for this type of word search. See "WORDPOS (Word Position)" on page 106 for a complete description.

►►─ FIND(*string* ,*phrase*) ─►◄

returns the word number of the first word of *phrase* found in *string* or returns `0` if *phrase* is not found or if there are no words in *phrase*. The *phrase* is a sequence of blank-delimited words. Multiple blanks between words in *phrase* or *string* are treated as a single blank for the comparison.

Here are some examples:

```
FIND('now is the time','is the time')    ->    2
FIND('now is  the time','is    the')     ->    2
FIND('now is  the time','is  time ')     ->    0
```

FORM

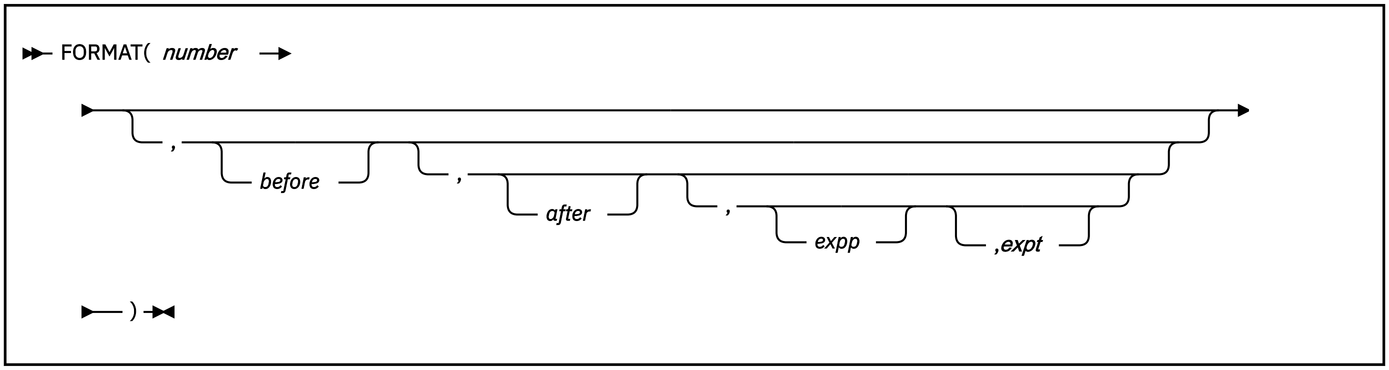
►►─ FORM() ─►◄

returns the current setting of NUMERIC FORM. See "NUMERIC" on page 55 for more information.

Here is an example:

```
FORM()    ->    'SCIENTIFIC'   /* by default */
```

FORMAT



returns *number*, rounded and formatted.

The *number* is first rounded according to standard REXX rules, just as though the operation `number+0` had been carried out. The result is precisely that of this operation if you specify only *number*. If you specify any other options, the *number* is formatted as follows.

The *before* and *after* options describe how many characters are used for the integer and decimal parts of the result, respectively. If you omit either or both of these, the number of characters used for that part is as needed.

If *before* is not large enough to contain the integer part of the number (plus the sign for a negative number), an error results. If *before* is larger than needed for that part, the number is padded on the left with blanks. If *after* is not the same size as the decimal part of the number, the number is rounded (or extended with zeros) to fit. Specifying `0` causes the number to be rounded to an integer.

Here are some examples:

```
FORMAT('3',4)            ->    '   3'
FORMAT('1.73',4,0)       ->    '   2'
FORMAT('1.73',4,3)       ->    '   1.730'
FORMAT('-.76',4,1)       ->    '  -0.8'
FORMAT('3.03',4)         ->    '   3.03'
FORMAT(' - 12.73',,4)    ->    '-12.7300'
FORMAT(' - 12.73')       ->    '-12.73'
FORMAT('0.000')          ->    '0'
```

The first three arguments are as described previously. In addition, *expp* and *expt* control the exponent part of the result, which, by default, is formatted according to the current NUMERIC settings of DIGITS and FORM. The *expp* sets the number of places for the exponent part; the default is to use as many as needed (which may be zero). The *expt* sets the trigger point for use of exponential notation. The default is the current setting of NUMERIC DIGITS.

If *expp* is `0`, no exponent is supplied, and the number is expressed in *simple* form with added zeros as necessary. If *expp* is not large enough to contain the exponent, an error results.

If the number of places needed for the integer or decimal part exceeds *expt* or twice *expt*, respectively, exponential notation is used. If *expt* is `0`, exponential notation is always used unless the exponent would be `0`. (If *expp* is `0`, this overrides a `0` value of *expt*.) If the exponent would be `0` when a nonzero *expp* is specified, then *expp*+2 blanks are supplied for the exponent part of the result. If the exponent would be `0` and *expp* is not specified, simple form is used.

Here are some examples:

```
FORMAT('12345.73',,,2,2)    ->    '1.234573E+04'
FORMAT('12345.73',,3,,0)    ->    '1.235E+4'
FORMAT('1.234573',,3,,0)    ->    '1.235'
FORMAT('12345.73',,,3,6)    ->    '12345.73'
FORMAT('1234567e5',,3,0)    ->    '123456700000.000'
```

FUZZ

```
▸▸— FUZZ() —▸◂
```

returns the current setting of NUMERIC FUZZ. See “NUMERIC” on page 55 for more information.

Here is an example:

```
FUZZ()    ->    0     /* by default */
```

GETMSG

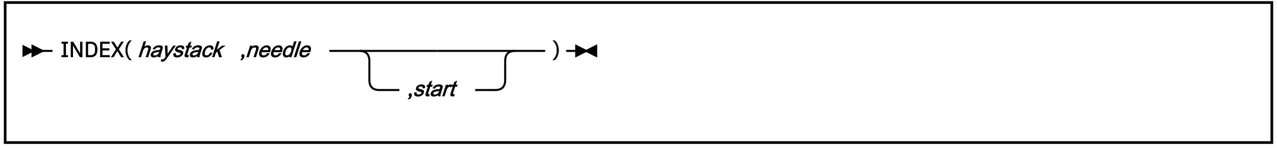
GETMSG is a TSO/E external function. See “GETMSG” on page 110.

INDEX

(Non-SAA Function)

INDEX is a non-SAA built-in function provided only by TSO/E and VM.

POS is the preferred built-in function for obtaining the position of one string in another. See "POS (Position)" on page 96 for a complete description.

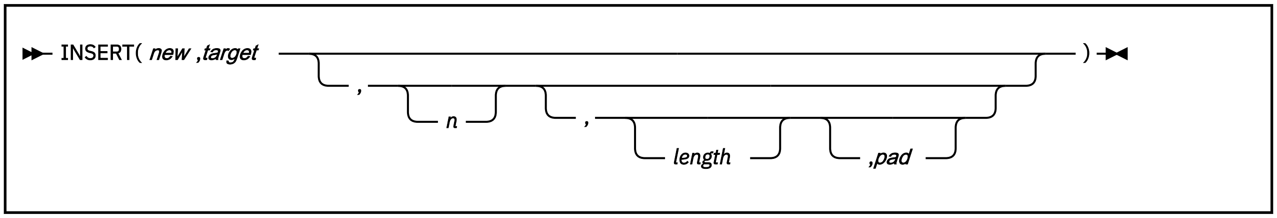


returns the character position of one string, *needle*, in another, *haystack*, or returns 0 if the string *needle* is not found or is a null string. By default the search starts at the first character of *haystack* (*start* has the value 1). You can override this by specifying a different *start* point, which must be a positive whole number.

Here are some examples:

```
INDEX('abcdef','cd')      ->    3
INDEX('abcdef','xd')      ->    0
INDEX('abcdef','bc',3)    ->    0
INDEX('abcabc','bc',3)    ->    5
INDEX('abcabc','bc',6)    ->    0
```

INSERT



inserts the string *new*, padded or truncated to length *length*, into the string *target* after the *n*th character. The default value for *n* is 0, which means insert before the beginning of the string. If specified, *n* and *length* must be positive whole numbers or zero. If *n* is greater than the length of the target string, padding is added before the string *new* also. The default value for *length* is the length of *new*. If *length* is less than the length of the string *new*, then INSERT truncates *new* to length *length*. The default *pad* character is a blank.

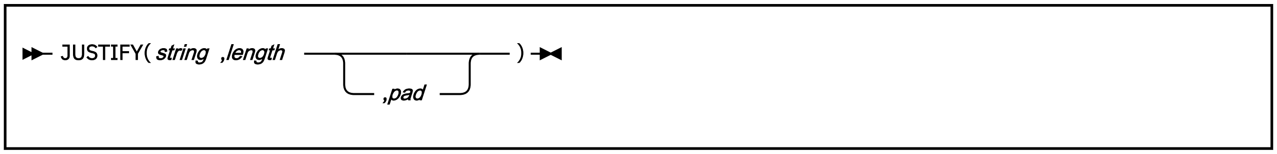
Here are some examples:

```
INSERT(' ','abcdef',3)          ->    'abc def'
INSERT('123','abc',5,6)         ->    'abc  123   '
INSERT('123','abc',5,6,'+')     ->    'abc++123+++'
INSERT('123','abc')             ->    '123abc'
INSERT('123','abc',,5,'-')      ->    '123--abc'
```

JUSTIFY

(Non-SAA Function)

JUSTIFY is a non-SAA built-in function provided only by TSO/E and VM.



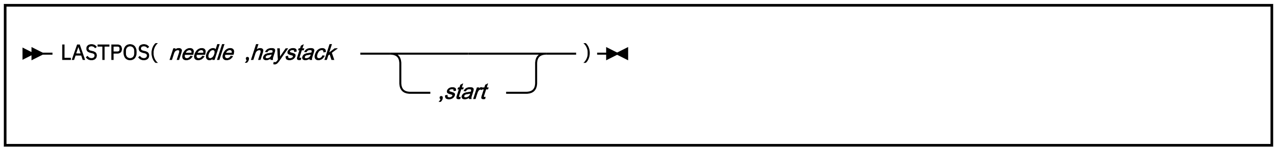
returns *string* formatted by adding *pad* characters between blank-delimited words to justify to both margins. This is done to width *length* (*length* must be a positive whole number or zero). The default *pad* character is a blank.

The first step is to remove extra blanks as though `SPACE(string)` had been run (that is, multiple blanks are converted to single blanks, and leading and trailing blanks are removed). If *length* is less than the width of the changed string, the string is then truncated on the right and any trailing blank is removed. Extra *pad* characters are then added evenly from left to right to provide the required length, and the *pad* character replaces the blanks between words.

Here are some examples:

```
JUSTIFY('The blue sky',14)       ->    'The  blue  sky'
JUSTIFY('The blue sky',8)        ->    'The blue'
JUSTIFY('The blue sky',9)        ->    'The  blue'
JUSTIFY('The blue sky',9,'+')    ->    'The++blue'
```

LASTPOS (Last Position)

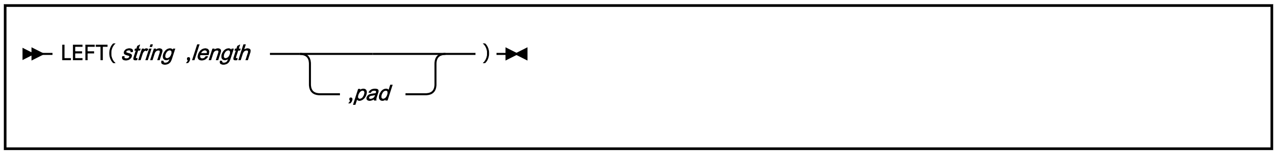


returns the position of the last occurrence of one string, *needle*, in another, *haystack*. (See also the POS function.) Returns `0` if *needle* is the null string or is not found. By default the search starts at the last character of *haystack* and scans backward. You can override this by specifying *start*, the point at which the backward scan starts. *start* must be a positive whole number and defaults to `LENGTH(haystack)` if larger than that value or omitted.

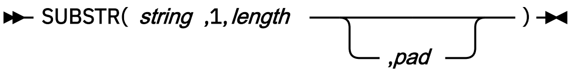
Here are some examples:

```
LASTPOS(' ','abc def ghi')      ->    8
LASTPOS(' ','abcdefghi')        ->    0
LASTPOS('xy','efgxyz')          ->    4
LASTPOS(' ','abc def ghi',7)    ->    4
```

LEFT



returns a string of length *length*, containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank. *length* must be a positive whole number or zero. The LEFT function is exactly equivalent to:



Here are some examples:

```
LEFT('abc d',8)        ->    'abc d   '
LEFT('abc d',8,'.')    ->    'abc d...'
LEFT('abc  def',7)     ->    'abc  de'
```

LENGTH

```
►► LENGTH( string ) ►◄
```

returns the length of *string*.

Here are some examples:

```
LENGTH('abcdefgh')    ->    8
LENGTH('abc defg')    ->    8
LENGTH('')            ->    0
```

LINESIZE

(Non-SAA Function)

LINESIZE is a non-SAA built-in function provided only by TSO/E and VM.

```
►► LINESIZE() ►◄
```

returns the current terminal line width minus 1 (the point at which the language processor breaks lines displayed using the SAY instruction).

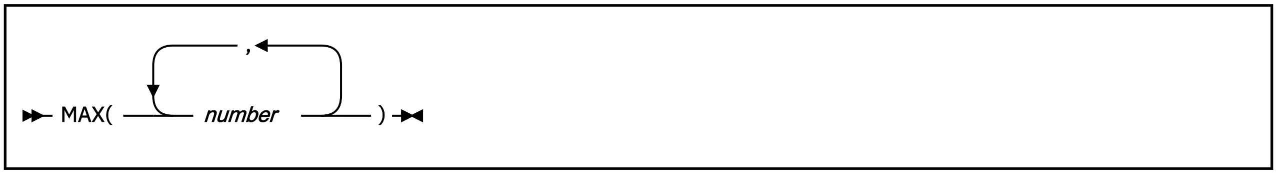
If the REXX exec is running in TSO/E background (that is, on the JCL EXEC statement PGM=IKJEFT01), LINESIZE always returns the value 131. If the REXX exec is running in TSO/E foreground, the LINESIZE function always returns the current terminal width (as defined by the TSO/E TERMINAL command) minus one character.

If the exec is running in a non-TSO/E address space, LINESIZE returns the logical record length of the OUTDD file (the default file is SYSTSPRT). The OUTDD file is specified in the module name table (see “Module name table” on page 326).

LISTDSI

LISTDSI is a TSO/E external function. See “LISTDSI” on page 115.

MAX (Maximum)



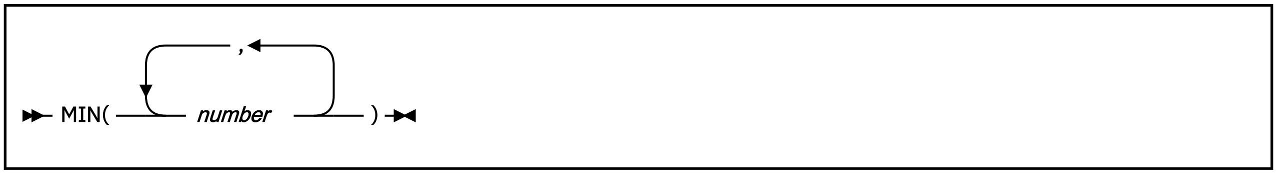
returns the largest number from the list specified, formatted according to the current NUMERIC settings.

Here are some examples:

```
MAX(12,6,7,9)                                                     ->    12
MAX(17.3,19,17.03)                                                ->    19
MAX(-7,-3,-4.3)                                                   ->    -3
MAX(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,MAX(20,21))   ->    21
```

Implementation maximum: You can specify up to 20 *number*s, and can nest calls to MAX if more arguments are needed.

MIN (Minimum)



returns the smallest number from the list specified, formatted according to the current NUMERIC settings.

Here are some examples:

```
MIN(12,6,7,9)                                                     ->   6
MIN(17.3,19,17.03)                                                ->  17.03
MIN(-7,-3,-4.3)                                                   ->  -7
MIN(21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,MIN(2,1))   ->  1
```

Implementation maximum: You can specify up to 20 *number*s, and can nest calls to MIN if more arguments are needed.

MSG

MSG is a TSO/E external function. See "MSG" on page 127.

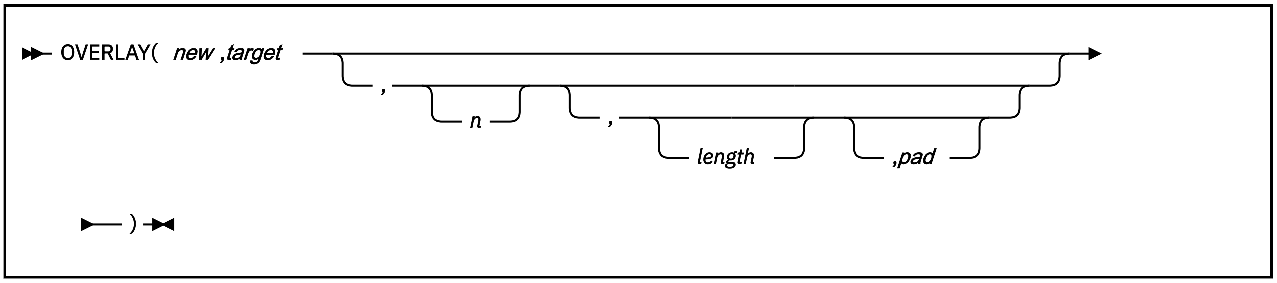
MVSVAR

MVSVAR is a TSO/E external function. See "MVSVAR" on page 128.

OUTTRAP

OUTTRAP is a TSO/E external function. See "OUTTRAP" on page 133.

OVERLAY

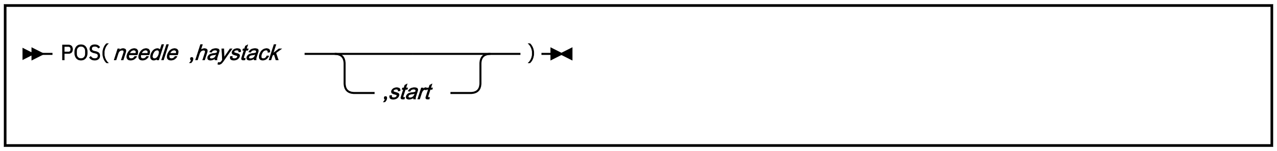


returns the string *target*, which, starting at the *n*th character, is overlaid with the string *new*, padded or truncated to length *length*. (The overlay may extend beyond the end of the original *target* string.) If you specify *length*, it must be a positive whole number or zero. The default value for *length* is the length of *new*. If *n* is greater than the length of the target string, padding is added before the *new* string. The default *pad* character is a blank, and the default value for *n* is 1. If you specify *n*, it must be a positive whole number.

Here are some examples:

```
OVERLAY(' ','abcdef',3)          ->    'ab def'
OVERLAY('.','abcdef',3,2)        ->    'ab. ef'
OVERLAY('qq','abcd')             ->    'qqcd'
OVERLAY('qq','abcd',4)           ->    'abcqq'
OVERLAY('123','abc',5,6,'+')     ->    'abc+123+++'
```

POS (Position)



returns the position of one string, *needle*, in another, *haystack*. (See also the INDEX and LASTPOS functions.) Returns 0 if *needle* is the null string or is not found or if *start* is greater than the length of *haystack*. By default the search starts at the first character of *haystack* (that is, the value of *start* is 1). You can override this by specifying *start* (which must be a positive whole number), the point at which the search starts.

Here are some examples:

```
POS('day','Saturday')       ->    6
POS('x','abc def ghi')      ->    0
POS(' ','abc def ghi')      ->    4
POS(' ','abc def ghi',5)    ->    8
```

PROMPT

PROMPT is a TSO/E external function. See "PROMPT" on page 139.

QUEUED

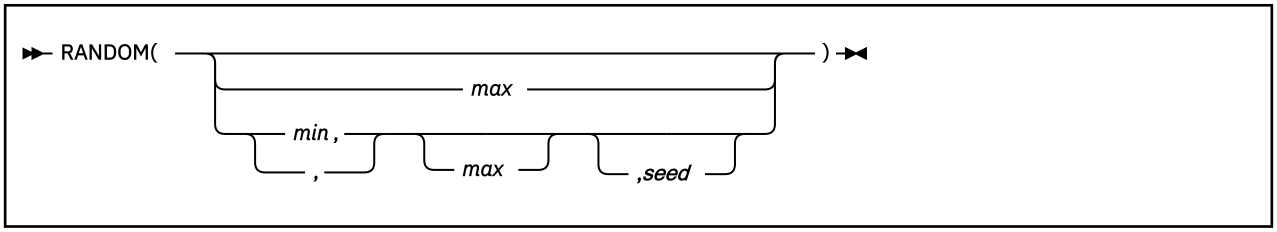
returns the number of lines remaining in the external data queue when the function is called.

The TSO/E implementation of the external data queue is the data stack.

Here is an example:

```
QUEUED()    ->    5    /* Perhaps */
```

RANDOM



returns a quasi-random nonnegative whole number in the range *min* to *max* inclusive. If you specify *max* or *min* or both, *max* minus *min* cannot exceed 100000. The *min* and *max* default to 0 and 999, respectively. To start a repeatable sequence of results, use a specific *seed* as the third argument, as described in Note "1" on page 97. This *seed* must be a positive whole number ranging from 0 to 999999999.

Here are some examples:

```
RANDOM()          ->    305
RANDOM(5,8)       ->      7
RANDOM(2)         ->      0  /*  0  to  2    */
RANDOM(,,1983)    ->    123  /* reproducible */
```

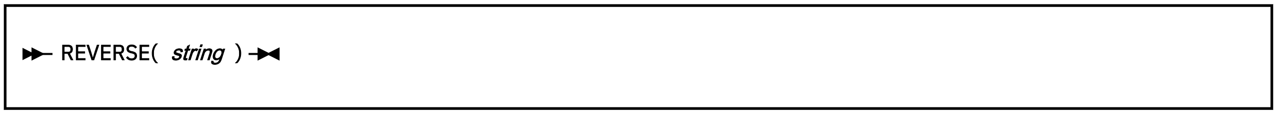
Note:

1. To obtain a predictable sequence of quasi-random numbers, use RANDOM a number of times, but specify a *seed* only the first time. For example, to simulate 40 throws of a 6-sided, unbiased die:

   ```
   sequence = RANDOM(1,6,12345)  /* any number would */
                                 /* do for a seed    */
   do 39
      sequence = sequence RANDOM(1,6)
      end
   say sequence
   ```

   The numbers are generated mathematically, using the initial *seed*, so that as far as possible they appear to be random. Running the program again produces the same sequence; using a different initial *seed* almost certainly produces a different sequence. If you do not supply a *seed*, the first time RANDOM is called, an arbitrary seed is used. Hence, your program usually gives different results each time it is run.
2. The random number generator is global for an entire program; the current seed is not saved across internal routine calls.

REVERSE

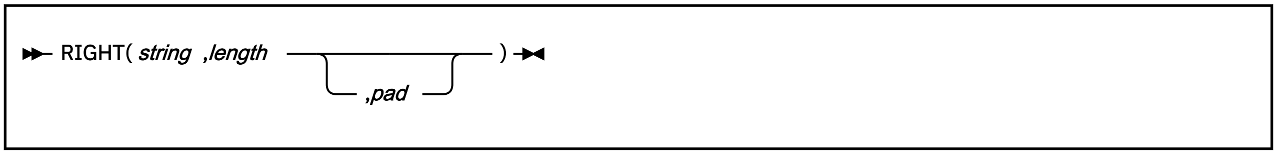


returns *string*, swapped end for end.

Here are some examples:

```
REVERSE('ABc.')    ->    '.cBA'
REVERSE('XYZ ')    ->    ' ZYX'
```

RIGHT



returns a string of length *length* containing the rightmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank. The *length* must be a positive whole number or zero.

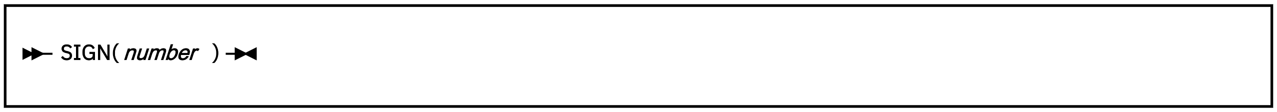
Here are some examples:

```
RIGHT('abc  d',8)     ->    '  abc  d'
RIGHT('abc def',5)    ->    'c def'
RIGHT('12',5,'0')     ->    '00012'
```

SETLANG

SETLANG is a TSO/E external function. See “SETLANG” on page 142.

SIGN

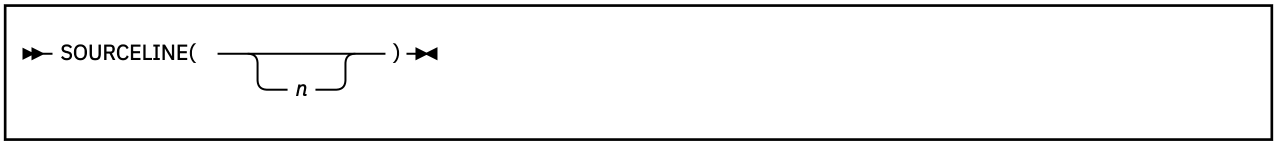


returns a number that indicates the sign of *number*. The *number* is first rounded according to standard REXX rules, just as though the operation `number+0` had been carried out. Returns `-1` if *number* is less than `0`; returns `0` if it is `0`; and returns `1` if it is greater than `0`.

Here are some examples:

```
SIGN('12.3')       ->     1
SIGN(' -0.307')    ->    -1
SIGN(0.0)          ->     0
```

SOURCELINE

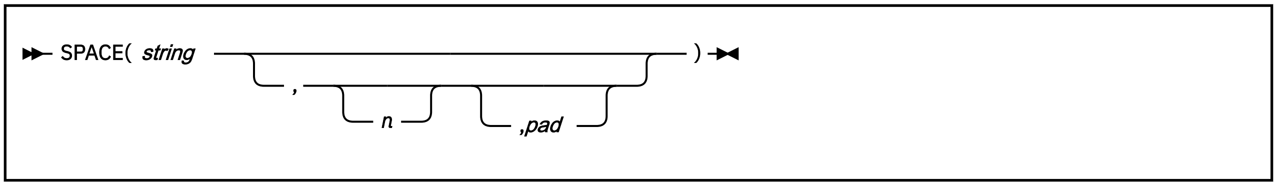


returns the line number of the final line in the program if you omit *n* or `0` if the implementation does not allow access to the source lines. If you specify *n*, returns the *n*th line in the program if available at the time of execution; otherwise, returns the null string. If specified, *n* must be a positive whole number and must not exceed the number that a call to SOURCELINE with no arguments returns.

Here are some examples:

```
SOURCELINE()    ->   10
SOURCELINE(1)   ->   '/* This is a 10-line REXX program */'
```

SPACE



returns the blank-delimited words in *string* with *n* *pad* characters between each word. If you specify *n*, it must be a positive whole number or zero. If it is `0`, all blanks are removed. Leading and trailing blanks are always removed. The default for *n* is `1`, and the default *pad* character is a blank.

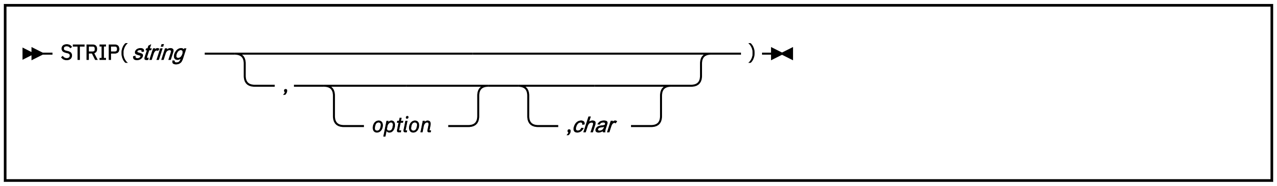
Here are some examples:

```
SPACE('abc  def  ')          ->    'abc def'
SPACE('  abc def',3)         ->    'abc   def'
SPACE('abc  def  ',1)        ->    'abc def'
SPACE('abc  def  ',0)        ->    'abcdef'
SPACE('abc  def  ',2,'+')    ->    'abc++def'
```

STORAGE

STORAGE is a TSO/E external function. See “STORAGE” on page 144.

STRIP



returns *string* with leading or trailing characters or both removed, based on the *option* you specify. The following are valid *option*s. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Both
removes both leading and trailing characters from *string*. This is the default.

Leading
removes leading characters from *string*.

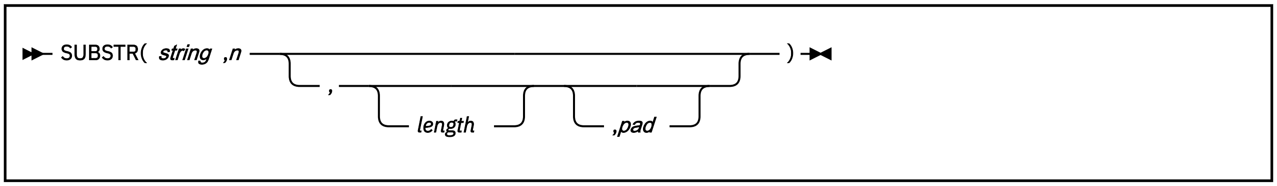
Trailing
removes trailing characters from *string*.

The third argument, *char*, specifies the character to be removed, and the default is a blank. If you specify *char*, it must be exactly one character long.

Here are some examples:

```
STRIP('  ab c  ')        ->    'ab c'
STRIP('  ab c  ','L')    ->    'ab c  '
STRIP('  ab c  ','t')    ->    '  ab c'
STRIP('12.7000',,0)      ->    '12.7'
STRIP('0012.700',,0)     ->    '12.7'
```

SUBSTR (Substring)



returns the substring of *string* that begins at the *n*th character and is of length *length*, padded with *pad* if necessary. The *n* must be a positive whole number. If *n* is greater than `LENGTH(string)`, then only pad characters are returned.

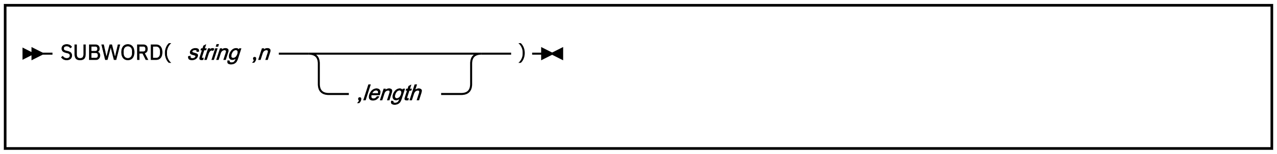
If you omit *length*, the rest of the string is returned. The default *pad* character is a blank.

Here are some examples:

```
SUBSTR('abc',2)          ->    'bc'
SUBSTR('abc',2,4)        ->    'bc  '
SUBSTR('abc',2,6,'.')    ->    'bc....'
```

Note: In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting substrings, especially if more than one substring is to be extracted from a string. See also the LEFT and RIGHT functions.

SUBWORD

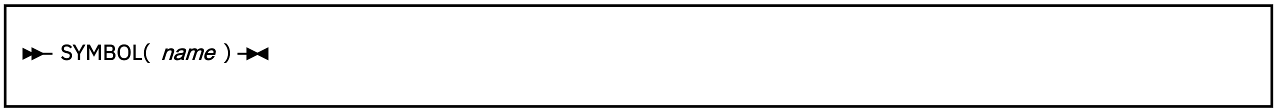


returns the substring of *string* that starts at the *n*th word, and is up to *length* blank-delimited words. The *n* must be a positive whole number. If you omit *length*, it defaults to the number of remaining words in *string*. The returned string never has leading or trailing blanks, but includes all blanks between the selected words.

Here are some examples:

```
SUBWORD('Now is the  time',2,2)    ->    'is the'
SUBWORD('Now is the  time',3)      ->    'the  time'
SUBWORD('Now is the  time',5)      ->    ''
```

SYMBOL



returns the state of the symbol named by *name*. Returns BAD if *name* is not a valid REXX symbol. Returns VAR if it is the name of a variable (that is, a symbol that has been assigned a value). Otherwise returns LIT, indicating that it is either a constant symbol or a symbol that has not yet been assigned a value (that is, a literal).

As with symbols in REXX expressions, lowercase characters in *name* are translated to uppercase and substitution in a compound name occurs if possible.

You should specify *name* as a literal string (or it should be derived from an expression) to prevent substitution before it is passed to the function.

Here are some examples:

```
/* following: Drop A.3;  J=3 */
SYMBOL('J')      ->   'VAR'
SYMBOL(J)        ->   'LIT' /* has tested "3"     */
SYMBOL('a.j')    ->   'LIT' /* has tested A.3     */
SYMBOL(2)        ->   'LIT' /* a constant symbol  */
SYMBOL('*')      ->   'BAD' /* not a valid symbol */
```

SYSCPUS

SYSCPUS is a TSO/E external function. See "SYSCPUS" on page 146.

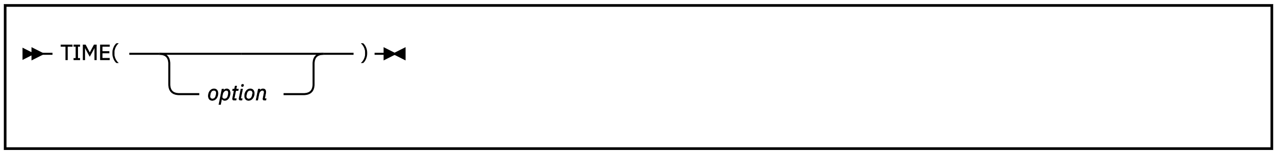
SYSDSN

SYSDSN is a TSO/E external function. See page "SYSDSN" on page 147.

SYSVAR

SYSVAR is a TSO/E external function. See "#unique_236/unique_236_Connect_42_tsosvar" on page 148.

TIME



returns the local time in the 24-hour clock format: hh:mm:ss (hours, minutes, and seconds) by default, for example, `04:41:37`.

You can use the following *option*s to obtain alternative formats, or to gain access to the elapsed-time clock. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Civil
returns the time in Civil format: hh:mmxx. The hours may take the values `1` through `12`, and the minutes the values `00` through `59`. The minutes are followed immediately by the letters `am` or `pm`. This distinguishes times in the morning (12 midnight through 11:59 a.m.—appearing as `12:00am` through `11:59am`) from noon and afternoon (12 noon through 11:59 p.m.—appearing as `12:00pm` through `11:59pm`). The hour has no leading zero. The minute field shows the current minute (rather than the nearest minute) for consistency with other TIME results.

Elapsed
returns sssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock (described later) was started or reset. The number has no leading zeros or blanks, and the setting of NUMERIC DIGITS does not affect the number. The fractional part always has six digits.

Hours
returns up to two characters giving the number of hours since midnight in the format: hh (no leading zeros or blanks, except for a result of `0`).

Long
returns time in the format: hh:mm:ss.uuuuuu (uuuuuu is the fraction of seconds, in microseconds). The first eight characters of the result follow the same rules as for the Normal form, and the fractional part is always six digits.

Minutes
returns up to four characters giving the number of minutes since midnight in the format: mmmm (no leading zeros or blanks, except for a result of `0`).

Normal
returns the time in the default format hh:mm:ss, as described previously. The hours can have the values `00` through `23`, and minutes and seconds, `00` through `59`. All these are always two digits. Any fractions of seconds are ignored (times are never rounded up). **This is the default**.

Reset
returns sssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock (described later) was started or reset and also resets the elapsed-time clock to zero. The number has no leading zeros or blanks, and the setting of NUMERIC DIGITS does not affect the number. The fractional part always has six digits.

Seconds
returns up to five characters giving the number of seconds since midnight in the format: sssss (no leading zeros or blanks, except for a result of `0`).

Here are some examples, assuming that the time is 4:54 p.m.:

```
TIME()       ->   '16:54:22'
TIME('C')    ->   '4:54pm'
TIME('H')    ->   '16'
TIME('L')    ->   '16:54:22.123456'   /* Perhaps */
TIME('M')    ->   '1014'           /* 54 + 60*16 */
TIME('N')    ->   '16:54:22'
TIME('S')    ->   '60862'  /* 22 + 60*(54+60*16) */
```

The elapsed-time clock: You can use the TIME function to measure real (elapsed) time intervals. On the first call in a program to `TIME('E')` or `TIME('R')`, the elapsed-time clock is started, and either call returns `0`. From then on, calls to `TIME('E')` and to `TIME('R')` return the elapsed time since that first call or since the last call to `TIME('R')`.

The clock is saved across internal routine calls, which is to say that an internal routine inherits the time clock its caller started. Any timing the caller is doing is not affected, even if an internal routine resets the clock. An example of the elapsed-time clock:

```
time('E')    ->    0          /* The first call */
/* pause of one second here */
time('E')    ->    1.002345   /* or thereabouts */
/* pause of one second here */
time('R')    ->    2.004690   /* or thereabouts */
/* pause of one second here */
time('R')    ->    1.002345   /* or thereabouts */
```

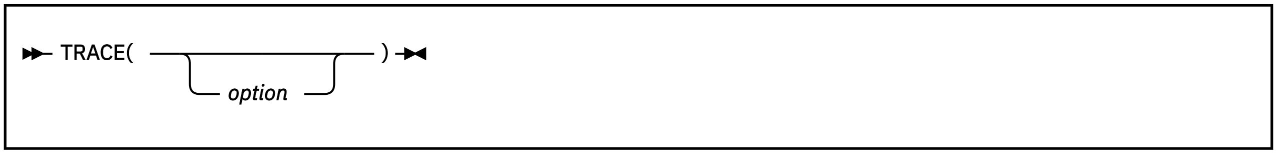
Restriction: See the note under DATE about consistency of times within a single clause. The elapsed-time clock is synchronized to the other calls to TIME and DATE, so multiple calls to the elapsed-time clock in a single clause always return the same result. For the same reason, the interval between two usual TIME/DATE results may be calculated exactly using the elapsed-time clock.

Note: Starting with z/OS V2R1, the REXX elapsed timer can tolerate a change of the local time while the elapsed timer is running.

For example, if the local time is adjusted ahead (as with a change from Standard Time to Daylight Savings Time), or backwards (as with a fall back from Daylight Savings Time to Standard Time) while a REXX program is running with an active elapsed timer, the REXX program will see no disruption to the elapsed timer and will continue to run normally, because internally the elapsed timer start time is adjusted, relative to the new local time, to produce a resultant elapsed local time that is within approximately +/- 2 seconds of the actual elapsed UTC timer. If the local time is not changed across an elapsed timer interval, the elapsed timer remains precise down to the nearest microsecond.

Implementation maximum: If the number of seconds in the elapsed time exceeds nine digits (equivalent to over 31.6 years), an error results.

TRACE



returns trace actions currently in effect and, optionally, alters the setting.

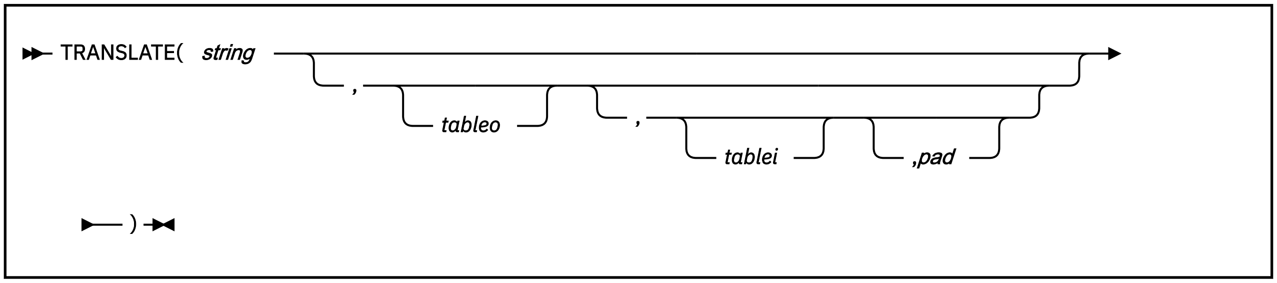
If you specify *option*, it selects the trace setting. It must be one of the valid prefixes `?` or `!` or one of the alphabetic character options associated with the TRACE instruction (that is, starting with A, C, E, F, I, L, N, O, R, or S) or both. (See "Alphabetic character (word) options" on page 68 for full details.)

Unlike the TRACE instruction, the TRACE function alters the trace action even if interactive debug is active. Also unlike the TRACE instruction, *option* cannot be a number.

Here are some examples:

```
TRACE()       ->   '?R' /* maybe */
TRACE('O')    ->   '?R' /* also sets tracing off    */
TRACE('?I')   ->   'O'  /* now in interactive debug */
```

TRANSLATE



returns *string* with each character translated to another character or unchanged. You can also use this function to reorder the characters in *string*.

The output table is *tableo* and the input translation table is *tablei*. TRANSLATE searches *tablei* for each character in *string*. If the character is found, then the corresponding character in *tableo* is used in the result string; if there are duplicates in *tablei*, the first (leftmost) occurrence is used. If the character is not found, the original character in *string* is used. The result string is always the same length as *string*.

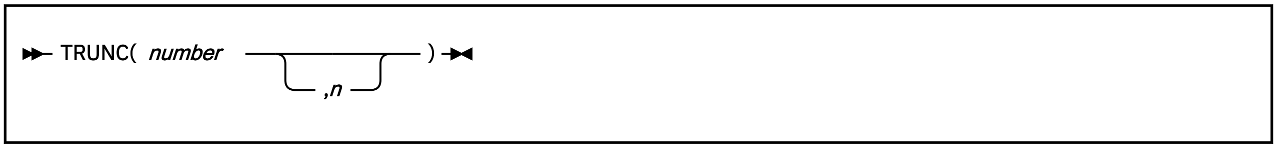
The tables can be of any length. If you specify neither translation table and omit *pad*, *string* is simply translated to uppercase (that is, lowercase a–z to uppercase A–Z), but, if you include *pad*, the language processor translates the entire string to *pad* characters. *tablei* defaults to XRANGE('00'x,'FF'x), and *tableo* defaults to the null string and is padded with *pad* or truncated as necessary. The default *pad* is a blank.

Here are some examples:

```
TRANSLATE('abcdef')                     ->    'ABCDEF'
TRANSLATE('abbc','&','b')               ->    'a&&c'
TRANSLATE('abcdef','12','ec')           ->    'ab2d1f'
TRANSLATE('abcdef','12','abcd','.')     ->    '12..ef'
TRANSLATE('APQRV',,'PR')                ->    'A Q V'
TRANSLATE('APQRV',XRANGE('00'X,'Q'))    ->    'APQ  '
TRANSLATE('4123','abcd','1234')         ->    'dabc'
```

The last example shows how to use the TRANSLATE function to reorder the characters in a string. In the example, the last character of any four-character string specified as the second argument would be moved to the beginning of the string.

TRUNC (Truncate)



returns the integer part of *number* and *n* decimal places. The default *n* is 0 and returns an integer with no decimal point. If you specify *n*, it must be a positive whole number or zero. The *number* is first rounded according to standard REXX rules, just as though the operation number+0 had been carried out. The number is then truncated to *n* decimal places (or trailing zeros are added if needed to make up the specified length). The result is never in exponential form.

Here are some examples:

```
TRUNC(12.3)          ->    12
TRUNC(127.09782,3)   ->    127.097
TRUNC(127.1,3)       ->    127.100
TRUNC(127,2)         ->    127.00
```

The *number* is rounded according to the current setting of NUMERIC DIGITS if necessary before the function processes it.

USERID

(Non-SAA Function)

USERID is a non-SAA built-in function provided only by TSO/E and VM.

```
►► USERID() ►◄
```

Returns the TSO/E user ID, if the REXX exec is running in the TSO/E address space. For example:

```
USERID()    ->    'ARTHUR' /* Maybe */
```

If the exec is running in a non-TSO/E address space, USERID returns one of the following values:

- User ID specified (if the value specified is between one and seveneight characters in length)
- Stepname specified
- Jobname specified

The value that USERID returns is the first one that does not have a null value. For example, if the user ID is null but the stepname is specified, USERID returns the value of the stepname.

You can use TSO/E to replace the routine (module) that is called to determine the value the USERID function returns. This is known as the user ID replaceable routine and is described in “User ID routine” on page 422. You can replace the routine only in non-TSO/E address spaces. Chapter 16, “Replaceable routines and exits,” on page 389 describes replaceable routines in detail and any exceptions to this rule.

VALUE

```
►► VALUE( name ─┬──────────────────┬─ ) ►◄
                └─ , ─ newvalue ───┘
```

returns the value of the symbol that *name* (often constructed dynamically) represents and optionally assigns it a new value. By default, VALUE refers to the current REXX-variables environment. If you use the function to refer to REXX variables, then *name* must be a valid REXX symbol. (You can confirm this by using the SYMBOL function.) Lowercase characters in *name* are translated to uppercase. Substitution in a compound name (see “Compound symbols” on page 19) occurs if possible.

If you specify *newvalue*, then the named variable is assigned this new value. This does not affect the result returned; that is, the function returns the value of *name* as it was before the new assignment.

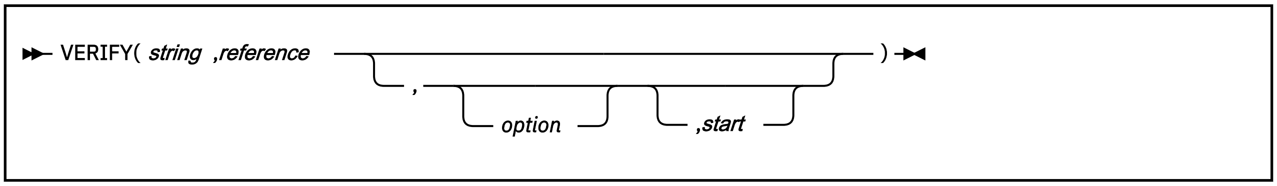
Here are some examples:

```
/* After: Drop A3; A33=7; K=3; fred='K'; list.5='Hi' */
VALUE('a'k)     ->  'A3' /* looks up A3     */
VALUE('fred')   ->  'K'  /* looks up FRED   */
VALUE(fred)     ->  '3'  /* looks up K      */
VALUE(fred,5)   ->  '3'  /* looks up K and  */
                         /* then sets K=5   */
```

```
VALUE(fred)      ->  '5'  /* looks up K       */
VALUE('LIST.'k) ->  'Hi' /* looks up LIST.5 */
```

Guideline: If the VALUE function refers to an uninitialized REXX variable then the default value of the variable is always returned; the NOVALUE condition is not raised. If you specify the *name* as a single literal the symbol is a constant and so the string between the quotation marks can usually replace the whole function call. (For example, `fred=VALUE('k');` is identical with the assignment `fred=k;`, unless the NOVALUE condition is being trapped. See Chapter 7, "Conditions and condition traps," on page 181.)

VERIFY



returns a number that, by default, indicates whether *string* is composed only of characters from *reference*; returns `0` if all characters in *string* are in *reference*, or returns the position of the first character in *string* **not** in *reference*.

The *option* can be either **N**`omatch` (the default) or **M**`atch`. (Only the capitalized and highlighted letter is needed. All characters following it are ignored, and it can be in upper- or lowercase, as usual.) If you specify **M**`atch`, the function returns the position of the first character in *string* that **is** in *reference*, or returns `0` if none of the characters are found.

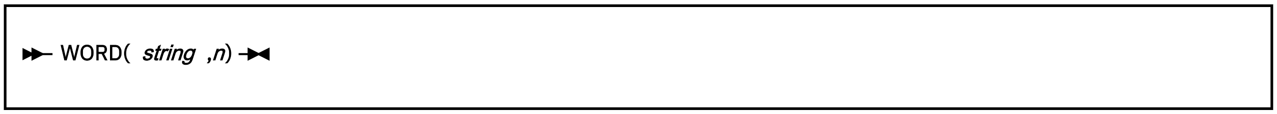
The default for *start* is `1`; thus, the search starts at the first character of *string*. You can override this by specifying a different *start* point, which must be a positive whole number.

If *string* is null, the function returns `0`, regardless of the value of the third argument. Similarly, if *start* is greater than `LENGTH(string)`, the function returns `0`. If *reference* is null, the function returns `0` if you specify **M**`atch`; otherwise the function returns the *start* value.

Here are some examples:

```
VERIFY('123','1234567890')             ->    0
VERIFY('1Z3','1234567890')             ->    2
VERIFY('AB4T','1234567890')            ->    1
VERIFY('AB4T','1234567890','M')        ->    3
VERIFY('AB4T','1234567890','N')        ->    1
VERIFY('1P3Q4','1234567890',,3)        ->    4
VERIFY('123','',N,2)                   ->    2
VERIFY('ABCDE','',,3)                  ->    3
VERIFY('AB3CD5','1234567890','M',4)    ->    6
```

WORD

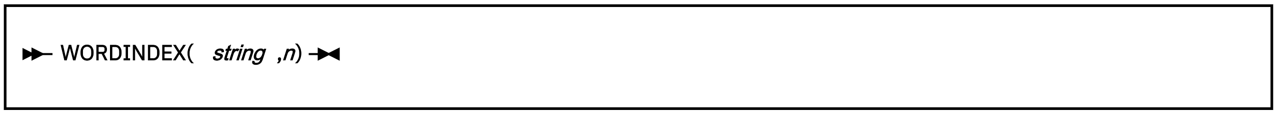


returns the *n*th blank-delimited word in *string* or returns the null string if fewer than *n* words are in *string*. The *n* must be a positive whole number. This function is exactly equivalent to SUBWORD(*string*,*n*,1).

Here are some examples:

```
WORD('Now is the time',3)    ->    'the'
WORD('Now is the time',5)    ->    ''
```

WORDINDEX

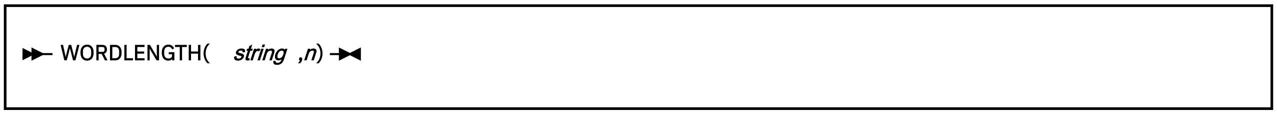


returns the position of the first character in the *n*th blank-delimited word in *string* or returns 0 if fewer than *n* words are in *string*. The *n* must be a positive whole number.

Here are some examples:

```
WORDINDEX('Now is the time',3)    ->    8
WORDINDEX('Now is the time',6)    ->    0
```

WORDLENGTH

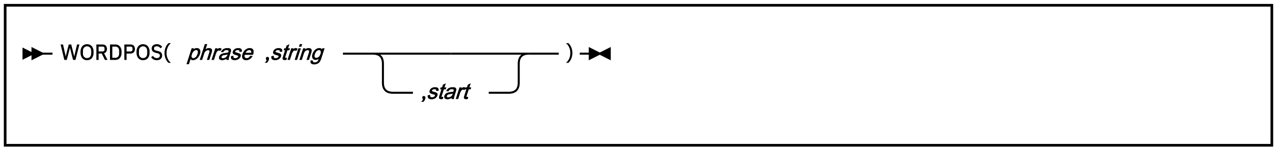


returns the length of the *n*th blank-delimited word in *string* or returns 0 if fewer than *n* words are in *string*. The *n* must be a positive whole number.

Here are some examples:

```
WORDLENGTH('Now is the time',2)       ->    2
WORDLENGTH('Now comes the time',2)    ->    5
WORDLENGTH('Now is the time',6)       ->    0
```

WORDPOS (Word Position)



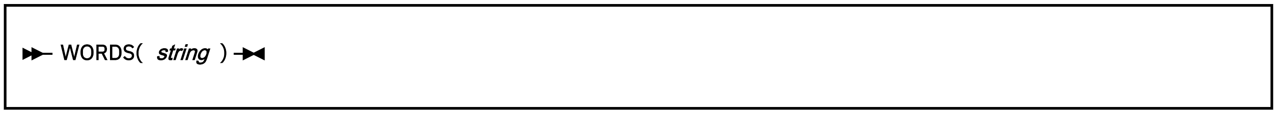
returns the word number of the first word of *phrase* found in *string* or returns 0 if *phrase* contains no words or if *phrase* is not found. Multiple blanks between words in either *phrase* or *string* are treated as a single blank for the comparison, but otherwise the words must match exactly.

By default the search starts at the first word in *string*. You can override this by specifying *start* (which must be positive), the word at which to start the search.

Here are some examples:

```
WORDPOS('the','now is the time')              ->  3
WORDPOS('The','now is the time')              ->  0
WORDPOS('is the','now is the time')           ->  2
WORDPOS('is   the','now is the time')         ->  2
WORDPOS('is   time ','now is   the time')     ->  0
WORDPOS('be','To be or not to be')            ->  2
WORDPOS('be','To be or not to be',3)          ->  6
```

WORDS

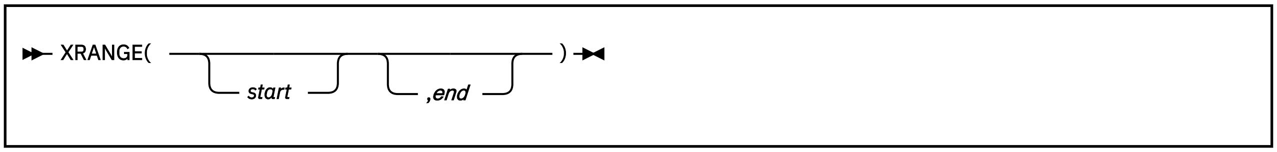


returns the number of blank-delimited words in *string*.

Here are some examples:

```
WORDS('Now is the time')    ->    4
WORDS(' ')                  ->    0
```

XRANGE (Hexadecimal Range)



returns a string of all valid 1-byte encodings (in ascending order) between and including the values *start* and *end*. The default value for *start* is `'00'x`, and the default value for *end* is `'FF'x`. If *start* is greater than *end*, the values wrap from `'FF'x` to `'00'x`. If specified, *start* and *end* must be single characters.

Here are some examples:

```
XRANGE('a','f')      ->   'abcdef'
XRANGE('03'x,'07'x)  ->   '0304050607'x
XRANGE(,'04'x)       ->   '0001020304'x
XRANGE('i','j')      ->   '898A8B8C8D8E8F9091'x  /* EBCDIC */
XRANGE('FE'x,'02'x)  ->   'FEFF000102'x
XRANGE('i','j')      ->   'ij'                   /* ASCII  */
```

X2B (Hexadecimal to Binary)

►► X2B(*hexstring*) ►◄

returns a string, in character format, that represents *hexstring* converted to binary. The *hexstring* is a string of hexadecimal characters. It can be of any length. Each hexadecimal character is converted to a string of four binary digits. You can optionally include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

The returned string has a length that is a multiple of four, and does not include any blanks.

If *hexstring* is null, the function returns a null string.

Here are some examples:

```
X2B('C3')        ->  '11000011'
X2B('7')         ->  '0111'
X2B('1 C1')      ->  '000111000001'
```

You can combine X2B with the functions D2X and C2X to convert numbers or character strings into binary form.

Here are some examples:

```
X2B(C2X('C3'x))  ->  '11000011'
X2B(D2X('129'))  ->  '10000001'
X2B(D2X('12'))   ->  '1100'
```

X2C (Hexadecimal to Character)

►► X2C(*hexstring*) ►◄

returns a string, in character format, that represents *hexstring* converted to character. The returned string is half as many bytes as the original *hexstring*. *hexstring* can be of any length. If necessary, it is padded with a leading 0 to make an even number of hexadecimal digits.

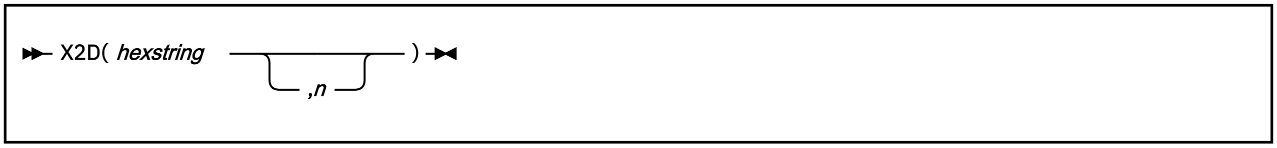
You can optionally include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If *hexstring* is null, the function returns a null string.

Here are some examples:

```
X2C('F7F2 A2')    ->    '72s'       /*  EBCDIC                    */
X2C('F7f2a2')     ->    '72s'       /*  EBCDIC                    */
X2C('F')          ->    ' '         /*  '0F' is unprintable EBCDIC */
```

X2D (Hexadecimal to Decimal)



returns the decimal representation of *hexstring*. The *hexstring* is a string of hexadecimal characters. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS.

You can optionally include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If *hexstring* is null, the function returns `0`.

If you do not specify *n*, *hexstring* is processed as an unsigned binary number.

Here are some examples:

```
X2D('0E')
->    14
X2D('81')        ->    129
X2D('F81')       ->    3969
X2D('FF81')      ->    65409
X2D('F0')        ->    240
X2D('c6 f0'X)    ->    240         /* 'C6 F0'X is equivalent
                                      to'F0'.*/
                                   /* See discussion elsewhere of
                                      hexadecimal strings.*/
X2D('C6F0')      ->    61646       /* 'C6F0'is a 4 digit hex number.*/
```

If you specify *n*, the string is taken as a signed number expressed in *n* hexadecimal digits. If the leftmost bit is off, then the number is positive; otherwise, it is a negative number in two's complement notation. In both cases it is converted to a whole number, which may, therefore, be negative. If *n* is `0`, the function returns 0.

If necessary, *hexstring* is padded on the left with `0` characters (note, not "sign-extended"), or truncated on the left to *n* characters.

Here are some examples:

```
X2D('81',2)      ->    -127
X2D('81',4)      ->    129
X2D('F081',4)    ->    -3967
X2D('F081',3)    ->    129
X2D('F081',2)    ->    -127
X2D('F081',1)    ->    1
X2D('0031',0)    ->    0
```

Implementation maximum: The input string may not have more than 500 hexadecimal characters that will be significant in forming the final result. Leading sign characters (0 and F) do not count towards this total.

TSO/E external functions

TSO/E provides the following external functions you can use to perform different tasks:

- DBCJUSTIFY
- EXTERNALS
- FIND
- GETMSG
- INDEX
- JUSTIFY
- LINESIZE
- LISTDSI
- MSG
- MVSVAR
- OUTTRAP
- PROMPT
- SETLANG
- STORAGE
- SYSCPUS
- SYSDSN
- SYSVAR
- TRAPMSG
- USERID

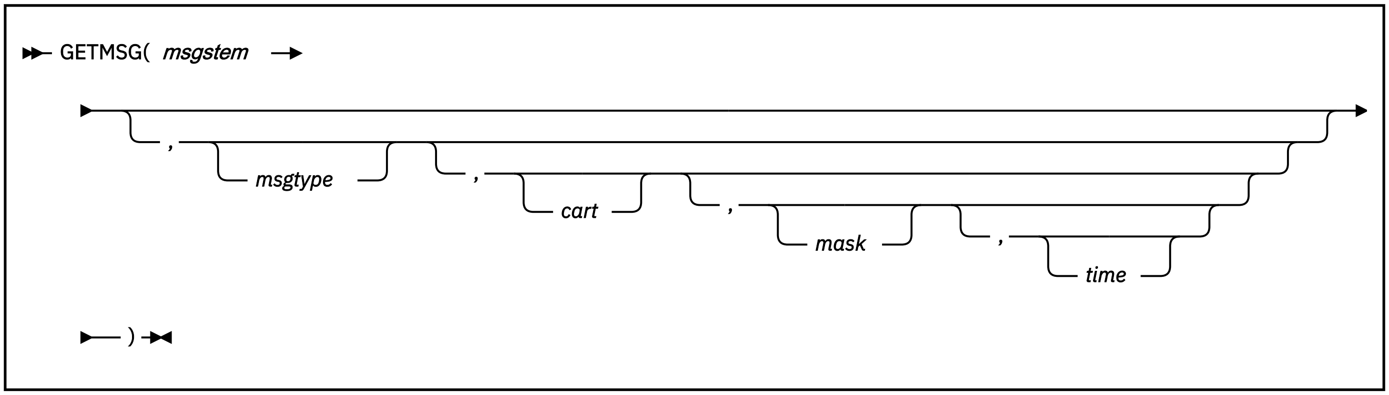
You can use the MVSVAR, SETLANG, STORAGE and SYSCPUS external functions in REXX execs that run **in any address space**, TSO/E and non-TSO/E. You can use the other external functions only in REXX execs that run in the TSO/E address space.

The following topics describe the TSO/E external functions. For general information about the syntax of function calls, see "Syntax" on page 73.

In this section, examples are provided that show how to use the TSO/E external functions. The examples may include data set names. When an example includes a data set name that is not enclosed in either single quotation marks or double quotation marks, the prefix is added to the beginning of the data set name to form the final, fully qualified data set name. If the data set name is enclosed within quotation marks (single or double), it is considered to be fully qualified, and the data set name is used exactly as specified. In the examples, the user ID is the prefix.

Guideline: If you customize REXX processing and use the initialization routine IRXINIT, you can initialize a language processor environment that is not integrated into TSO/E (see "Types of environments - integrated and not integrated into TSO/E" on page 316). You can use the SETLANG and STORAGE external functions in any type of language processor environment. You can use the other TSO/E external functions only if the environment is integrated into TSO/E. Chapter 13, "TSO/E REXX customizing services," on page 305 describes customization and language processor environments in more detail.

GETMSG



GETMSG returns a function code that replaces the function call and retrieves, in variables, a message that has been issued during a console session. Table 4 on page 110 lists the function codes that GETMSG returns.

Use GETMSG during an extended MCS console session that you established using the TSO/E CONSOLE command. Use GETMSG to retrieve messages that are routed to the user's console but that are not being displayed at the user's terminal. The message can be either solicited (a command response) or unsolicited (other system messages), or either. GETMSG retrieves only one message at a time. The message itself may be more than one line. Each line of message text is stored in successive variables. For more information, see **msgstem**.

To use GETMSG, you must:

- Have CONSOLE command authority
- Have solicited or unsolicited messages stored rather than displayed at the terminal during a console session. Your installation may have set up a console profile for you so that the messages are not displayed. You can also use the TSO/E CONSPROF command to specify that solicited or unsolicited messages should not be displayed during a console session.
- Issue the TSO/E CONSOLE command to activate a console session.

You can use the GETMSG function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations: If you use IRXINIT to initialize language processor environments, note that you can use GETMSG only in environments that are integrated into TSO/E (see "Types of environments - integrated and not integrated into TSO/E" on page 316).

Responses to commands sent through the network to another system might be affected as follows:

- The responses might not be returned as solicited even if a CART was specified and preserved; UNSOLDISPLAY(YES) may be required.
- If the receiving system does not preserve the extended console identifier, ROUTCODE(ALL) and UNSOLDISPLAY(YES) might be required to receive the responses.

For information about ROUTCODE, see *z/OS MVS Initialization and Tuning Reference*. For information about UNSOLDISPLAY, see *z/OS TSO/E System Programming Command Reference*.

Table 4 on page 110 lists the function codes that replace the function call. The GETMSG function raises the SYNTAX condition if you specify an incorrect argument on the function call or you specify too many arguments. A SYNTAX condition is also raised if a severe error occurs during GETMSG processing.

Table 4. Function codes for GETMSG that replace the function call

Function code	Description
0	GETMSG processing was successful. GETMSG retrieved the message.

Table 4. Function codes for GETMSG that replace the function call (continued)

Function code	Description
4	GETMSG processing was successful. However, GETMSG did not retrieve the message. There are several reasons why GETMSG may not be able to retrieve the message based on the arguments you specify on the function call. GETMSG returns a function code of 4 if one of the following occurs: • No messages were available to be retrieved • The messages did not match the search criteria you specified on the function call • You specified the `time` argument and the time limit expired before the message was available.
8	GETMSG processing was successful. However, you pressed the attention interrupt key during GETMSG processing. GETMSG did not retrieve the message.
12	GETMSG processing was not successful. A console session is not active. The system issues a message that describes the error. You must issue the TSO/E CONSOLE command to activate a console session.
16	GETMSG processing was not successful. The console session was being deactivated while GETMSG was processing. The system issues a message that describes the error.

The arguments you can specify on the GETMSG function are:

msgstem

the stem of the list of variables into which GETMSG places the message text. To place the message text into compound variables, which allow for indexing, `msgstem` should end with a period (for example, "messg."). GETMSG places each line of the retrieved message into successive variables. For example, if GETMSG retrieves a message that has three lines of text, GETMSG places each line of message text into the variables *messg.1*, *messg.2*, *messg.3*. GETMSG stores the number of lines of message text in the variable ending in 0, *messg.0*.

Note: If *messg.0*=0, no lines are associated with this message. This message might be a delete operator message (DOM) request. For more information about the DOM macro, see *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*.

If `msgstem` does not end with a period, the variable names are appended with consecutive numbers. For example, suppose you specify `msgstem` as "conmsg" (without a period). If GETMSG retrieves a message that has two lines of message text, GETMSG places the text into the variables `conmsg1` and `conmsg2`. The variable `conmsg0` contains the number of lines of message text, which is 2.

In addition to the variables into which GETMSG places the retrieved message text, GETMSG also sets additional variables. The additional variables relate to the field names in the message data block (MDB) for MVS/ESA System Product. For more information about these variables, see Appendix D, “Additional variables that GETMSG sets,” on page 455.

msgtype

the type of message you want to retrieve. Specify one of the following values for *msgtype*:

- SOL

  indicates that you want to retrieve a solicited message. A solicited message is the response from an MVS system or subsystem command.

- UNSOL

  indicates that you want to retrieve an unsolicited message. An unsolicited message is any message that is not issued in response to an MVS system or subsystem command. For example, an unsolicited message may be a message that another user sends you or a broadcast message.

- EITHER

  indicates that you want to retrieve either type of message (solicited or unsolicited). If you do not specify the *msgtype* argument, EITHER is the default.

cart

the command and response token (CART). The CART is a token that lets you associate MVS system commands and subcommands with their responses. When you issue an MVS system or subsystem command, you can specify a CART on the command invocation. To use GETMSG to retrieve a particular message that is in direct response to the command invoked, specify the same CART value.

GETMSG uses the CART you specify as a search argument to obtain the message. If you specify a CART, GETMSG compares the CART you specify with the CARTs for the messages that have been routed to the user's console. GETMSG retrieves the message, only if the CART you specify matches the CART associated with the message. Otherwise, no message is retrieved.

The `cart` argument is used only if you are retrieving solicited messages, that is, the value for the `msgtype` argument is SOL. The CART is ignored if you specify UNSOL or EITHER for `msgtype`.

The `cart` argument is optional. If you do not specify a CART, GETMSG retrieves the oldest message that is available. The type of message retrieved depends on the *msgtype* argument.

For `cart`, you can specify a character string of 1-8 characters or a hexadecimal string of 1-16 hexadecimal digits. For example:

```
'C1D7D7C1F4F9F4F1'X
```

If you specify less than 8 characters or less than 16 hexadecimal digits, the value is padded on the right with blanks. If you specify more than 8 characters or more than 16 hexadecimal digits, the value is truncated to the first 8 characters or 16 digits and no error message is issued.

For more information, see “Using the command and response token (CART) and mask” on page 113.

mask

search argument that GETMSG uses as a mask with the `cart` argument for obtaining a message. If you specify a mask, GETMSG ANDs the mask value with the CART value that you specify on the GETMSG function. GETMSG also ANDs the mask with the CARTs associated with the messages that have been routed to the user's console. GETMSG then compares the results of the AND operations. If a comparison matches, GETMSG retrieves the message. Otherwise, no message is retrieved.

The `mask` argument is valid only if you are retrieving solicited messages and are using a CART. That is, *mask* is valid only if you specify SOL for `msgtype` and you specify the `cart` argument.

The `mask` argument is optional. If you do not specify a mask, GETMSG does not use a mask value when comparing CART values.

For `mask`, you can specify a character string of 1-8 characters or a hexadecimal string of 1-16 hexadecimal digits. For example:

```
'FFFFFFFF00000000'X
```

If you specify less than 8 characters or less than 16 hexadecimal digits, the value is padded on the right with blanks. If you specify more than eight characters or more than 16 hexadecimal digits, the value is truncated to the first eight characters or 16 digits and no error message is issued.

For more information, see “Using the command and response token (CART) and mask” on page 113.

time

the amount of time, in seconds, that GETMSG should wait, if the requested message has not yet been routed to the user's console. If you specify a time value and the time expires before the message is routed to the user's console, GETMSG does not retrieve the message. Otherwise, if the message is available before the time expires, GETMSG retrieves the message.

If you do not specify `time`, GETMSG uses a time value of 0 seconds. If the message has not been routed to the user's console, GETMSG does not retrieve the message.

Overview of using GETMSG during a console session

You can use the GETMSG external function with the TSO/E CONSOLE and CONSPROF commands and the CONSOLE host command environment to write REXX execs that perform MVS operator activities from TSO/E. Using the TSO/E CONSOLE command, you can activate an extended MCS console session with MCS console services. After you activate a console session, you can then use the TSO/E CONSOLE command and the CONSOLE host command environment to issue MVS system and subsystem commands. You can use the TSO/E CONSPROF command to specify that messages that are routed to the user's console during a console session are not to be displayed at the user's terminal. You can then use the GETMSG external function to retrieve messages that are not being displayed and perform different types of processing.

The TSO/E external function SYSVAR has various arguments you can use to determine the type of processing you want to perform. For example, using SYSVAR, you can determine the console session options currently in effect, such as whether solicited and unsolicited messages are being displayed. If you want to display a message that GETMSG retrieved, you can use SYSVAR arguments to obtain information about displaying the message. For example, you can determine whether certain information, such as a time stamp, should be displayed with the message. For more information, see "SYSVAR" on page 148.

Your installation may customize TSO/E to display certain types of information at the terminal in different languages. Your installation can define a primary and secondary language for the display of information. The language codes for the primary and secondary languages are stored in the user profile table (UPT). If your installation customizes TSO/E for different languages, messages that are routed to the user's console during a console session and that are displayed at the user's terminal are displayed in the user's primary or secondary language. However, if you specify that messages are not displayed at the terminal and you then use GETMSG to retrieve the message, the message you retrieve is not in the user's primary or secondary language. The message you retrieve is in US English. For information about customizing TSO/E for different languages, see *z/OS TSO/E Customization*.

For more information about writing execs to perform MVS operator tasks from TSO/E, see Appendix C, "Writing REXX Execs to perform MVS operator activities," on page 449.

Using the command and response token (CART) and mask

The *command and response token* (CART) is a keyword and subcommand for the TSO/E CONSOLE command and an argument on the GETMSG function. You can use the CART to associate MVS system and subsystem commands you issue with their corresponding responses.

To associate MVS system and subsystem commands with their responses, when you issue an MVS command, specify a CART on the command invocation. The CART is then associated with any messages that the command issues. During the console session, solicited messages that are routed to your user's console should not be displayed at the terminal. Use GETMSG to retrieve the solicited message from the command you issued. When you use GETMSG to retrieve the solicited message, specify the same CART that you used on the command invocation.

If several programs use the CONSOLE command's services and run simultaneously in one TSO/E address space, each program must use unique CART values to ensure it retrieves only messages that are intended for that program. You should issue all MVS system and subsystem commands with a CART. Each program should establish an application identifier that the program uses as the first four bytes of the CART. Establishing application identifiers is useful when you use GETMSG to retrieve messages. On GETMSG, you can use both the `cart` and `mask` arguments to ensure you retrieve only messages that begin with the application identifier. Specify the hexadecimal digits FFFFFFFF for at least the first four bytes of the mask value. For example, for the mask, use the value 'FFFFFFFF00000000'X.

For the `cart` argument, specify the application identifier as the first four bytes followed by blanks to pad the value to eight bytes. For example, if you use a four character application identifier of APPL, specify `'APPL    '` for the CART. If you use a hexadecimal application identifier of C19793F7, specify `'C19793F7'`X for the CART. GETMSG ANDs the mask and CART values you specify, and also ANDs the mask with the CART values for the messages. GETMSG compares the results of the AND operations, and if a comparison matches, GETMSG retrieves the message.

You may also want to use CART values if you have an exec using console services that calls a second exec that also uses console services. The CART ensures that each exec retrieves only the messages intended for that exec.

Using different CART values in one exec is useful to retrieve the responses from specific commands and perform appropriate processing based on the command response. In general, it is suggested that your exec uses a CART for issuing commands and retrieving messages. For more information about console sessions and how to use the CART, see Appendix C, “Writing REXX Execs to perform MVS operator activities,” on page 449.

Examples:

The following are some examples of using GETMSG.

1. You want to retrieve a solicited message in variables starting with the stem "CONSMSG.". You do not want GETMSG to wait if the message has not yet been routed to the user's console. Specify GETMSG as follows:

   ```
   msg = GETMSG('CONSMSG.','SOL')
   ```

2. You want to retrieve a solicited message in variables starting with the stem "DISPMSG.". You want GETMSG to wait up to 2 minutes (120 seconds) for the message. Specify GETMSG as follows:

   ```
   mcode = getmsg('dispmsg.','sol',,,120)
   ```

3. You issued an MVS command using a CART value of ‘C1D7D7D3F2F9F6F8’X. You want to retrieve the message that was issued in response to the command and place the message in variables starting with the stem "DMSG". You want GETMSG to wait up to 1 minute (60 seconds) for the message. Specify GETMSG as follows.

   ```
   msgrett = getmsg('dmsg','sol','C1D7D7D3F2F9F6F8'X,,60)
   ```

4. You want to retrieve a list of unsolicited messages, including both single line messages and multiple line messages.

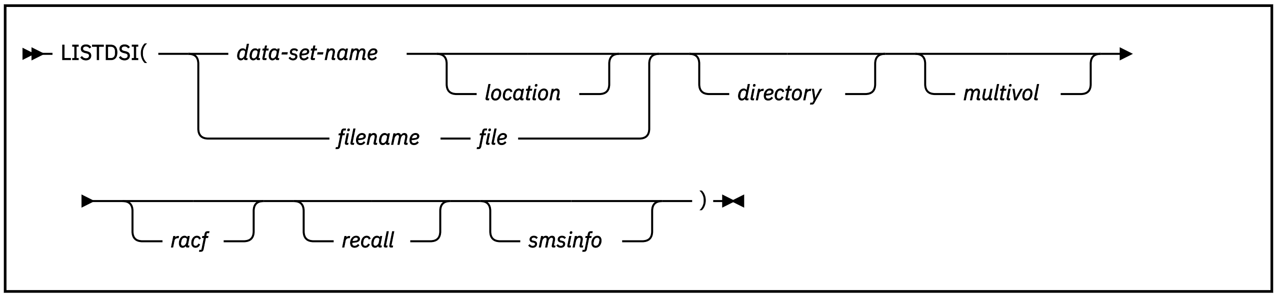
   ```
   mrc = 0
   msgindex = 0
   do while mrc = 0
     mrc = GETMSG('CNSL.','UNSOL',,,3)
     if mrc > 0 then leave
     do i = 1 to CNSL.0
       msgindex = msgindex+1
       msg.msgindex = CNSL.i
     end
   end

   msg.0 = msgindex

   do i = 1 to msg.0
     say msg.i
   end
   ```

5. Your exec has defined an application identifier of APPL for using CARTs. Whenever you issue an MVS command, you specify a CART of APPLxxxx, where xxxx is a four-digit number. For example, for the first MVS command, you use a CART of APPL0001. For the second MVS command, you use a CART of APPL0002, and so on.

   You want to use GETMSG to retrieve solicited messages that are intended only for your exec. You can specify the `mask` and `cart` arguments to ensure that GETMSG retrieves only messages that are for the MVS commands your exec invoked. Specify `'FFFFFFFF00000000'X` for the mask. Specify `'APPL    '` (padded with blanks to 8 characters) for the CART. You also want to wait up to 30 seconds for the message.

   ```
   conmess = getmsg('msgc.','sol','APPL    ','FFFFFFFF00000000'X,30)
   ```

LISTDSI



LISTDSI returns one of the following function codes that replace the function call, and retrieves information about a data set's allocation, protection, and directory and stores it in specific variables. Table 5 on page 115 shows the function codes that replace the function call.

Table 5. Function codes for LISTDSI that replace the function call

Function code	Description
0	LISTDSI processing was successful. Data set information was retrieved.
4	LISTDSI processing was successful. Some data set information is unavailable. Review the reason code in the variable SYSREASON and check the messages that are returned returned in SYSMSGLVL1 and SYSMSGLVL2 to determine which information is unavailable.
16	LISTDSI processing was not successful. An error occurred. None of the variables containing information about the data set can be considered valid, except for SYSREASON. The SYSREASON variable contains the LISTDSI reason code (see **Reason Codes**).

Note: To be compatible with CLIST processing, a function code of 16 is provided. LISTDSI does not raise the syntax condition in this case, even though the processing was not successful.

You can use LISTDSI to obtain information about a data set that is available on DASD. LISTDSI does not directly support data that is on tape. LISTDSI supports generation data group (GDG) data sets when using absolute generation names, but does not support relative GDG names. LISTDSI does not support hierarchical file system (HFS) data sets. Unpredictable results may occur.

LISTDSI is passed a single argument string. That string might consist of several values which are the parameters to LISTDSI, separated by one or more blanks. For example:

```
argument_string = "REXXEXEC VOLUME(PACK1) NODIRECTORY NORECALL"
x = LISTDSI(argument_string)
```

If LISTDSI causes a syntax error (for example, if you specify more than one argument string), a function code is not returned. In addition, none of the LISTDSI variables are set.

The variables in which LISTDSI stores data set information are described in Table 6 on page 118.

To suppress TSO/E messages issued by the LISTDSI function, use the MSG("OFF") function. For information about the MSG function, see “MSG” on page 127.

The argument strings that you can specify on the LISTDSI function are:

data-set-name

the name of the data set about which you want to retrieve information. See **Specifying Data Set Names** for more information.

location

specifies how you want the data set (as specified in *data-set-name*) located. You can specify *location*, only if you specify a data set name, not a *filename*. For *location*, specify one of the following values.

If you do not specify either VOLUME or PREALLOC, the system locates the data set through catalog search.

- 'VOLUME(serial ID)' specifies the serial number of the volume where the data set is located.
- 'PREALLOC' specifies that the location of the specified data set is determined by allocating the data set, rather than through a catalog search. PREALLOC allows data sets that have been previously allocated to be located without searching a catalog and allows unmounted volumes to be mounted.

filename
the name of an allocated file (ddname) about which you want to retrieve information.

file
you must specify the word "FILE" if you specify *filename* instead of *data-set-name*. If you do not specify FILE, LISTDSI assumes that you specified a data-set-name.

directory
indicates whether you want directory information for a partitioned data set (PDS or PDSE). For *directory*, specify one of the following:

- 'DIRECTORY' indicates that you want directory information.

  Requesting DIRECTORY information for a PDS might cause the date last referenced (SYSREFDATE) to be updated by LISTDSI. Refer to the description of the SYSREFDATE variable for more information about when SYSREFDATE might be updated by LISTDSI.
- 'NODIRECTORY' indicates that you do not want directory information. If you do not require directory information, NODIRECTORY can significantly improve processing. NODIRECTORY is the default.

multivol
Specify MULTIVOL or NOMULTIVOL. The default if not specified is NOMULTIVOL. Indicates whether data size calculations should include all volumes, when a data set resides on more than one volume. The new SYSNUMVOLS and SYSVOLUMES variables are not affected by this operand, as these are always set.

If the VOLUME keyword and the MULTIVOL keyword are both specified, the MULTIVOL keyword is ignored. In this case, data set size information is returned just for the specified volume.

racf
Specify RACF or NORACF. The default if not specified is RACF. Indicates whether a check for RACF authority is done or not. If not done, the data set is not opened by LISTDSI, for example, to read directory information.

recall
indicates whether you want to recall a data set migrated by Data Facility Hierarchical Storage Manager (DFHSM). For *recall*, specify one of the following:

- 'RECALL' indicates that you want to recall a data set migrated by DFHSM. The system recalls the data set regardless of its level of migration or the type of device to which it has been migrated.
- 'NORECALL' indicates that you do not want to recall a data set. If the data set is migrated, the system stores an error message.

  If you do not specify either RECALL or NORECALL, the system recalls the data set only if it has been migrated to a direct access storage device (DASD).

smsinfo
indicates whether you want System Managed Storage (SMS) information about an SMS-managed data set. This information includes

- type of data set
- used space
- data class name
- storage class name
- management class name.

See also the following figure where the corresponding REXX variables are described.

For *smsinfo*, specify one of the following:

- 'SMSINFO' indicates that you want SMS information about *data-set-name* or *filename*. Neither *data-set-name* nor *filename* might refer to a VSAM index or data component.

  If the specified data set is not managed by SMS, LISTDSI continues, but no SMS information is provided in the corresponding REXX variables.

  Specify SMSINFO only if you want SMS information about a data set. NOSMSINFO (the default) might significantly reduce the execution time of the LISTDSI statement.

  Requesting SMSINFO for a PDSE data set might cause the date last referenced (SYSREFDATE) to be updated by LISTDSI. See the description of the SYSREFDATE variable for more information about when SYSREFDATE might be updated by LISTDSI.
- 'NOSMSINFO' indicates that you do not want SMS information about the specified data set. NOSMSINFO is the default.

You can use the LISTDSI function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations If you use IRXINIT to initialize language processor environments, note that you can use LISTDSI only in environments that are integrated into TSO/E (see "Types of environments - integrated and not integrated into TSO/E" on page 316).

You can use the LISTDSI information to determine whether the data set is the right size or has the right organization or format for a given task. You can also use the LISTDSI information as input to the ALLOCATE command, for example, to create a new data set using some attributes from the old data set while modifying others.

If you use LISTDSI to retrieve information about a VSAM data set, LISTDSI stores only the volume serial ID (in variable SYSVOLUME), the device unit (in variable SYSUNIT), and the data set organization (in variable SYSDSORG).

If you use LISTDSI to retrieve information about a multiple volume data set, LISTDSI stores information for the first volume only. Similarly, if you specify a file name or you specify PREALLOC for *location* and you have other data sets allocated to the same file name, the system might not retrieve information for the data set you wanted.

LISTDSI

When you use LISTDSI to obtain information about a file, LISTDSI returns information only about the first data set in the file, if the file consists of a concatenation of more than one data set. Likewise, if the ddname specified by *filename* points to a multi-volume data set, LISTDSI can return information only about the first volume, and is not able to detect that the data is multi-volume.

If the data set is SMS managed and is capable of expanding to multiple volumes, but did not, it is considered a single volume data set by LISTDSI until it expands to the second volume. In any case, LISTDSI only retrieves information for the first volume that is referenced by the request.

As part of its processing, LISTDSI issues a RACF authority check against the provided data set which causes a security violation to occur if the user does not have at least READ access to the data set. RACF does not issue an ICH408I message due to message suppression requested by LISTDSI, and therefore LISTDSI issues a return code of 0 (NODIRECTORY specified) or 4 (DIRECTORY specified). The only indication a security violation occurred is that an SMF type-80 RACF audit record is created.

LISTDSI considers file names in the form SYSnnnnn as system-generated file names. If LISTDSI is used to obtain information about a data set that was pre-allocated multiple times using a file name of the form SYSnnnnn, an existing file might be unintentionally freed.

Specifying Data Set Names

On the LISTDSI function, if you use *data-set-name* instead of *filename*, you can specify the name of a sequential data set or a partitioned data set (PDS). You can specify the *data-set-name* in any of the following ways:

- Non fully-qualified data set name that follows the naming conventions — When there is only one set of quotation marks or no quotation marks, TSO/E adds your prefix to the data set name.

```
    x = LISTDSI('myrexx.exec')

    x = LISTDSI(myrexx.exec)
```

- Fully-qualified data set name — The extra quotation marks prevent TSO/E from adding your prefix to the data set name.

```
    x = LISTDSI("'sys1.proj.new'")

    x = LISTDSI('''sys1.proj.new''')
```

- Variable name that represents a fully-qualified or non fully-qualified data set name — The variable name must not be enclosed in quotation marks because quotation marks prevent variable substitution. An example of using a variable for a fully-qualified data set name is:

```
/* REXX program for ....   */
⋮
var1 = "'sys1.proj.monthly'"
⋮
dsinfo = LISTDSI(var1)
⋮
EXIT
```

Variables that LISTDSI sets

Table 6 on page 118 describes the variables that LISTDSI sets. For VSAM data sets, only the variables SYSDSNAME, SYSEATTR, SYSEADSCB, SYSVOLUME, SYSUNIT, and SYSDSORG are accurate; all other variables are set to question marks.

Table 6. Variables that LISTDSI sets

Variable	Contents
SYSDSNAME	Data set name
SYSVOLUME	Volume serial ID
SYSUNIT	Generic device type on which volume resides, for example 3390.

Table 6. Variables that LISTDSI sets (continued)

Variable	Contents
SYSDSORG	Data set organization: **PS** - Physical sequential **PSU** - Physical sequential unmovable **DA** - Direct organization **DAU** - Direct organization unmovable **IS** - Indexed sequential **ISU** - Indexed sequential unmovable **PO** - Partitioned organization **POU** - Partitioned organization unmovable **VS** - VSAM **???** - Unknown
SYSRECFM	Record format; one to six character combination of the following: **U** - Records of undefined length **F** - Records of fixed length **V** - Records of variable length **T** - Records written with the track overflow feature of the device (no currently supported device supports the track overflow feature) **B** - Records blocked **S** - Records written as standard or spanned variable-length blocks **A** - Records contain ANSI control characters **M** - Records contain machine code control characters **??????** - Unknown
SYSLRECL	Logical record length
SYSBLKSIZE	Block size
SYSKEYLEN	Key length
SYSALLOC	Allocation, in space units
SYSUSED	Allocation used, in space units. For a partitioned data set extended (PDSE), 'N/A' is returned; see the description of the variable SYSUSEDPAGES for used space of a PDSE.

Table 6. Variables that LISTDSI sets (continued)

Variable	Contents
SYSUSEDPAGES	The used space of a partitioned data set extended (PDSE) in 4K pages.
SYSPRIMARY	Primary allocation in space units
SYSSECONDS	Secondary allocation in space units
SYSUNITS	Space units: **CYLINDER** - Space units in cylinders **TRACK** - Space units in tracks **BLOCK** - Space units in blocks **????????** - Space units are unknown
SYSEXTENTS	Number of extents allocated
SYSUSEDEXTENTS	Indicates the number of extents used. For a partitioned data set extended (PDSE), this variable returns 'N/A'; see the descriptions of variables SYSUSEDPAGES and SYSUSEDPERCENT for more information about used space of a PDSE.
SYSCREATE	Creation date Year/day format, for example: 1990/102
SYSREFDATE	Last referenced date Year/day format, for example: 2010/107 Specifying DIRECTORY or SMSINFO may cause the last referenced date to be updated to the current date under the following circumstances: • Specifying DIRECTORY causes the date to be updated only if the data set is a PDS and the user running LISTDSI has RACF READ authority to the data set. In all other cases, including when the data set is a PDSE, DIRECTORY has no effect on this date. • Specifying SMSINFO causes the date to be updated only if the data set is a PDSE and the user running LISTDSI has RACF READ authority to the data set. In all other cases, SMSINFO has no effect on this date.
SYSEXDATE	Expiration date Year/day format, for example: 1990/365
SYSPASSWORD	Password indication: **NONE** - No password protection **READ** - Password required to read **WRITE** - Password required to write
SYSRACFA	RACF indication: **NONE** - No RACF protection **GENERIC** - Generic profile covers this data set **DISCRETE** - Discrete profile covers this data set

Table 6. Variables that LISTDSI sets (continued)

Variable	Contents
SYSUPDATED	Backup change indicator: **YES** - Data set is updated since its last backup by DFSMShsm (or its equivalent). **NO** - Data set has not been updated since its last backup.
SYSTRKSCYL	Tracks per cylinder for the unit identified in the SYSUNIT variable
SYSBLKSTRK	Blocks (whose size is given in variable SYSBLKSIZE) per track for the unit identified in the SYSUNIT variable. For a PDSE, the value "N/A" is returned because a block of size SYSBLKSIZE can 'span' a track in a PDSE. The value contained in SYSUSEDPAGES is a more meaningful measurement of space usage for a PDSE.
SYSADIRBLK	For a partitioned data set (PDS) the number of directory blocks allocated are returned. For a partitioned data set extended (PDSE), "NO_LIM" is returned because there is no static allocation for its directory. A value is returned only if DIRECTORY is specified on the LISTDSI statement.
SYSUDIRBLK	For a partitioned data set (PDS) the number of directory blocks used are returned. For a partitioned data set extended (PDSE), "N/A" is returned because it is not a static value. A value is returned only if DIRECTORY is specified on the LISTDSI statement.
SYSMEMBERS	Number of members - returned only for partitioned data sets when DIRECTORY is specified
SYSREASON	LISTDSI reason code
SYSMSGLVL1	First-level message if an error occurred
SYSMSGLVL2	Second-level message if an error occurred
SYSDSSMS	Contains information about the type of a data set. If the SMS data set type could not be retrieved, the SYSDSSMS variable contains: **SEQ** for a sequential data set **PDS** for a partitioned data set **PDSE** for a partitioned data set extended. If the data set is a PDSE and the SMSINFO operand was specified on the LISTDSI call and SMS data set type information could be retrieved, the SYSDSSMS variable contains: **LIBRARY** for an empty PDSE **PROGRAM_LIBRARY** for a partitioned data set extended program library **DATA_LIBRARY** for a partitioned data set extended data library. **Note:** This detailed data set type information for a PDSE is not returned if the user issuing the LISTDSI call does not have RACF READ authority to the data set.
SYSDATACLASS([1])	The SMS data class name - returned only if SMSINFO is specified on the LISTDSI statement and the data set is managed by SMS.
SYSSTORCLASS([1])	The SMS storage class name - returned only if SMSINFO is specified on the LISTDSI statement and the data set is managed by SMS.
SYSMGMTCLASS([1])	The SMS management class name - returned only if SMSINFO is specified on the LISTDSI statement and the data set is managed by SMS.

Table 6. Variables that LISTDSI sets (continued)

Variable	Contents
SYSSEQDSNTYPE	Indicates the type of sequential data set: BASIC - regular sequential data set LARGE - large sequential data set EXTENDED - extended sequential data set If the data set is not sequential, this variable returns a null string.
SYSEATTR	Indicates the current status of the EATTR bits in the DSCB that describe the EAS eligibility status of a data set. The EAS can only contain data sets that are EAS-eligible. Default blank indicates that the EATTR bits are ‘00’b. The defaults for EAS eligibility apply: - VSAM data sets are EAS-eligible, and can have extended attributes (format 8 and 9 DSCBs). - Non-VSAM data sets are EAS-eligible if EATTR=OPT is in effect. In that case, they can have extended attributes (format 8 and 9 DSCBs). If the data set does not now have those attributes, it can gain them by being moved to a volume that supports them. **NO** Indicates that ‘01’b is specified for the EATTR bits. The data set is not EAS-eligible, and cannot have extended attributes (format 8 and 9 DSCBs). **OPT** Indicates that ‘10’b is specified for the EATTR bits. The data set is ESA-eligible, and can have extended attributes (format 8 and 9 DSCBs).
SYSEADSCB	Indicates whether the data set has extended attributes: **YES** The data set has extended attributes (format 8 and 9 DSCBs) and can reside in the EAS. **NO** The data set does not have extended attributes (format 8 and 9 DSCBs) and cannot reside in the EAS.
SYSALLOCPAGES	Indicates number of pages allocated to a PDSE.
SYSUSEDPERCENT	Indicates percentage of pages used out of pages allocated for a PDSE. This is a number from 0 to 100, rounded down to the nearest integer value.
SYSNUMVOLS	Indicates a number from 1 to 59 as tape is supported, but always returns 1 if a volume is specified, instead of trying LOCATE all volumes for the data set.
SYSVOLUMES	Indicates up to 412 characters with a list of volumes separated by spaces where the first six characters always matches SYSVOLUME and each volume name takes up six spaces padded with blanks to help simplify parsing. If a volume is specified on the LISTDSI call, just that volume name is returned.
SYSCREATETIME	Indicates the time a data set was created in the format *hh:mm:ss* where *hh* is hours since midnight, *mm* is minutes since midnight, and *ss* is seconds since midnight. This variable is set only if the data set has extended attributes. That means that the value of SYSEADSCB is YES. This variable can be used together with the SYSCREATE variable to determine the date and time when the data set was created.
SYSCREATESTEP	Indicates the name of the job step that created the data set. This variable is set only if the value of SYSEADSCB is YES.

Table 6. Variables that LISTDSI sets (continued)

Variable	Contents
SYSCREATEJOB	Indicates the name of the job that created the data set. This variable is set only if the value of SYSEADSCB is YES.
SYSDSVERSION	Returns the PDSE data set version. By default, the original PDSE data sets are version 1. This variable is returned by LISTDSI for a PDSE data set when the DIR operand is also specified.
SYSENCRYPT	Returns the encryption status for data set: • 'YES' - if the data set is encrypted, or • 'NO' - if the data set is not encrypted. If SYSENCRYPT returns 'YES', then SYSKEYLABEL is also returned.
SYSKEYLABEL	Returns the encryption key label of an encrypted data set. It only returns information if SYSENCRYPT is set to 'YES', indicating that the data set is encrypted. In all other cases, this variable is not set.
SYSMAXGENS	Maximum number of generations of a PDSE member that can be maintained. This number can be between 0 and 2147483647. This variable is only returned when the LISTDSI data set is a PDSE and the DIR operand of LISTDSI was also specified. In all other cases, this variable is not set.

Note: For data sets not managed by SMS, these variables return a null string.

Reason Codes

Reason codes from the LISTDSI function appear in variable SYSREASON. Table 7 on page 123 shows the LISTDSI reason codes. With each reason code the REXX variable SYSMSGLVL2 is set to message IKJ584*nn*I, where *nn* is the reason code. These messages are described in *z/OS TSO/E Messages*.

Table 7. LISTDSI reason codes

Reason code	Description
0	Normal completion.
1	Error parsing the function.
2	Dynamic allocation processing error.
3	The data set is a type that cannot be processed.
4	Error determining UNIT name.
5	Data set not cataloged.
6	Error obtaining the data set name.
7	Error finding device type.
8	The data set does not reside on a direct access storage device.
9	DFHSM migrated the data set. NORECALL prevents retrieval.
11	Directory information was requested, but you lack authority to access the data set.
12	VSAM data sets are not supported.
13	The data set could not be opened.
14	Device type not found in unit control block (UCB) tables.
17	System or user abend occurred.
18	Partial data set information was obtained.

Table 7. LISTDSI reason codes (continued)

Reason code	Description
19	Data set resides on multiple volumes. Consider using the MULTIVOL keyword to obtain data set size information totaled across all volumes.
20	Device type not found in eligible device table (EDT).
21	Catalog error trying to locate the data set.
22	Volume not mounted.
23	Permanent I/O error on volume.
24	Data set not found.
25	Data set migrated to non-DASD device.
26	Data set on MSS (Mass Storage) device. **Note:** This device type is no longer supported.
27	No volume serial is allocated to the data set.
28	The ddname must be one to eight characters.
29	Data set name or ddname must be specified.
30	Data set is not SMS-managed.
31	ISITMGD macro returned with bad return code and reason code. Return code and reason code can be found in message IKJ58431I, which is returned in variable &SYSMSGLVL2.
32	Unable to retrieve SMS information. DFSMS has incorrect level.
33	Unable to retrieve SMS information. SMS is not active.
34	Unable to retrieve SMS information. OPEN error.
35	Unexpected error from DFSMS internal service IGWFAMS.
36	Unexpected error from the SMS service module.
37	Unexpected error from DFSMS service IGGCSI00.

Examples:

The following are some examples of using LISTDSI.

1. To set variables with information about data set USERID.WORK.EXEC, use the LISTDSI function as follows:

```
x = LISTDSI(work.exec)
SAY 'Function code from LISTDSI is:                 ' x
SAY 'The data set name is:                          ' sysdsname
SAY 'The device unit on which the volume resides is:' sysunit
SAY 'The record format is:                          ' sysrecfm
SAY 'The logical record length is:                  ' syslrecl
SAY 'The block size is:                             ' sysblksize
SAY 'The allocation in space units is:              ' sysalloc
SAY 'Type of RACF protection is:                    ' sysracfa
```

   Output from the example might be:

```
Function code from LISTDSI is:                   0
The data set name is:                            USERID.WORK.EXEC
The device unit on which the volume resides is:  3380
The record format is:                            VB
The logical record length is:                    255
The block size is:                               6124
The allocation in space units is:                33
Type of RACF protection is:                      GENERIC
```

2. To retrieve information about the DD called APPLPAY, you can use LISTDSI as follows:

```
ddinfo = LISTDSI("applpay" "FILE")
```

3. Suppose you want to retrieve information about a PDS called SYS1.APPL.PAYROLL, including directory information. You do not want the PDS to be located through a catalog search, but have the location determined by the allocation of the data set. You can specify LISTDSI as follows:

```
/*  REXX program for ....   */
⋮
var1 = "'sys1.appl.payroll'"
infod = "directory"
⋮
pdsinfo = LISTDSI(var1 infod "prealloc")
⋮
EXIT
```

   In the example, the variable *var1* was assigned the name of the PDS (SYS1.APPL.PAYROLL). Therefore, in the LISTDSI function call, *var1* is not enclosed in quotation marks to allow for variable substitution. Similarly, the variable *infod* was assigned the value "directory", so in the LISTDSI function, *infod* becomes the word "directory". The PREALLOC argument is enclosed in quotation marks to prevent any type of substitution. After the language processor evaluates the LISTDSI function, it results in the following function call being processed:

```
LISTDSI('sys1.appl.payroll' directory prealloc)
```

4. The LISTDSI function issues message IKJ56709I if a syntactically invalid data set name is passed to the function. To prevent this message from being displayed, use the MSG('OFF') function.

```
dsname = "'ABCDEFGHIJ.XYZ'"  /*syntactically invalid name,
                               because a qualifier is longer
                               than 8 characters               */

msgval = MSG('OFF')          /* save current MSG value and
                                suppress messages              */
x = LISTDSI(dsname)          /* Retrieve data set information  */
say 'Function Code returned by LISTDSI ==> ' x
msgval = MSG(msgval)         /* Restore MSG setting            */
exit 0
```

5. To use LISTDSI to set variables with information about a multivolume data set, 'BILL.WORK.MULTI01', use the LISTDSI function with the MULTIVOL operand as follows. Because the data set may reside on multiple volumes, code the MULTIVOL keyword to obtain data set size information aggregated across all volumes of the data set. SYSNUMVOLS returns the number of volumes allocated, and SYSVOLUMES is a list of the volumes on which the data set is allocated.

```
/* REXX - Test LISTDSI V2R1 example with MULYTIVOL usage             */
/*                                                                   */
/*********************************************************************/
/*  To set variables with information about a possibly               */
/*  multivolume data set 'BILL.WORK.MULTI01', use the LISTDSI        */
/*  function as follows. Because the data set may be on multiple     */
/*  volumes, code the MULTIVOL keyword to obtain data set size       */
/*  information aggregated across volumes of the data set.           */
/*  SYSNUMVOLS returns the number of volumes allocated, and          */
/*  SYSVOLUMES is a list of the volumes on which the data set is     */
/*  allocated.                                                       */
/*********************************************************************/
 x = LISTDSI('''BILL.WORK.MULTI01''' MULTIVOL)
 SAY 'Function code from LISTDSI is: '  x
 SAY 'The data set name is ....... : ' sysdsname
 SAY 'The device unit on which the volume resides is: ' sysunit
 SAY 'The data set organization is : ' sysdsorg
 SAY 'The record format is ....... : ' sysrecfm
 SAY 'The logical record length is : ' syslrecl
 SAY 'The block size is .......... : ' sysblksize
 SAY 'Type of RACF protection is . : ' sysracfa
 SAY

 SAY 'The space unit of allocation is ......... : ' sysunits
 SAY 'The primary allocation in space units is. : ' sysprimary
 SAY 'The secondary allocation in space units is: ' sysseconds
 SAY 'The space allocated in space units ...... : ' sysalloc
 SAY 'The spaced used in space units is ....... : ' sysused
 SAY 'The number of extents allocated is ...... : ' sysextents
 SAY 'The number of extents in use is ......... : ' sysusedextents
 SAY 'The first (or possibly only volume) is .. : ' sysvolume
 SAY
 SAY 'Number of volumes on which data set resides is: ' sysnumvols
  if sysnumvols > 1 then
    SAY 'Data Set is on multiple vols (vollist) ... :<'sysvolumes'>'
  else
    SAY 'Data set is on a single vol: ............ :<'sysvolumes'>'
  /****************************************************************/
  /*  Explain any function error code                             */
  /****************************************************************/
  IF x ^= 0 then       /* Report on any non-0 function code       */
    do
      say
      say '-------------------------------------'
      say 'LISTDSI Function Code is : ' x
      say 'LISTDSI Reason Code is   : ' sysreason
      say 'Error messages returned  : '
      say '==>' SYSMSGLVL1
      say '==>' SYSMSGLVL2
      say '-------------------------------------'
    end

 exit 0


 Output from this example might be:
 Function code from LISTDSI is:  0
 The data set name is ....... :  BILL.WORK.MULTI01
 The device unit on which the volume resides is:  3390
 The data set organization is :  PS
 The record format is ....... :  FB
 The logical record length is :  4160
 The block size is .......... :  20800
 Type of RACF protection is . :  GENERIC

 The space unit of allocation is ......... :  TRACK
 The primary allocation in space units is. :  3
 The secondary allocation in space units is:  5
 The space allocated in space units ...... :  193
 The spaced used in space units is ....... :  162
 The number of extents allocated is ...... :  39
 The number of extents in use is ......... :  33
 The first (or possibly only volume) is .. :  SL2B07

 Number of volumes on which data set resides is:  3
 Data Set is on multiple vols (vollist) ... :<SL2B07 SL2B0C SL2B05>
```

6. In the previous example with a multivolume data set, if you just want to retrieve information about the part of the data set on the second volume, "SL2BOC", you could code the LISTDSI call in the example with the VOLUME operand, while omitting the MULTIVOL operand. Because information about a specific volume is requested, SYSNUMVOLS returns 1 (for the requested volume), and SYSVOLUME and SYSVOLUMES both return just the specified volume name.

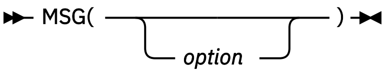
```
With this invocation in the above exec:
 x = LISTDSI('''BILL.WORK.MULTI01''' 'VOLUME(SL2B0C)')

The output might look something like the following :

Function code from LISTDSI is:  0
The data set name is ....... :  BILL.WORK.MULTI01
The device unit on which the volume resides is:  3390
The data set organization is :  PS
The record format is ....... :  FB
The logical record length is :  4160
The block size is .......... :  20800
Type of RACF protection is . :  GENERIC

The space unit of allocation is ......... :  TRACK
The primary allocation in space units is. :  5
The secondary allocation in space units is:  5
The space allocated in space units ...... :  80
The spaced used in space units is ....... :  80
The number of extents allocated is ...... :  16
The number of extents in use is ......... :  16
The first (or possibly only volume) is .. :  SL2B0C
```

```
Number of volumes on which data set resides is:  1
Data set is on a single vol: ............. :<SL2B0C>
```

Special Considerations

LISTDSI considers file names starting with ‘SYS’ followed by five digits to be system-generated file names. If you use LISTDSI to obtain information about a data set that was preallocated multiple times using file names starting with ‘SYS’ followed by five digits, an existing file may be freed.

MSG



MSG returns the value ON or OFF, which indicates the status of the displaying of TSO/E messages. That is, MSG indicates whether TSO/E messages are being displayed while the exec is running.

Using MSG, you can control the display of TSO/E messages from TSO/E commands and TSO/E external functions. Use the following options to control the display of TSO/E informational messages. Informational messages are automatically displayed unless an exec uses MSG(OFF) to inhibit their display.

ON
: returns the previous status of message issuing (ON or OFF) and allows TSO/E informational messages to be displayed while an exec is running.

OFF
: returns the previous status of message issuing (ON or OFF) and inhibits the display of TSO/E informational messages while an exec is running.

Here are some examples:

```
msgstat = MSG()     ->   'OFF'  /* returns current setting (OFF)      */
stat = MSG('off')   ->   'ON'   /* returns previous setting (ON) and
                                   inhibits message display           */
```

You can use the MSG function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations: If you use IRXINIT to initialize language processor environments, note that you can use MSG only in environments that are integrated into TSO/E (see “Types of environments - integrated and not integrated into TSO/E” on page 316).

When an exec uses the MSG(OFF) function to inhibit the display of TSO/E messages, messages are not issued while the exec runs and while functions and subroutines called by that exec run. The displaying of TSO/E messages resumes if you use the MSG(ON) function or when the original exec ends. However, if an exec invokes another exec or CLIST using the EXEC command, message issuing status from the invoking exec is not carried over into the newly-invoked program. The newly-invoked program automatically displays TSO/E messages, which is the default.

The MSG function is functionally equivalent to the CONTROL MSG and CONTROL NOMSG statements for TSO/E CLISTs.

In non-TSO/E address spaces, you cannot control message output using the MSG function. However, if you use the TRACE OFF keyword instruction, messages do not go to the output file (SYSTSPRT, by default).

For more information about trapping or suppressing output with the MSG function, see “OUTTRAP versus MSG function when trapping or suppressing output” on page 39.

Examples:

The following are some examples of using MSG.

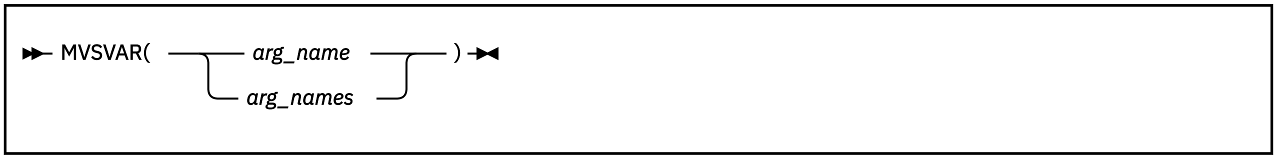
1. To inhibit the display of TSO/E informational messages while an exec is running, use MSG as follows:

```
msg_status = MSG("OFF")
```

2. To ensure that messages associated with the TSO/E TRANSMIT command are not displayed before including the TRANSMIT command in an exec, use the MSG function as follows:

```
IF MSG() = 'OFF' THEN,
  "TRANSMIT node.userid DA(myrexx.exec)"
ELSE
  DO
    x = MSG("OFF")
    "TRANSMIT node.userid DA(myrexx.exec)"
    a = MSG(x)        /* resets message value */
  END
```

MVSVAR



MVSVAR returns information about MVS, TSO/E, and the current session, such as the symbolic name of the MVS system, or the security label of the TSO/E session.

The MVSVAR function is available **in any MVS address space.**

The information that is returned depends on the *arg_name* or *arg_names* value that is specified on the function call. In most cases only one argument arg_name) is allowed, with the exception that two arguments (arg_names) are needed if the first argument is SYMDEF, which requests the value of the symbolic name that is specified by the second argument. The following items of information are available for retrieval:

SYSAPPCLU
The APPC/MVS logical unit (LU) name.

SYSDFP
The level of DFSMSdfp, a basic element of the operating system.

SYSMVS
The level of the base control program (BCP) component of z/OS.

SYSNAME
The name of the system your REXX exec is running on, as specified in the SYSNAME statement in SYS1.PARMLIB member IEASYSxx.

SYSOPSYS
The z/OS name, version, release, modification level, and FMID.

SYSSECLAB
The security label (SECLABEL) name of the TSO/E session.

SYSSMFID
Identification of the system on which System Management Facilities (SMF) is active.

SYSSMS
Indicator whether SMS (storage management subsystem) is running on the system.

SYSCLONE
MVS system symbol representing its system name.

SYSPLEX
The MVS sysplex name as found in the COUPLExx or LOADxx member of SYS1.PARMLIB.

SYMDEF, symbolic-name
Symbolic variables of your MVS system.

Note: For information about other system variables, see "SYSVAR" on page 148.

These items of information are described as:

SYSAPPCLU
The APPC/MVS logical unit (LU) name. The LU name identifies the TSO/E address space, where your REXX exec is running, as the SNA addressable unit for Advanced Program-to-Program Communications (APPC). The LU name is obtained by using the APPC/MVS Advanced TP Callable Services (ATBEXAI - Information Extract Service).

The LU name is returned as a character string. Trailing blanks are truncated. A null string is returned if:

- There is no APPC activity in the address space where the REXX exec is running.
- No LU name is provided by the APPC/MVS Advanced TP Callable Services.

SYSDFP
The level of DFSMSdfp installed on your system. The value that is returned is in the format *cc.vv.rr.mm*, where *cc* is a code, *vv* the version, *rr* the release number, and *mm* the modification level. All values are two-digit decimal numbers.

The *cc* code has one of these meanings:

00
MVS/XA DFP Version 2 or MVS/DFP Version 3 running with MVS/SP on MVS/XA or MVS.

01
DFSMSdfp in DFSMS running with MVS/SP on MVS or OS/390®.

02
DFSMSdfp in OS/390 Version 2 Release 10, or in z/OS Version 1 Release 1 or z/OS Version 1 Release 2. All three releases returned "02.02.10.00".

03
DFSMSdfp in z/OS Version 1 Release 3 or later.

Larger values for the *cc* code represent later products.

SYSMVS
The level of the base control program (BCP) component of z/OS.

The value that is returned is that of the CVTPRODN field in the communications vector table (CVT), for example `SP7.0.1`. Trailing blanks are removed.

The format of the value that is returned by SYSMVS might change in future, but remains the content of the CVTPRODN field.

OS/390 Users: To provide customers with the least disruptive change when changing from MVS SP 5.x to OS/390, the format of the CVTPRODN field is maintained and contains `SP5.3.0` for OS/390 Release 1. This is because some products test byte 3 to see if it is "5", which indicates that certain functions are available.

SYSNAME
The name of the system your REXX exec is running on, as specified in the SYSNAME statement in SYS1.PARMLIB member IEASYSxx.

The system name can be used in various ways:

- In a multi-system global resource serialization complex, the name identifies each system in the complex.
- The system also uses this value to uniquely identify the originating system in messages in the multiple console support (MCS) hardcopy log and in the display that is created by the DISPLAY R command.

- The value of SYSNAME is used as the name of the system log (SYSLOG).

SYSOPSYS

The z/OS name, version, release, modification level, and FMID. For example,

```
/* REXX */
mvsstring = MVSVAR('SYSOPSYS')
say mvsstring
exit 0
```

Might return a string of `z/OS 01.01.00 JBB7713`, where `z/OS` represents the product name, followed by a blank character, followed by an eight-character string representing version, release, modification number, followed by a blank character, followed by the FMID.

SYSOPSYS was introduced after TSO/E Version 2 Release 5 with APAR OW17844. If you use this variable in an environment earlier than TSO/E 2.5, or without the PTF associated with APAR OW17844, the system returns a null string.

SYSOSSEQ

This variable returns the value from the ECVTPSEQ field of the ECVT, which has the format nn vv rr mm (where nn = x'00' for OS/390, x'01' for z/OS). This value is referred to as the product sequence number. It can be used to determine whether the operating system is at a suitable level for a wanted function. Its value always increases from one release level to the next regardless of the character release identifier that is assigned to that release.

In the returned value, each 2 digit part of the ECVTPSEQ value is separated from the next by a period. For example, '01.01.13.00' is returned for z/OS V1R13.

SYSSECLAB

The security label (SECLABEL) name of the TSO/E session where the REXX exec was started. Trailing blanks are removed.

Note: The use of this argument requires that RACF is installed, and that security label checking is activated. If no security information is found, the function returns a null string.

SYSSMFID

Identification of the system on which System Management Facilities (SMF) is active. The value that is returned is as specified in SYS1.PARMLIB member SMFPRMxx on the SID statement. Trailing blanks are removed.

Note that the value returned by arguments SYSSMFID and SYSNAME can be the same in your installation. See *z/OS MVS Initialization and Tuning Reference* for more details on the SYSNAME and SID statement in member SMFPRMxx.

SYSSMS

Indicator whether SMS (storage management subsystem) is running on the system. The function returns one of the following character strings:

UNAVAILABLE

Obsolete and should no longer occur. System logic error. Contact your IBM® service representative.

INACTIVE

SMS is available on your system but not active.

ACTIVE

SMS is available and active, so your REXX exec can depend on it.

The following three arguments are in support of a SYSPLEX configuration. They return information about the SYSPLEX as stored in various members of SYS1.PARMLIB. The returned values can be used, for example, to uniquely identify or build data sets or other resources belonging to a specific system within the SYSPLEX.

SYSCLONE

MVS system symbol representing its system name. It is a 1- to 2-byte shorthand notation for the system name. The value is obtained from SYS1.PARMLIB member IEASYMxx[4]. For example, if SYSCLONE(A1) is specified in IEASYMxx, then

```
MVSVAR('SYSCLONE')
```

Returns a value of A1. A null string is returned if no MVS SYSCLONE ID is specified in IEASYMxx.

SYSPLEX

The MVS sysplex name as found in the COUPLExx or LOADxx member of SYS1.PARMLIB. The returned value has a maximum of eight characters. Trailing blanks are removed. If no sysplex name is specified in SYS1.PARMLIB, the function returns a null string.

SYMDEF,symbolic-name

Returns the value that is associated with the symbolic-name defined in an IEASYSxx member of SYS1.PARMLIB on a SYSDEF ... SYMDEF statement.

Or, symbolic-name can also be one of the system static or dynamic symbols as defined in *z/OS MVS Initialization and Tuning Reference*. For example, if SYMDEF(&SYSTEMA = 'SA') is specified in IEASYMxx, then

```
     X = MVSVAR('SYMDEF','SYSTEMA')
```

Returns a value of SA. A null string is returned if the symbolic-name is not specified in IEASYMxx and is not one of the standard static or dynamic symbols that are defined by MVS.

You can also retrieve the value for one of the MVS defined static or dynamic systems symbols. For example:

```
     X = MVSVAR('SYMDEF','JOBNAME')  /*Returns JOBNAME
                                     BOB perhaps */
```

Refer to *z/OS MVS Initialization and Tuning Reference* for a discussion and list of the currently defined MVS static and dynamic symbols.

For example, you can retrieve the IPL Volume Serial Name of your system by using

```
     SAY MVSVAR('SYMDEF','SYMR1')   /*may return 640S06
                                      as IPL Vol. Ser. Name */
```

The MVSVAR('SYMDEF',string) function goes through REXX substitution for string first, the result of which should be a 1-16 character symbolic-name specifying the symbol that is defined in the SYMDEF statement. (String is evaluated by REXX before MVSVAR is invoked.) Also, note that string can be longer than the length of just the variable name, if you substring it. For example, if a symbol SYSR1 is defined as having a value of 'XSMY2', then MVSVAR('SYMDEF', 'SYSR1(2:4)') returns 'SMY', the substring value of SYSR1 from position 2 to 4.

Therefore, the current limit on length of symbolic-names is 16 and if a name longer than 16 characters is passed, it is undefined and results in a null value being returned. However, if a symbolic-name longer than 16 becomes valid in the future, that name might be validly passed to MVSVAR as the symbolic-name because MVSVAR does not impose a length limit on string, when it is <= 250 characters.

Any values other than the symbolic-name, or substring of that name, passed in string, including REXX delimiters might cause unpredictable or undefined results, and is not considered as part of the intended interface.

Examples:

1. This example shows how to retrieve the current operating system level.

[4] Introduced with MVS SP 5.2; provides a mechanism to assign system substitution symbols names and values.

```
opersys = MVSVAR('SYSOPSYS')
```

2. This example shows how to retrieve information about a SYSPLEX configuration.

   Assume that your installation defined, in member SYS1.PARMLIB(IEASYM11), certain variables that are applicable on a system-wide basis. Assume further that one of them starts with the string BOOK and is concatenated by the sysclone ID, for example

   ```
   SYMDEF(&BOOKA1='DIXI')
   ```

   You can obtain the value of this variable as follows.

   ```
   tempvar = 'BOOK'||MVSVAR('SYSCLONE') /* the result could be BOOKA1    */
                                        /* where A1 is obtained as the   */
                                        /* result of MVSVAR('SYSCLONE')  */

   instvar = MVSVAR('SYMDEF',tempvar)   /* the result could be DIXI if   */
                                  /*    in IEASYM11 the statement        */
                                  /*    SYMDEF(&DATASA1='DIXI') had been */
                                  /*included by the system administrator */
   ```

3. This example shows the use of long symbol names, or names ending in underscore.

   If the following symbols are defined with SYMDEF statements in the active IEASYMxx member of 'SYS1.PARMLIB', then

   LONG_SYMBOL_NAME
   The value is: "SY1T_ON_PLEX_A44T"
   EXTSYM_
   The value is: "<==THIS VALUE CAN BE UP TO 44 CHARS LONG===>"

   That is, IEASYMxx contains the definitions:

   ```
   SYMDEF(&LONG_SYMBOL_NAME='SY1T_ON_PLEX_A44T')
   SYMDEF(&EXTSYM_='<==THIS VALUE CAN BE UP TO 44 CHARS LONG===>')
   ```

   Then, the MVSVAR function can be used to retrieve the values of these symbols as shown:

   ```
   z1 = MVSVAR('SYMDEF','LONG_SYMBOL_NAME')
   z2 = MVSVAR('SYMDEF','EXTSYM_')
   ```

   For z1:

   ```
   <- Returns z1: SY1T_ON_PLEX_A44T
   ```

   For z2:

   ```
   <- Returns z2: <==THIS VALUE CAN BE UP TO 44 CHARS LONG===>
   ```

Checking for Prerequisite Program Level

Several of the MVSVAR arguments require a minimum prerequisite program level.

Running on an earlier release causes a syntax error that is accompanied by an error message. If you do not have SYNTAX trap that is enabled, the REXX exec ends. You might avoid termination of the REXX exec by testing for the proper program level as shown in the following examples.

Example 1: Testing for Proper MVS Level:

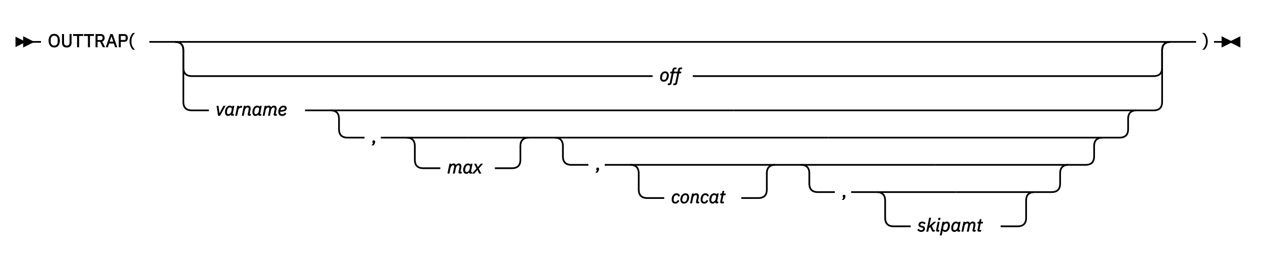
```
/*REXX*/
IF MVSVAR('SYSMVS') >= 'SP5.2.0' THEN
SAY MVSVAR('SYSCLONE')       /* yes, we can use the SYSCLONE argument */
EXIT 0
```

Example 2: Testing for Proper DFP Level:

```
/*REXX*/
IF MVSVAR('SYSDFP') >= '00.03.03.00' THEN
```

```
SAY MVSVAR('SYSSMS')          /* yes, we can use the SYSSMS argument   */
EXIT 0
```

OUTTRAP



OUTTRAP returns the name of the variable in which trapped output is stored, or if trapping is not in effect, OUTTRAP returns the word `off`.

You can use the following arguments to trap lines of command output into compound variables or a series of numbered variables, or to turn trapping off that was previously started.

off
: specify the word OFF to turn trapping off.

varname
: the stem of the compound variables or the variable prefix assigned to receive the command output. Compound variables contain a period and allow for indexing, but lists of variables with the same prefix cannot be accessed by an index in a loop.

  Note: Do not use "OFF" as a variable name.

max
: the maximum number of lines to trap. You can specify a number, an asterisk in quotation marks (‘*’), or a blank. If you specify ‘*’ or a blank, all the output is trapped. The default is 999,999,999. If the maximum number of lines are trapped, subsequent lines are not stored in variables.

concat
: indicates how output should be trapped. For *concat*, specify one of the following:

  - CONCAT

    indicates that output from commands be trapped in consecutive order until the maximum number of lines is reached. For example, if the first command has three lines of output, they are stored in variables ending in 1, 2, and 3. If the second command has two lines of output, they are stored in variables ending in 4 and 5. The default order for trapping is CONCAT.
  - NOCONCAT

    indicates that output from each command be trapped starting at the variable ending in 1. For example, if the first command has three lines of output, they are stored in variables ending in 1, 2, and 3. If another command has two lines of output, they replace the first command's output in variables 1 and 2.

skipamt
: specifies the number of lines to be skipped before trapping begins. If `skipamt` is not specified, the default is 0. The highest skip amount allowed is 999,999,999.

Lines of output are stored in successive variable names (as specified by *varname*) concatenated with integers starting with 1. All unused variables display their own names. The number of lines that were trapped is stored in the variable name followed by 0. For example, if you specify `cmdout.` as the *varname*, the number of lines stored is in:

```
cmdout.0
```

If you specify cmdout as the *varname*, the number of lines stored is in:

```
cmdout0
```

An exec can use these variables to display or process TSO/E command output. Error messages from TSO/E commands are trapped, but other types of error messages are sent to the terminal. Trapping, once begun, continues from one exec to other invoked execs or CLISTs. Trapping ends when the original exec ends or when trapping is turned off.

You can use the OUTTRAP function only in REXX execs that run using ADDRESS TSO.

Environment Customization Considerations: If you use IRXINIT to initialize language processor environments, note that you can use OUTTRAP only in environments that are integrated into TSO/E (see "Types of environments - integrated and not integrated into TSO/E" on page 316).

OUTTRAP traps output from commands, including those written in REXX. A command written in REXX cannot turn output trapping off on behalf of its invoker. Output trapping should be turned on and off at the same exec level. Therefore, a command written in REXX should only turn output trapping off if that command turned it on. In the following examples, the first illustrates *correct* usage of OUTTRAP; the second *incorrect* usage. Note that the placement of the y = OUTTRAP('OFF') statement must be within the REXX1 exec, not the REXX2 command.

- **Correct** usage of OUTTRAP

```
x = OUTTRAP('VAR.')
"%REXX2"
y = OUTTRAP('OFF')
EXIT

/* REXX2 command */
SAY "This is output from the REXX2 command " /* This will be trapped */
RETURN
```

- **Incorrect** usage of OUTTRAP

```
/* REXX1 */
x = OUTTRAP('VAR.')
"%REXX2"
EXIT

/* REXX2 command */
SAY "This is output from the REXX2 command " /* This will be trapped */
y = OUTTRAP('OFF')
RETURN
```

To trap the output of TSO/E commands under ISPF, you must invoke an exec with command output **after** ISPF or one of its services has been invoked.

The output of authorized commands listed under the AUTHCMD parameter in the active IKJTSOxx parmlib member cannot be trapped by a REXX exec invoked under any application that builds its own ECT. For example, a REXX exec must be prefixed by the TSO subcommand of IPCS to trap the output of authorized commands when invoked from IPCS under ISPF.

Regardless of the user's region size, the variables used to trap output from authorized commands and programs are limited to storage below the 16MB line, unless the PROFILE setting of VARSTORAGE(HIGH) is in use. If VARSTORAGE(HIGH) is specified in the user's PROFILE setting, using storage above the 16MB line is possible, and REXX can trap more lines of output from an authorized command or program.

OUTTRAP may not trap all of the output from a TSO/E command. The output that the OUTTRAP function traps depends on the type of output that the command produces. For example, the TSO/E command OUTPUT PRINT(*) directs the output from a job to your terminal. The OUTTRAP external function traps messages from the OUTPUT PRINT(*) command, but does not trap the job output itself that is directed to the terminal.

In general, the OUTTRAP function traps all output from a TSO/E command. For example, OUTTRAP traps broadcast messages from LISTBC, the list of allocated data sets from LISTALC, catalog entries from LISTCAT, and so on.

If you plan to write your own command processors for use in REXX execs, and you plan to use the OUTTRAP external function to trap command output, note the OUTTRAP function does not trap command output that is sent to the terminal by:

- TPUT
- WTO macro
- messages issued by TSO/E REXX (that is, messages beginning with IRX)

  Note: The TRAPMSG can be used to override this setting to allow IRX messages issued by TSO/E REXX or by a REXX host command like EXECIO to be trapped.
- messages issued by TRACE output.

  Note: Messages issued by TRACE output can be trapped if the exec being traced has been invoked by a higher level exec that uses OUTTRAP to enable output trapping and then invokes the exec to be traced.

However, OUTTRAP does trap output from the PUTLINE macro with DATA or INFOR keywords. Therefore, if you write any command processors, you may want to use the PUTLINE macro rather than the TPUT or WTO macros. *z/OS TSO/E Programming Guide* describes how to write a TSO/E command processor. For information about the PUTLINE macro, see *z/OS TSO/E Programming Services*.

Additional Variables That OUTTRAP Sets

In addition to the variables that store the lines of output, OUTTRAP stores information in the following variables:

varname0
: contains the largest index into which output was trapped. The number in this variable cannot be larger than *varname*MAX or *varname*TRAPPED.

varnameMAX
: contains the maximum number of output lines that can be trapped. That is, the total number of lines generated by commands while OUTPUT trapping is in effect. See example "1" on page 135.

varnameTRAPPED
: contains the total number of lines of command output. The number in this variable can be larger than *varname*0 or *varname*MAX.

varnameCON
: contains the status of the *concat* argument, which is either CONCAT or NOCONCAT.

varnameSKIPPED
: contains the number of output lines that have been skipped so far. This variable cannot be larger than varnameSKIPAMT.

varnameSKIPAMT
: contains the total number of output lines that must be skipped before the first output line is trapped in varname1.

  If a non-zero skip amount is specified, the number of lines that are skipped is still counted as part of the varnameTRAPPED value. Any lines of output ignored after the vanameMAX limit has been reached are still be counted as part of the varnameTRAPPED value.

For more information about trapping or suppressing output with the OUTTRAP function, see "OUTTRAP versus MSG function when trapping or suppressing output" on page 39.

Examples:

The following are some examples of using OUTTRAP.

1. This example shows the resulting values in variables after the following OUTTRAP function is processed.

```
x = OUTTRAP("ABC",4,"CONCAT")
```

```
 "Command1" /* sample command producing 3 lines of output */
```

Command 1 has three lines of output. After command 1 runs, the following variables are set.

```
ABC0            -->  3
ABC1            -->  command 1 output line 1
ABC2            -->  command 1 output line 2
ABC3            -->  command 1 output line 3
ABC4            -->  ABC4
ABCMAX          -->  4
ABCTRAPPED      -->  3
ABCCONCAT      --> CONCAT
ABCSKIPPED      --> 0
ABCSKIPAMT      --> 0
```

```
"Command2" /* sample command producing 2 lines of output */
```

Command 2 has two lines of output. After command 2 runs, the following variables are set. Note that the second line of output from command 2 is not trapped because the MAX (4) is reached when the first line of output from command 2 is produced.

```
ABC0            -->  4
ABC1            -->  command 1 output line 1
ABC2            -->  command 1 output line 2
ABC3            -->  command 1 output line 3
ABC4            -->  command 2 output line 1
ABCMAX          -->  4
ABCTRAPPED      -->  5
ABCCONCAT      --> CONCAT
ABCSKIPPED      --> 0
ABCSKIPAMT      --> 0
```

2. This example shows the resulting values in variables after the following OUTTRAP function is processed.

```
x = OUTTRAP("XYZ.",4,"NOCONCAT")
```

```
"Command1" /* sample command producing 3 lines of output */
```

Command 1 has three lines of output. After command1 runs, the following variables are set.

```
XYZ.0           -->  3
XYZ.1           -->  command 1 output line 1
XYZ.2           -->  command 1 output line 2
XYZ.3           -->  command 1 output line 3
XYZ.4           -->  XYZ.4
XYZ.MAX         -->  4
XYZ.TRAPPED     -->  3
XYZ.CONCAT     --> NOCONCAT
XYZ.SKIPPED     --> 0
XYZ.SKIPAMT     --> 0
```

```
"Command2" /* sample command producing 2 lines of output */
```

Command 2 has two lines of output. Because NOCONCAT was specified, output from Command 2 may replace output values trapped from Command 1. However, because only 2 lines were trapped by command 2, only 2 of the three lines trapped by Command 1 are replaced. XYZ.3 continues to contain the residual output from Command 1.

```
XYZ.0           -->  2
XYZ.1           -->  command 2 output line 1
XYZ.2           -->  command 2 output line 2
XYZ.3           -->  command 1 output line 3
XYZ.4           -->  XYZ.4
XYZ.MAX         -->  4
XYZ.TRAPPED     -->  2
XYZ.CONCAT     --> NOCONCAT
XYZ.SKIPPED     --> 0
XYZ.SKIPAMT     --> 0
```

3. To determine if trapping is in effect, enter the following:

   ```
   x = OUTTRAP()
   SAY x           /* If the exec is trapping output, displays the  */
                   /* variable name; if it is not trapping output, */
                   /* displays OFF */
   ```

4. To trap output from commands in consecutive order into the stem

   ```
   output.
   ```

   use one of the following:

   ```
   x = OUTTRAP("output.",'*',"CONCAT")

   x = OUTTRAP("output.")

   x = OUTTRAP("output.",,"CONCAT")
   ```

5. To trap 6 lines of output into the variable prefix `line` and not concatenate the output, enter the following:

   ```
   x = OUTTRAP(line,6,"NOCONCAT")
   ```

6. To suppress all command output, enter the following:

   ```
   x = OUTTRAP("output",0)
   ```

   Guideline: This form of OUTTRAP provides the best performance for suppressing command output.

7. Allocate a new data set like an existing one and if the allocation is successful, delete the existing data set. If the allocation is not successful, display the trapped output from the ALLOCATE command.

   ```
   x = OUTTRAP("var.")
   "ALLOC DA(new.data) LIKE(old.data) NEW"
   IF RC = 0 THEN
     "DELETE old.data"
   ELSE
     DO i = 1 TO var.0
        SAY var.i
     END
   ```

   If the ALLOCATE command is not successful, error messages are trapped in the following compound variables.

   - VAR.1 = *error message*
   - VAR.2 = *error message*
   - VAR.3 = *error message*

8. This example shows the resulting values in variables after the following OUTTRAP function is performed, using the SKIP argument of OUTTRAP to skip over lines of output before trapping begins.

   ```
   x = OUTTRAP("XYZ.",10,"CONCAT",50)
   ```

   ```
   "CMD1"             /* CMD1 produces 40 lines of output */
   ```

   CMD1 has 40 lines of output. Because 40 is less than 50, no lines of CMD1 are trapped. After CMD1, OUTTRAP variables have the following values:

   ```
   XYZ.0                   --> 0           No lines trapped yet
   XYZ.1                   --> XYZ.1
   XYZ.2                   --> XYZ.2
   XYZ.3                   --> XYZ.3
   XYZ.4                   --> XYZ.4
   XYZ.5                   --> XYZ.5
   XYZ.6                   --> XYZ.6
   ```

```
XYZ.7                   --> XYZ.7
XYZ.8                   --> XYZ.8
XYZ.9                   --> XYZ.9
XYZ.10                 --> XYZ.10
XYZ.MAX             --> 10
XYZ.TRAPPED    --> 40          40 lines of output handled so far
XYZ.CON             --> CONCAT
XYZ.SKIPPED     --> 40          Number of lines skipped so far
XYZ.SKIPAMT     --> 50          Total number to skip before trapping
```

```
"CMD2"             /* CMD2 produces 15 more lines of output */
```

CMD2 has 15 lines of output. 10 additional lines will be skipped, and trapping will begin with the 11th line (51st line overall).

```
XYZ.0                   --> 5           5 lines trapped
XYZ.1                   --> cmd2 output line 11 (output line 51 overall)
XYZ.2                   --> cmd2 output line 12 (output line 52 overall)
XYZ.3                   --> cmd2 output line 13 (output line 53 overall)
XYZ.4                   --> cmd2 output line 14 (output line 54 overall)
XYZ.5                   --> cmd2 output line 15 (output line 55 overall)
XYZ.6                   --> XYZ.6
XYZ.7                   --> XYZ.7
XYZ.8                   --> XYZ.8
XYZ.9                   --> XYZ.9
XYZ.MAX             --> 10
XYZ.10                 --> XYZ.10
XYZ.TRAPPED    --> 55          55 lines of output handled so far
XYZ.CON             --> CONCAT
XYZ.SKIPPED     --> 50          Number of lines skipped so far
XYZ.SKIPAMT     --> 50          Total number to skip before trapping
```

When skipping is in effect, the 1st line trapped into variable XYZ.1 is the (XYZ.SKIPAMT +1) line of output.

```
"CMD3"             /* CMD3 produces 10 more lines of output */
```

CMD3 has 10 lines of output. 5 additional lines will be trapped until MAX has been reached, and the remaining lines are discarded.

```
XYZ.0                   --> 10          10 lines trapped
XYZ.1                   --> cmd2 output line 11 (output line 51 overall)
XYZ.2                   --> cmd2 output line 12 (output line 52 overall)
XYZ.3                   --> cmd2 output line 13 (output line 53 overall)
XYZ.4                   --> cmd2 output line 14 (output line 54 overall)
XYZ.5                   --> cmd2 output line 15 (output line 55 overall)
XYZ.6                   --> cmd3 output line 1 (output line 56 overall)
XYZ.7                   --> cmd3 output line 2 (output line 57 overall)
XYZ.8                   --> cmd3 output line 3 (output line 58 overall)
XYZ.9                   --> cmd3 output line 4 (output line 59 overall)
XYZ.10                 --> cmd3 output line 5 (output line 60 overall)
XYZ.MAX             --> 10
XYZ.TRAPPED    --> 65          65 lines of output handled so far
XYZ.CON             --> CONCAT
XYZ.SKIPPED     --> 50          Number of lines skipped so far
XYZ.SKIPAMT     --> 50          Total number to skip before trapping
```

- The first line of output trapped is the XYZ.SKIPAMT+1 -th output line. The last line trapped is the XYZ.SKIPAMT+XYZ.MAX -th output line. Any output lines after this are not trapped.
- The first line trapped is always placed in the XYZ.1 variable.
- The highest indexed variable into which output can be trapped is XYZ.n (where n=XYZ.MAX).
- PUTLINE output lines that are skipped behave as if they were trapped and discarded. The output is not written to the screen, and the skipped lines are not actually trapped. However, each skipped line does count in the total number of output lines processed (varnameTRAPPED).

9. A user is about to use a REXX exec to invoke an authorized command "MYACMD". The exec uses OUTTRAP to trap and process the output. Although the command can produce 500,000 or more lines of output, the user is only interested in the last 70,000 lines and wants to capture these lines and write them to a data set to be processed later.

a. The user issues the PROFILE command without operands to see the current settings, and to ensures that the VARSTORAGE setting is HIGH.

```
PROFILE
IKJ56688I CHAR(0) LINE(0) PROMPT INTERCOM NOPAUSE MSGID NOMODE WTPMSG
NORECOVER PREFIX(TUSER01) PlANGUAGE(ENU) SLANGUAGE(ENU) VARSTORAGE(HIGH)
IKJ56689I DEFAULT LINE/CHARACTER DELETE CHARACTERS IN EFFECT FOR THIS
TERMINAL
```

If the VARSTORAGE setting is LOW he can change it to HIGH

```
PROFILE VARSTORAGE(HIGH)
```

b. The user then invokes the following exec:

```
/* REXX */
"ALLOC FI(OUTDD) DA(...my output data set...) SHR REUSE"
/* ... other processing ... */

/***********************************************************/
/* First determine how much output the command produces.   */
/* Specify a MAXAMT of 0 (2nd operand on OUTTRAP) to       */
/* indicate that we don't want to actually trap any output */
/* yet. We just want to determine the number of lines of   */
/* output produced.                                        */
/***********************************************************/
X = OUTTRAP(line.,0)
```

```
"MYACMD"
No_of_lines = line.trapped            /* Total number of lines of output */
/*********************************************************/
/* Throw away everything except the last 70,000 lines by */
/* setting SKIPAMT                                       */
/*********************************************************/
```

```
Skipamt = MAX(No_of_lines - 70000,0)
X = OUTTRAP (line.,'*','CONCAT',skipamt)
" MYACMD"
/****************************************************************/
/* Last 70,000 lines are trapped in line.1 to line.n where n is */
/* the line.0 value. If fewer than 70,000 lines of ouput        */
/* produced, then all of them are trapped.                      */
/****************************************************************/
```

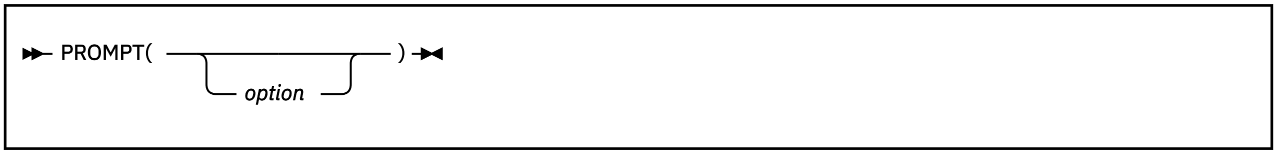
```
n = line.0                 /*number actually trapped  - expecting 70,000 */
```

```
" execio" n" diskw outdd (stem line. finis" /*Write the trapped lines */
```

```
If rc=0 then
Say n ' lines of data have been written to file OUTDD'
/* ... more processing ... */
```

```
" FREE FI(OUTDD)"
exit 0
```

PROMPT



PROMPT returns the value ON or OFF, which indicates the setting of prompting for the exec.

You can use the following options to set prompting on or off for interactive TSO/E commands, provided your profile allows for prompting. Only when your profile specifies PROMPT, can prompting be made available to TSO/E commands issued in an exec.

ON

returns the previous setting of prompt (ON or OFF) and sets prompting on for TSO/E commands issued within an exec.

OFF

returns the previous setting of prompt (ON or OFF) and sets prompting off for TSO/E commands issued within an exec.

Here are some examples:

```
promset = PROMPT()    -> 'OFF'  /* returns current setting (OFF)  */

setprom = PROMPT("ON")-> 'OFF'  /* returns previous setting (OFF)
                                   and sets prompting on          */
```

You can use the PROMPT function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations: If you use IRXINIT to initialize language processor environments, note that you can use PROMPT only in environments that are integrated into TSO/E (see "Interaction of Three Ways to Affect Prompting").

You can set prompting for an exec using the PROMPT keyword of the TSO/E EXEC command or the PROMPT function. The PROMPT function overrides the PROMPT keyword of the EXEC command. For more information about situations when one option overrides the other, see **Interaction of Three Ways to Affect Prompting**.

When an exec sets prompting on, prompting continues in other functions and subroutines called by the exec. Prompting ends when the PROMPT(OFF) function is used or when the original exec ends. When an exec invokes another exec or CLIST with the EXEC command, prompting in the new exec or CLIST depends on the setting in the profile and the use of the PROMPT keyword on the EXEC command.

If the data stack is not empty, commands that prompt retrieve information from the data stack before prompting a user at the terminal. To prevent a prompt from retrieving information from the data stack, issue a NEWSTACK command to create a new data stack for the exec.

When your TSO/E profile specifies NOPROMPT, no prompting is allowed in your terminal session even though the PROMPT function returns ON.

Interaction of Three Ways to Affect Prompting

You can control prompting within an exec in three ways:

1. TSO/E profile

   The TSO/E PROFILE command controls whether prompting is allowed for TSO/E commands in your terminal session. The PROMPT operand of the PROFILE command sets prompting on and the NOPROMPT operand sets prompting off.

2. TSO/E EXEC command

   When you invoke an exec with the EXEC command, you can specify the PROMPT operand to set prompting on for the TSO/E commands issued within the exec. The default is NOPROMPT.

3. PROMPT external function

   You can use the PROMPT function to set prompting on or off within an exec.

Table 8 on page 141 shows how the three ways to affect prompting interact and the final outcome of various interactions.

Table 8. Different ways prompting is affected

Interaction	Prompting	No prompting
PROFILE PROMPT EXEC PROMPT PROMPT(ON)	X	
PROFILE PROMPT EXEC NOPROMPT PROMPT(ON)	X	
PROFILE PROMPT EXEC NOPROMPT PROMPT()		X
PROFILE PROMPT EXEC NOPROMPT PROMPT(OFF)		X
PROFILE PROMPT EXEC PROMPT PROMPT()	X	
PROFILE PROMPT EXEC PROMPT PROMPT(OFF)		X
PROFILE NOPROMPT EXEC PROMPT PROMPT(ON)		X
PROFILE NOPROMPT EXEC NOPROMPT PROMPT(ON)		X
PROFILE NOPROMPT EXEC PROMPT PROMPT(OFF)		X
PROFILE NOPROMPT EXEC NOPROMPT PROMPT(OFF)		X
PROFILE NOPROMPT EXEC PROMPT PROMPT()		X
PROFILE NOPROMPT EXEC NOPROMPT PROMPT()		X

Examples:

The following are some examples of using PROMPT.

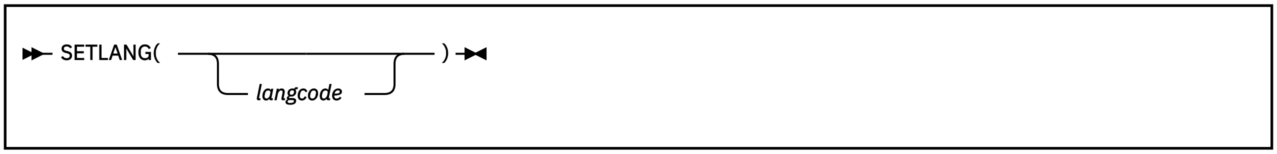
1. To check if prompting is available before issuing the interactive TRANSMIT command, use the PROMPT function as follows:

```
"PROFILE PROMPT"
IF PROMPT() = 'ON' THEN
  "TRANSMIT"
ELSE
  DO
    x = PROMPT('ON')  /* Save prompt setting and turn prompting ON */
    "TRANSMIT"        /* Restore prompting setting */
    y = PROMPT(x)
  END
```

2. Suppose you want to use the LISTDS command in an exec and want to ensure that prompting is done to the terminal. First check whether the data stack is empty. If the data stack is not empty, use the NEWSTACK command to create a new data stack. Use the PROMPT function before issuing the LISTDS command.

```
IF QUEUED() > 0 THEN
  "NEWSTACK"
ELSE NOP
x = PROMPT('ON')
"LISTDS"
y = PROMPT(x)
```

SETLANG



SETLANG returns a three character code that indicates the language in which REXX messages are currently being displayed. Table 9 on page 142 shows the language codes that replace the function call and the corresponding languages for each code.

You can optionally specify one of the language codes as an argument on the function to set the language in which REXX messages are displayed. In this case, SETLANG returns the code of the language in which messages are currently displayed and changes the language in which subsequent REXX messages will be displayed.

Table 9. Language codes for SETLANG function that replace the function call

Language code	Language
CHS	Simplified Chinese
CHT	Traditional Chinese
DAN	Danish
DEU	German
ENP	US English - all uppercase
ENU	US English - mixed case (upper and lowercase)
ESP	Spanish
FRA	French
JPN	Japanese
KOR	Korean

Table 9. Language codes for SETLANG function that replace the function call (continued)

Language code	Language
PTB	Brazilian Portuguese

Here are some examples::

```
curlang = SETLANG()     -> 'ENU'  /* returns current language (ENU)  */

oldlang = SETLANG("ENP")-> 'ENU'  /* returns current language (ENU)
                                     and sets language to US English
                                     uppercase (ENP)                 */
```

You can use the SETLANG function in an exec that runs in any MVS address space (TSO/E and non-TSO/E).

After an exec uses SETLANG to set a specific language, any REXX message the system issues is displayed in that language. If the exec calls another exec (either as a function or subroutine or using the TSO/E EXEC command), any REXX messages are displayed in the language you specified on the SETLANG function. The language specified on SETLANG is used as the language for displaying REXX messages until another SETLANG function is invoked or the environment in which the exec is running terminates.

Note:

1. The default language for REXX messages depends on the language feature that is installed on your system. The default language is in the language field of the parameters module (see "Characteristics of a Language Processor Environment" on page 317). You can use the SETLANG function to determine and set the language for REXX messages.
2. The language codes you can specify on the SETLANG function also depend on the language features that are installed on your system. If you specify a language code on the SETLANG function and the corresponding language feature is not installed on your system, SETLANG does not issue an error message. However, if the system needs to display a REXX message and cannot locate the message for the particular language you specified, the system issues an error message. The system then tries to display the REXX message in US English.
3. Your installation can customize TSO/E to display certain information at the terminal in different languages. Your installation can define a primary and secondary language for the display of information. The language codes for the primary and secondary languages are stored in the user profile table (UPT). You can use the TSO/E PROFILE command to change the languages specified in the UPT.

   The languages stored in the UPT do not affect the language in which REXX messages are displayed. The language for REXX messages is controlled only by the default in the language field of the parameters module and the SETLANG function.

   For information about customizing TSO/E for different languages and the types of information that are displayed in different languages, see *z/OS TSO/E Customization*.
4. The SYSVAR external function has the SYSPLANG and SYSSLANG arguments that return the user's primary and secondary language stored in the UPT. You can use the SYSVAR function to determine the setting of the user's primary and secondary language. You can then use the SETLANG function to set the language in which REXX messages are displayed to the same language as the primary or secondary language specified for the user. See "SYSVAR" on page 148 for more information.

Examples:

The following are some examples of using SETLANG:

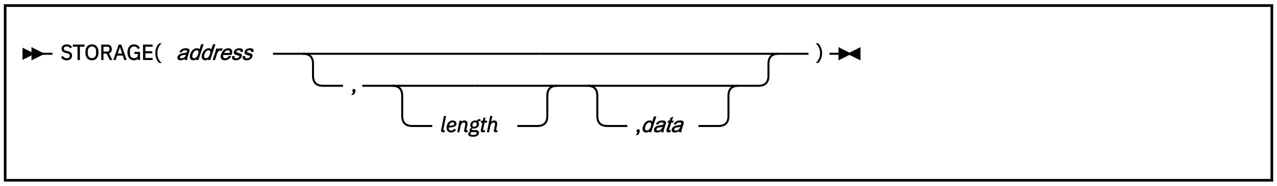
1. To check the language in which REXX messages are currently being displayed, use the SETLANG function as follows:

   ```
   currlng = SETLANG()     /* for example, returns ENU */
   ```
2. The SYSPLANG argument of the SYSVAR function returns the user's primary language that is stored in the user profile table (UPT).

The following example uses the SYSVAR function to determine the user's primary language and then uses the SETLANG function to check the language in which REXX messages are displayed. If the two languages are the same, no processing is performed. If the languages are different, the exec uses the SETLANG function to set the language for REXX messages to the same language as the user's primary language.

```
/*  REXX ...   */
⋮
proflang = SYSVAR('SYSPLANG')  /* check primary language in UPT     */
rexxlang = SETLANG()           /* check language for REXX messages  */
IF proflang ¬= rexxlang THEN
   newlang = SETLANG(proflang) /* set language for REXX messages    */
                               /* to user's primary language        */
ELSE NOP                       /* otherwise, no processing needed   */
⋮
EXIT
```

STORAGE



STORAGE returns *length* bytes of data from the specified *address* in storage. The *address* is a character string containing the hexadecimal representation of the storage address from which data is retrieved.

The address can be a 31-bit address, represented by 1 to 8 hexadecimal characters. The address can also be a 64-bit address represented by 9 to 17 characters which consists of 8 to 16 hexadecimal characters plus an optional underscore ("_") character separating the high order half and low order half of the 64-bit address. If an "_" is part of the 64-bit address, it must be followed by exactly 8 hexadecimal digits in the low order (or right) half of the 64-bit address.

Optionally, you can specify *length*, which is the decimal number of bytes to be retrieved from *address*. The default *length* is one byte. When *length* is 0, STORAGE returns a null character string.

If you specify *data*, STORAGE returns the information from *address* and then overwrites the storage starting at *address* with *data* you specified on the function call. The *data* is the character string to be stored at *address*. The *length* argument has no effect on how much storage is overwritten; the entire *data* is written.

If the REXX environment under which STORAGE is executing is configured to allow STORAGE to run in read-only mode, then the STORAGE function can be used to read but not alter storage. In this case, do not specify a *data* argument. If you do specify a new value in the third argument while executing in read-only mode, error message IRX0241I will be issued and the STORAGE function will end in error.

You can use the STORAGE function in REXX execs that run in any MVS address space (TSO/E and non-TSO/E).

If the STORAGE function tries to retrieve or change data beyond the storage limit, only the storage up to the limit is retrieved or changed.

Virtual storage addresses may be fetch protected, update protected, or may not be defined as valid addresses to the system. Any particular invocation of the STORAGE function may fail if it references a non-existent address, attempts to retrieve the contents of fetch protected storage, or attempts to update non-existent storage or is attempting to modify store protected storage. In all cases, a null string is returned to the REXX exec.

The STORAGE function returns a null string if any part of the request fails. Because the STORAGE function can both retrieve and update virtual storage at the same time, it is not evident whether the retrieve or update caused the null string to be returned. In addition, a request for retrieving or updating storage of a shorter length might have been successful. When part of a request fails, the failure point is on a decimal 4096 boundary.

Examples:

The following are some examples of using STORAGE:

1. To retrieve 25 bytes of data from address 000AAE35, use the STORAGE function as follows:

   ```
   storret = STORAGE(000AAE35,25)
   ```

2. To replace the data at address 0035D41F with ‘TSO/E REXX’, use the following STORAGE function:

   ```
   storrep = STORAGE(0035D41F,,'TSO/E REXX')
   ```

   This example first returns one byte of information found at address 0035D41F and then replaces the data beginning at address 0035D41F with the characters ‘TSO/E REXX’.

   Note: Information is retrieved before it is replaced.

3. Some areas may be accessible to be fetched but not written. That storage can be read as the actual hex data. You can then use the X2D function to then display that hex data in displaceable character format.

   ```
   say '<'C2X(STORAGE(10,4))'>'         /* Returns <00FDC248>, perhaps.  This
                                        area in PSA is update protected, but
                                        not fetch protected.  The CVT addr.*/
   ```

   Trying to update this same area will fail because address x'10' is a write protected area in PSA at PSA+x'10'.

   ```
   say '<'C2X<STORAGE(10,4,'XXXX'))'>'  /* Returns <> (a null string)
                                        because the storage at x'10' is at
                                        PSA+x'10' and is write protected and
                                        cannot be overwritten by STORAGE  */
   ```

4. STORAGE can access 31-bit storage (including 24-bit areas), as well as 64-bit storage. The following shows some possible STORAGE addresses, and the resulting binary addresses that is actually accessed by the STORAGE function.

Table 10. Valid and invalid 31-bit addresses

Hex Address passed to STORAGE	Binary Address used by STORAGE	Comment
10	'00000010'x	- Valid 31-bit.
00000010	'00000010'x	- Valid 31-bit.
80000010	'00000010'x	- Valid 31-bit. High order sign bit ignored in a 31-bit addr.
B2BC8DF8	'32BC8DF8'x	- Valid 31-bit addr. High-order sign bit ignored in a 31-bit addr.
200EF00	'0200EF00'x	- Valid 31-bit addr.

Table 11. Valid and invalid 64-bit addresses

Hex Address passed to STORAGE	Binary Address used by STORAGE	Comment
_00000010	'0000000000000010'x	- Valid 64-bit addr. (Padded to left with 0's to 64-bits.) Addresses same area as 31-bit '00000010'x addr.
0_00000010	'0000000000000010'x	- Valid 64-bit addr. Addresses same area as _00000010.

Table 11. Valid and invalid 64-bit addresses (continued)

Hex Address passed to STORAGE	Binary Address used by STORAGE	Comment
0_80000010	'0000000080000010'x	- Valid 64-bit addr. Addr is 2GB beyond the 0_00000010 addr..
000001EF10	'000000000001EF10'x	- Valid 64-bit addr.
1EF_80000010	'000001EF80000010'x	- Valid 64-bit addr.
1EF80000010	'000001EF80000010'x	- Valid 64-bit addr without "_" separator.
000001EF_80000010	'000001EF80000010'x	- Valid 64-bit addr.
000001EF_10	Invalid Addr	- Right half of 64-bit addr <8 chars.
00000001EF_000010	Invalid Addr	- Left hald of addr >8 chars, right half <8 chars.
0000001EF_000001000	Invalid Addr	- More than 16 hex chars.

Note: As an example of what you might expect, consider STORAGE used to retrieve 24 bytes from a 64-bit addressable area:

```
say '<'C2X(STORAGE(1ef_80000010,25))'>'  /* Returns  ...
                                         <IARST64 COMM SIZE 000512 > perhaps
*/
```

SYSCPUS

```
▸▸— SYSCPUS( cpus_stem ) —▸◂
```

SYSCPUS places, in a stem variable, information about those CPUs that are online.

The number of online CPUs is returned in variable `cpus_stem.0`. The serial numbers of each of those CPUs are returned in variables whose names are derived by appending a number (1 through `cpus_stem.0`) to the stem. Trailing blanks are removed. The serial number is obtained from the field PCCACPID in the MVS control block IHAPCCA. On a z990 machine or later, all CPU numbers are identical; therefore, SYSCPUS returns the same value for all CPUs.

The SYSCPUS function runs **in any MVS address space.**

Function Codes

The SYSCPUS function replaces the function call by the following function codes.

Table 12. SYSCPUS function codes

Function code	Description
0	SYSCPUS processing was successful.
4	SYSCPUs processing was successful. However, the function detected some inconsistency during processing, for example when the number of online CPUs varies or becomes zero during processing. This can happen when the operator changes an online CPU to offline while the function is in process. In this case, it may be advisable to repeat the function call.

Example:

Consider a system with two online CPUs. Their serial numbers are FF0000149221 and FF1000149221. Assuming you issue the following sequence of statements

```
/* REXX */
x = SYSCPUS('CPUS.')
say '0, if function performed okay: ' x
say 'Number of on-line CPUs is ' CPUS.0
do i = 1 to CPUS.0
  say 'CPU' i ' has CPU info ' CPUS.i
end
```

you get the following output:

```
0, if function performed okay: 0
Number of on-line CPUs is 2
CPU 1 has CPU info FF0000149221
CPU 2 has CPU info FF1000149221
                  /* ↑     ↑                      */
                  /* |     4 digits = model number   */
                  /* 6 digits       = CPU ID         */
```

SYSDSN

```
►►─ SYSDSN( dsname ) ─►◄
```

SYSDSN returns one of the following messages indicating whether the specified *dsname* exists and is available for use. The *dsname* can be the name of any cataloged data set or cataloged partitioned data set with a member name. Additionally, if you specify a member of a partitioned data set, SYSDSN checks to see if you have access to the data set.

You can use SYSDSN to obtain information about a data set that is available on DASD. SYSDSN does not directly support data that is on tape. SYSDSN supports generation data group (GDG) data sets when using absolute generation names, but does not support relative GDG names.

To suppress TSO/E messages issued by the SYSDSN function, use the MSG("OFF") function. For information about the MSG function, see "MSG" on page 127.

```
OK                                  /* data set or member is available */
MEMBER NOT FOUND
MEMBER SPECIFIED, BUT DATASET IS NOT PARTITIONED
DATASET NOT FOUND
ERROR PROCESSING REQUESTED DATASET /* This error can be presented if
                                      there is an error parsing the data
                                      set name provided, an error on the
                                      dynamic allocation (SVC 99) of the
                                      data set, or an error during OPEN
                                      processing.  Review the joblog and
                                      syslog for any error messages that
                                      may help identify what error was
                                      encountered */
PROTECTED DATASET                  /* a member was specified and RACF
                                      prevents this user from accessing
                                      this data set */
VOLUME NOT ON SYSTEM
INVALID DATASET NAME, dsname
MISSING DATASET NAME
UNAVAILABLE DATASET                /* another user has an exclusive ENQ
                                      on the specified data set */
```

After a data set is available for use, you may find it useful to get more detailed information. For example, if you later need to invoke a service that requires a specific data set organization, then use the LISTDSI function. For a description of the LISTDSI function, see "LISTDSI" on page 115.

You can use the SYSDSN function only in REXX execs that run in the TSO/E address space.

Environment Customization Considerations: If you use IRXINIT to initialize language processor environments, note that you can use SYSDSN only in environments that are integrated into TSO/E (see "Types of environments - integrated and not integrated into TSO/E" on page 316).

You can specify the *dsname* in any of the following ways:

- Fully-qualified data set name — The extra quotation marks prevent TSO/E from adding your prefix to the data set name.

  ```
  x = SYSDSN("'sys1.proj.new'")

  x = SYSDSN('''sys1.proj.new''')
  ```

- Non fully-qualified data set name that follows the naming conventions — When there is only one set of quotation marks or no quotation marks, TSO/E adds your prefix to the data set name.

  ```
  x = SYSDSN('myrexx.exec')

  x = SYSDSN(myrexx.exec)
  ```

- Variable name that represents a fully-qualified or non fully-qualified data set name — The variable name must not be enclosed in quotation marks because quotation marks prevent variable substitution.

  ```
  variable = "exec"
  x = SYSDSN(variable)       /* looks for 'userid.exec'      */
  y = SYSDSN('variable')     /* looks for 'userid.variable'  */
  z = SYSDSN("'"variable"'") /* looks for 'exec'             */
  ```

If the specified data set has been migrated, SYSDSN attempts to recall it.

Examples:

The following are some examples of using SYSDSN.

1. To determine the availability of prefix.PROJ.EXEC(MEM1):

   ```
   x = SYSDSN("proj.exec(mem1)")
   IF x = 'OK' THEN
     CALL routine1
   ELSE
     CALL routine2
   ```

2. To determine the availability of DEPT.REXX.EXEC:

   ```
   s = SYSDSN("'dept.rexx.exec'")
   say s
   ```

3. To suppress TSO/E messages:

   ```
   /*  REXX  */
   dsname = 'abcdefghij'
   y = MSG("OFF")
   x = SYSDSN(dsname)
   y = MSG(y)
   ```

SYSVAR

```
►► SYSVAR( arg_name ) ►◄
```

SYSVAR returns information about MVS, TSO/E, and the current session, such as levels of software available, your logon procedure, and your user ID. The information returned depends on the *arg_name* value specified on the function call. The *arg_name* values are divided into the following categories of information: user, terminal, exec, system, language, and console session information. The different categories are described below.

For information about system variables not being listed below, see "MVSVAR" on page 128.

User Information

Use the following arguments to obtain information related to the user.

SYSPREF
the prefix as defined in the user profile. The prefix is the string that is prefix to data set names that are not fully-qualified. The prefix is usually the user's user ID. You can use the TSO/E PROFILE command to change the prefix.

SYSPROC
the name of the logon procedure for the current session. You can use the SYSPROC argument to determine whether certain programs, such as the TSO/E Session Manager, are available to the user. For example, suppose your installation has the logon procedure SMPROC for the Session Manager. The exec can check that the user logged on using SMPROC before invoking a routine that uses Session Manager. Otherwise, the exec can display a message telling the user to log on using the SMPROC logon procedure.

SYSPROC returns the following values:

- When the REXX exec is invoked in the foreground (SYSVAR('SYSENV') returns 'FORE'), SYSVAR('SYSPROC') will return the name of the current LOGON procedure.
- When the REXX exec is invoked in batch (for example, from a job submitted by using the SUBMIT command), SYSVAR('SYSPROC') will return the value 'INIT', which is the ID for the initiator.
- When the REXX exec is invoked from a Started Task (for example, an address space that is started by using the Start operator command), SYSVAR('SYSPROC') will return the ID of the started task. If 'S procname' is issued from the operator console, the SYSVAR('SYSPROC') function will return the value 'procname'.

SYSUID
the user ID under which the current TSO/E session is logged on. The SYSUID argument returns the same value that the USERID built-in function returns in a TSO/E address space.

Terminal Information

Use the following arguments to obtain information related to the terminal.

SYSLTERM
number of lines available on the terminal screen. In the background, SYSLTERM returns 0.

SYSWTERM
width of the terminal screen. In the background, SYSWTERM returns 132.

Exec Information

Use the following arguments to obtain information related to the exec.

SYSENV
indicates whether the exec is running in the foreground or background. SYSENV returns the following values:

- FORE – exec is running in the foreground
- BACK – exec is running in the background

You can use the SYSENV argument to make logical decisions based on foreground or background processing.

SYSICMD
the name by which the user implicitly invoked the exec that is currently processing. If the user invoked the exec explicitly, SYSICMD returns a null value.

SYSISPF

indicates whether ISPF dialog manager services are available for the exec. SYSISPF returns the following values:

- ACTIVE – ISPF services are available. If the exec was invoked under ISPF by using the TSOEXEC interface, no ISPF services are available.
- NOT ACTIVE – ISPF services are not available

SYSNEST

indicates whether the exec was invoked from another program, such as an exec or CLIST. The invocation could be either implicit or explicit. SYSNEST returns YES if the exec was invoked from another program; otherwise, it returns NO.

SYSPCMD

the name or abbreviation of the TSO/E command processor that the exec most recently processed.

The *initial* value that SYSPCMD returns depends on how you invoked the exec. If you invoked the exec using the TSO/E EXEC command, the *initial* value returned is EXEC. If you invoked the exec using the EXEC subcommand of the TSO/E EDIT command, the *initial* value returned is EDIT.

You can use the SYSPCMD argument with the SYSSCMD argument for error and attention processing to determine where an error or attention interrupt occurred.

SYSSCMD

the name or abbreviation of the TSO/E subcommand processor that the exec most recently processed.

The *initial* value that SYSSCMD returns depends on how you invoked the exec. If you invoked the exec using the TSO/E EXEC command, the *initial* value returned is null. If you invoked the exec using the EXEC subcommand of the TSO/E EDIT command, the *initial* value returned is EXEC.

The SYSPCMD and SYSSCMD arguments are interdependent. After the initial invocation, the values that SYSPCMD and SYSSCMD return depend on the TSO/E command and subcommand processors that were most recently processed. For example, if SYSSCMD returns the value EQUATE, which is a subcommand unique to the TEST command, the value that SYSPCMD returns would be TEST.

You can use the SYSPCMD and SYSSCMD arguments for error and attention processing to determine where an error or attention interrupt occurred.

System Information

Use the following arguments to obtain information related to the system.

SYSCPU

the number of seconds of central processing unit (CPU) time used during the session in the form: *seconds.hundredths-of-seconds*.

You can use the SYSCPU argument and the SYSSRV argument, which returns the number of system resource manager (SRM) service units, to evaluate the:

- Performance of applications
- Duration of a session.

SYSHSM

indicates the level of the Data Facility Storage Management Subsystem Hierarchical Storage Manager (DFSMShsm). SYSHSM returns the following values:

- A character string of four two-digit decimal numbers separated by periods if DFSMShsm for OS/390 V2R10 or higher is active with PTF UW77424 applied. This represents the four-byte level in the MQCT_VRM field of the ARCQCT. For example, if DFSMShsm for OS/390 V2R10 is active with PTF UW77424 applied, SYSHSM returns: 02.02.10.00

  Note: For a description of the MQCT_VRM field in the ARCQCT control block, see OS/390 V2R10.

- A character string of four decimal digits if DFSMShsm before OS/390 V2R10 is installed and active. This represents the level of DFSMShsm. For example, if DFSMShsm 1.5.0 is installed and active, SYSHSM returns: 1050
- A null string if the Hierarchical Storage Manager is not active.

SYSJES
name and level of the JES installed on your system:

- A character string indicating name of the JES plus its version, release and modification level, for example

  ```
  JES2 OS 2.10
  ```

  where JES2 is the JES name and `OS 2.10` is the JES level. These two strings are separated by a blank character. If either the JES name or the level returns an empty character string, then no blank character is inserted. Trailing blanks are removed.
- -INACTIVE- (please note the delimiters) if the subsystem is not active.
- -DOWNLEVEL- (please note the delimiters) if the subsystem is neither JES2 SP4.3 or later, nor JES3 SP5.1.1 or later.

SYSLRACF
indicates the level of RACF installed on the system. If RACF is not installed, SYSLRACF contains a null value. The value of SYSLRACF is equal to the value in RCVTVRMN.

SYSNODE
network node name of your installation's JES. This name identifies the local JES in a network of systems or system complexes being used for network job entry (NJE) tasks. The name that is returned derives from the NODE initialization statement of JES.

The SYSNODE value is returned as either of the following:

- A character string indicating the node name, for example BOE9.
- -INACTIVE- (please note the delimiters) if the subsystem is not active.
- -DOWNLEVEL- (please note the delimiters) if the subsystem is neither JES2 SP4.3 or later, nor JES3 SP5.1.1 or later.

SYSRACF
indicates the status of RACF. SYSRACF returns the following values:

- AVAILABLE if RACF is installed and available
- NOT AVAILABLE if RACF is installed but is not available
- NOT INSTALLED if RACF is not installed.

SYSSRV
the number of system resource manager (SRM) service units used during the session.

You can use the SYSSRV argument and the SYSCPU argument, which returns the number of seconds of CPU time used, to evaluate the:

- Performance of applications
- Duration of a session.

SYSTERMID
the terminal ID of the terminal where the REXX exec was started.

- A character string indicating the terminal ID. Trailing blanks are removed.
- A null string if TSO runs in the background.

SYSTSOE
the level of TSO/E installed on the system. For OS/390 Version 2, Release 4 and later, SYSTSOE returns `2060`.

Language Information

Use the following arguments to obtain information related to the display of information in different languages.

SYSDTERM

indicates whether the user's terminal supports Double-Byte Character Set (DBCS). SYSDTERM returns the following values:

- YES – Terminal supports DBCS
- NO – Terminal does not support DBCS

The SYSDTERM argument is useful if you want to display messages or other information to the user and the information contains DBCS characters.

SYSKTERM

indicates whether the user's terminal supports Katakana. SYSKTERM returns the following values:

- YES – Terminal supports Katakana
- NO – Terminal does not support Katakana

The SYSKTERM argument is useful if you want to display messages or other information to the user and the information contains Katakana characters.

SYSPLANG

a three character code that indicates the user's primary language stored in the user profile table (UPT). For more information, see “Using the SYSPLANG and SYSSLANG arguments” on page 152.

SYSSLANG

a three character code that indicates the user's secondary language stored in the user profile table (UPT). For more information, see “Using the SYSPLANG and SYSSLANG arguments” on page 152.

Using the SYSPLANG and SYSSLANG arguments

Your installation can customize TSO/E to display certain types of information at the terminal in different languages. Your installation can define a primary and secondary language for the display of information. The language codes for the primary and secondary language are stored in the user profile table (UPT). You can use the TSO/E PROFILE command to change the languages specified in the UPT.

The SYSPLANG and SYSSLANG arguments return the three character language codes for the user's primary and secondary language that are stored in the UPT. The arguments are useful if you want to display messages or other information to the user in the primary or secondary language. The language codes that SYSVAR returns depend on the language support and codes that your installation has defined. *z/OS TSO/E Customization* describes how to customize TSO/E for different languages, the types of information that are displayed in different languages, and language codes.

TSO/E also provides the SETLANG external function that lets you determine and set the language in which REXX messages are displayed. SETLANG has no effect on the languages that are stored in the UPT. However, you can use both SETLANG and SYSVAR together for language processing. For example, you can use the SYSVAR function with the SYSPLANG or SYSSLANG argument to determine the language code stored in the UPT. You can then use the SETLANG function to set the language in which REXX messages are displayed to the same language as the user's primary or secondary language. See “SETLANG” on page 142 for more information.

Console Session Information

The console session arguments let you obtain information related to running an extended MCS console session that you have established using the TSO/E CONSOLE command.

The SOLDISP, UNSDISP, SOLNUM, and UNSNUM arguments provide information about the options that have been specified for a console session. The arguments relate to keywords on the TSO/E CONSPROF command. You can use the arguments to determine what options are in effect before you issue MVS system or subsystem commands or use the GETMSG function to retrieve a message.

The MFTIME, MFOSNM, MFJOB, and MFSNMJBX arguments provide information about messages that are issued during a console session. These arguments are useful if you use the GETMSG external function to retrieve messages that are not displayed at the terminal and you want to display a particular message that was retrieved. The arguments indicate whether certain types of information should be displayed with the message, such as the time stamp.

For information about console sessions, see Appendix C, "Writing REXX Execs to perform MVS operator activities," on page 449.

SOLDISP
indicates whether solicited messages that are routed to a user's console during a console session are to be displayed at the user's terminal. Solicited messages are responses from MVS system and subsystem commands that are issued during a console session. SOLDISP returns the following values:

- YES - solicited messages are displayed
- NO - solicited messages are not displayed

UNSDISP
indicates whether unsolicited messages that are routed to a user's console during a console session are to be displayed at the user's terminal. Unsolicited messages are messages that are not direct responses from MVS system and subsystem commands that are issued during a console session. UNSDISP returns the following values:

- YES - unsolicited messages are displayed
- NO - unsolicited messages are not displayed

SOLNUM
the size of the message table that contains solicited messages (that is, the number of solicited messages that can be stored). The system stores the messages in the table during a console session if you specify that solicited messages are not to be displayed at the terminal. You can use the TSO/E CONSPROF command to change the size of the table. For more information, see *z/OS TSO/E System Programming Command Reference*.

UNSNUM
the size of the message table that contains unsolicited messages (that is, the number of unsolicited messages that can be stored). The system stores the messages in the table during a console session if you specify that unsolicited messages are not to be displayed at the terminal. You can use the TSO/E CONSPROF command to change the size of the table. For more information, see *z/OS TSO/E System Programming Command Reference*.

MFTIME
indicates whether the user requested that the time stamp should be displayed with system messages. MFTIME returns the following values:

- YES – time stamp should be displayed
- NO – time stamp should not be displayed

MFOSNM
indicates whether the user requested that the originating system name should be displayed with system messages. MFOSNM returns the following values:

- YES – originating system name should be displayed
- NO – originating system name should not be displayed

MFJOB
indicates whether the user requested that the originating job name or job ID of the issuer should be displayed with system messages. MFJOB returns the following values:

- YES – originating job name should be displayed
- NO – originating job name should not be displayed

MFSNMJBX
indicates whether the user requested that the originating system name and job name should *not* be displayed with system messages. MFSNMJBX returns the following values:

- YES – originating system name and job name should *not* be displayed

- NO – originating system name and job name should be displayed

  MFSNMJBX is intended to override the values of MFOSNM and MFJOB. The value for MFSNMJBX may not be consistent with the values for MFOSNM and MFJOB.

You can use the SYSVAR function only in REXX execs that run in the TSO/E address space. Use SYSVAR to determine various characteristics to perform different processing within the exec.

Environment Customization Considerations: If you use IRXINIT to initialize language processor environments, note that you can use SYSVAR only in environments that are integrated into TSO/E (see "Types of environments - integrated and not integrated into TSO/E" on page 316).

Examples:

The following are some examples of using SYSVAR.

1. To display whether the exec is running in the foreground or background:

   ```
   SAY SYSVAR("sysenv")         /* Displays FORE or BACK */
   ```

2. To find out the level of RACF installed:

   ```
   level = SYSVAR("syslracf")   /* Returns RACF level */
   ```

3. To determine if the prefix is the same as the user ID:

   ```
   IF SYSVAR("syspref") = SYSVAR("sysuid") THEN
   ⋮
   ELSE
   ⋮
   EXIT
   ```

4. Suppose you want to use the GETMSG external function to retrieve a solicited message. Before using GETMSG, you want to save the current setting of message displaying and use the TSO/E CONSPROF command so that solicited messages are not displayed. After GETMSG processing, you want to restore the previous setting of message displaying.

   ```
   /*  REXX program ...   */
   ⋮
   mdisp = SYSVAR("SOLDISP")                 /* Save current message setting */
   "CONSPROF SOLDISPLAY(NO)"                 /* Inhibit message display      */
   ⋮
   msg = GETMSG('cons','sol','APP0096',,60) /* Retrieve message             */
   ⋮
   "CONSPROF SOLDISPLAY("mdisp")"           /* Restore message setting      */
   ⋮
   EXIT
   ```

5. This example shows how to retrieve the current JES node name:

   ```
   nodenam = SYSVAR('SYSNODE')
   ```

Relationship of CLIST Control Variables and SYSVAR Function

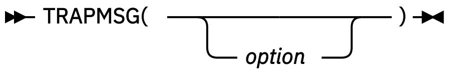
The information that the SYSVAR external function returns is similar to the information stored in CLIST control variables for TSO/E CLISTs. The SYSVAR external function does not support all the CLIST control variables. SYSVAR supports only the *arg_name* values described in this topic.

Some CLIST control variables do not apply to REXX. Other CLIST control variables duplicate other REXX functions. SYSVAR does not support the following CLIST control variables. However, for these CLIST control variables, there is an equivalent function in REXX, which is listed below.

```
SYSDATE   ===>  DATE(usa)
SYSJDATE  ===>  DATE(julian)
SYSSDATE  ===>  DATE(ordered)
```

```
SYSSTIME  ===>  SUBSTR(TIME(normal),1,5)
SYSTIME   ===>  TIME(normal) or TIME()
```

TRAPMSG



TRAPMSG returns the value ON or OFF, which indicates whether REXX messages (that is, messages of the form IRX.....) or CLIST error messages from CLISTs invoked by REXX are permitted to be trapped by the OUTTRAP function. If TRAPMSG is invoked with a null operand, the current TRAPMSG setting is returned, and the TRAPMSG setting is left unchanged.

With TRAPMSG(OFF), the default, any REXX messages issued by execs or host commands invoked by a REXX exec are always issued to the screen or to the output stream, even if OUTTRAP is active in the exec. TRAPMSG(ON) allows this to be changed so that such messages can be trapped. The valid values for option are:

OFF

Sets the current TRAPMSG status to OFF and returns the previous status. The default OFF indicates that REXX messages (IRX.....) and CLIST error messages issued by an REXX exec, CLIST or host command invoked by REXX is not trapped in the currently active OUTTRAP variable, if any. They go to the output stream or terminal.

ON

Sets the current TRAPMSG status to ON and returns the previous status. ON indicates that REXX messages (IRX.....) and CLIST error messages issued by any REXX exec, CLIST or host command invoked by REXX is trapped in the currently active OUTTRAP variable, if any. The exec can then examine the trapped error message.

Using TRAPMSG you can tell REXX that, when OUTTRAP is also active, that any REXX messages or CLIST error messages that would have otherwise been ignored by OUTTRAP and always written to the screen or to the output stream are now permitted to be trapped in the exec's active OUTTRAP variable.

Environment Customization Considerations: If you use IRXINIT to initialize language processor environments, note that you can use TRAPMSG only in environments that are integrated into TSO/E (see “Types of environments - integrated and not integrated into TSO/E” on page 316).

TRAPMSG(ON) can be used to enable your exec to capture, in an OUTTRAP variable, messages issued by a REXX exec, CLIST, or host command invoked from REXX, if those messages are issued as PUTLINE messages with the INFOR or DATA operands. Refer to the OUTTRAP function for a description of the kinds of output that can or cannot be trapped by OUTTRAP.

Examples:

Here are some examples:

1. A REXX exec invokes execio without allocating the indd file. EXECIO returns RC=20 and an error message. By trapping the message with OUTTRAP, the exec can decide what to do with the error.

   This same technique can be used to trap the IRX0250E message if execio were to take an abend, like a space B37 abend.

```
msgtrapstat = TRAPMSG('ON')      /* Save current status and set
                                 TRAPMSG ON to allow REXX msgs to
                                 be trapped                      */
outtrap_stat = OUTTRAP('err.')    /* Enable outtrap             */

/*****************************************************************/
/* Invoke the TSO host command, execio, and trap any error msgs  */
/* it might issue stem err.                                               */
/*****************************************************************/
"execio 1 diskr indd (stem rec. finis"

if RC = 20 then             /* If execio error occurred          */
  do i=1 to err.0
    say '==> ' err.i        /* Write any error msgs              */
  end
else
  do
  /* process execio record read, if any */
  end
outtrap_stat = OUTTRAP('OFF')   /* Disable outtrap               */
msgtrapstat = TRAPMSG('OFF')    /* Turn TRAPMSG off              */
```

2. A REXX exec turns on OUTTRAP and TRAPMSG and invokes a second REXX exec. The second REXX exec gets an IRX0040I message due to an invalid function call. Exec1 is able to trap the message issued from exec2.

   Note that if exec1 had made the bad TIME function call, it could not have trapped the error message because a function message is considered at the same level as the exec. This is similar to the fact that an exec can use OUTTRAP to trap SAY statement output from an exec that it invokes, but it cannot trap its own SAY output.

```
/* REXX - exec1 */
SIGNAL ON SYNTAX                   /* Trap syntax errors          */

trapit = OUTTRAP('line.')
trapmsg_stat = TRAPMSG('ON')

continue = 'continu1'               /* Set Retry label             */
call exec2

continu1:                          /* Retry label                 */
do i=1 to line.0   /* Display any output trapped fronm exec2      */
  if i=1 then      say '---------- start of trapped output ---------'
  say '==> ' line.i
  if i=line.0 then say '---------- end   of trapped output ---------'
end

trapit = OUTTRAP('OFF')
trapmsg_stat = TRAPMSG('OFF')
exit 0

SYNTAX:                     /* Syntax trap routine                */
 SIGNAL ON SYNTAX           /* Reenable syntax trap               */
 Interpret SIGNAL continue  /* Continue in mainline at retry label*/



/* REXX - exec2 */
say 'In exec2 ...'

time = TIME('P')     /* Invalid time operand, get msg IRX0040I     */

return time
```

Chapter 5. Parsing

The parsing instructions are ARG, PARSE, and PULL (see "ARG" on page 42, "PARSE" on page 57, and "PULL" on page 62).

The data to parse is a *source string*. Parsing splits up the data in a source string and assigns pieces of it into the variables named in a template. A *template* is a model specifying how to split the source string. The simplest kind of template consists of only a list of variable names. Here is an example:

```
variable1 variable2 variable3
```

This kind of template parses the source string into blank-delimited words. More complicated templates contain patterns in addition to variable names.

String patterns
Match characters in the source string to specify where to split it. (See "Templates containing string patterns" on page 159 for details.)

Positional patterns
Indicate the character positions at which to split the source string. (See "Templates containing positional (numeric) patterns" on page 159 for details.)

Parsing is essentially a two-step process.

1. Parse the source string into appropriate substrings using patterns.
2. Parse each substring into words.

Simple templates for parsing into words

Here is a parsing instruction:

```
parse value 'time and tide' with var1 var2 var3
```

The template in this instruction is: `var1 var2 var3`. The data to parse is between the keywords PARSE VALUE and the keyword WITH, the source string `time and tide`. Parsing divides the source string into blank-delimited words and assigns them to the variables named in the template as follows:

```
var1='time'
var2='and'
var3='tide'
```

In this example, the source string to parse is a literal string, `time and tide`. In the next example, the source string is a variable.

```
/* PARSE VALUE using a variable as the source string to parse    */
string='time and tide'
parse value string with var1 var2 var3           /* same results */
```

(PARSE VALUE does not convert lowercase a–z in the source string to uppercase A–Z. If you want to convert characters to uppercase, use PARSE UPPER VALUE. See "Using UPPER" on page 163 for a summary of the effect of parsing instructions on case.)

All of the parsing instructions assign the parts of a source string into the variables named in a template. There are various parsing instructions because of differences in the nature or origin of source strings. (See "Parsing instructions summary" on page 164 for a summary of all the parsing instructions .)

The PARSE VAR instruction is similar to PARSE VALUE except that the source string to parse is always a variable. In PARSE VAR, the name of the variable containing the source string follows the keywords PARSE

VAR. In the next example, the variable `stars` contains the source string. The template is `star1 star2 star3`.

```
/* PARSE VAR example                                             */
stars='Sirius Polaris Rigil'
parse var stars star1 star2 star3             /* star1='Sirius'  */
                                              /* star2='Polaris' */
                                              /* star3='Rigil'   */
```

All variables in a template receive new values. If there are *more variables in the template than words in the source string*, the leftover variables receive null (empty) values. This is true for all parsing: for parsing into words with simple templates and for parsing with templates containing patterns. Here is an example using parsing into words.

```
/* More variables in template than (words in) the source string  */
satellite='moon'
parse var satellite Earth Mercury               /* Earth='moon'  */
                                                /* Mercury=''    */
```

If there are *more words in the source string than variables in the template*, the last variable in the template receives all leftover data. Here is an example:

```
/* More (words in the) source string than variables in template  */
satellites='moon Io Europa Callisto...'
parse var satellites Earth Jupiter              /* Earth='moon'  */
                               /* Jupiter='Io Europa Callisto...'*/
```

Parsing into words removes leading and trailing blanks from each word before it is assigned to a variable. The exception to this is the word or group of words assigned to the last variable. The last variable in a template receives leftover data, *preserving extra leading and trailing blanks*. Here is an example:

```
/* Preserving extra blanks                                       */
solar5='Mercury Venus  Earth   Mars     Jupiter  '
parse var solar5 var1 var2 var3 var4
/* var1  ='Mercury'                                              */
/* var2  ='Venus'                                                */
/* var3  ='Earth'                                                */
/* var4  ='  Mars     Jupiter  '                                 */
```

In the source string, `Earth` has two leading blanks. Parsing removes both of them (the word-separator blank and the extra blank) before assigning `var3='Earth'`. `Mars` has three leading blanks. Parsing removes one word-separator blank and keeps the other two leading blanks. It also keeps all five blanks between `Mars` and `Jupiter` and both trailing blanks after `Jupiter`.

Parsing removes *no* blanks if the template contains only one variable. For example:

```
parse value '   Pluto   ' with var1        /* var1='   Pluto   '*/
```

The period as a placeholder

A period in a template is a placeholder. It is used instead of a variable name, but it receives no data. It is useful:

- As a “dummy variable” in a list of variables
- Or to collect unwanted information at the end of a string.

The period in the first example is a placeholder. Be sure to separate adjacent periods with spaces; otherwise, an error results.

```
/* Period as a placeholder                                       */
stars='Arcturus Betelgeuse Sirius Rigil'
parse var stars . . brightest .            /* brightest='Sirius' */

/* Alternative to period as placeholder                          */
stars='Arcturus Betelgeuse Sirius Rigil'
parse var stars drop junk brightest rest   /* brightest='Sirius' */
```

A placeholder saves the overhead of unneeded variables.

Templates containing string patterns

A *string pattern* matches characters in the source string to indicate where to split it. A string pattern can be a:

Literal string pattern
One or more characters within quotation marks.

Variable string pattern
A variable within parentheses with no plus (+) or minus (-) or equal sign (=) before the left parenthesis. (See "Parsing with variable patterns" on page 163 for details.)

Here are two templates: a simple template and a template containing a literal string pattern:

```
var1 var2          /* simple template                          */
var1 ', ' var2     /* template with literal string pattern       */
```

The literal string pattern is: `', '`. This template:

- Puts characters from the start of the source string up to (but not including) the first character of the match (the comma) into `var1`
- Puts characters starting with the character after the last character of the match (the character after the blank that follows the comma) and ending with the end of the string into `var2`.

A template with a string pattern can omit some of the data in a source string when assigning data into variables. The next two examples contrast simple templates with templates containing literal string patterns.

```
/* Simple template                                              */
name='Smith, John'
parse var name ln fn                     /* Assigns: ln='Smith,' */
                                         /*          fn='John'   */
```

Notice that the comma remains (the variable `ln` contains `'Smith,'`). In the next example the template is `ln ', ' fn`. This removes the comma.

```
/* Template with literal string pattern                         */
name='Smith, John'
parse var name ln ', ' fn                /* Assigns: ln='Smith'  */
                                         /*          fn='John'   */
```

First, the language processor scans the source string for ', '. It splits the source string at that point. The variable `ln` receives data starting with the first character of the source string and ending with the last character before the match. The variable `fn` receives data starting with the first character *after* the match and ending with the end of string.

A template with a string pattern omits data in the source string that matches the pattern. (See "Combining string and positional patterns: a special case" on page 166 in which a template with a string pattern does *not* omit matching data in the source string.) We used the pattern `', '` (with a blank) instead of `','` (no blank) because, without the blank in the pattern, the variable `fn` receives ' John' (including a blank).

If the source string *does not contain a match for a string pattern*, then any variables preceding the unmatched string pattern get all the data in question. Any variables after that pattern receive the null string.

A null string is never found. It always matches the end of the source string.

Templates containing positional (numeric) patterns

A *positional pattern* is a number that identifies the character position at which to split data in the source string. The number must be a whole number.

An *absolute positional pattern* is

- A number with no plus (+) or minus (-) sign preceding it or with an equal sign (=) preceding it
- A variable in parentheses with an equal sign before the left parenthesis. (See "Parsing with variable patterns" on page 163 for details on *variable positional patterns*.)

The number specifies the absolute character position at which to split the source string.

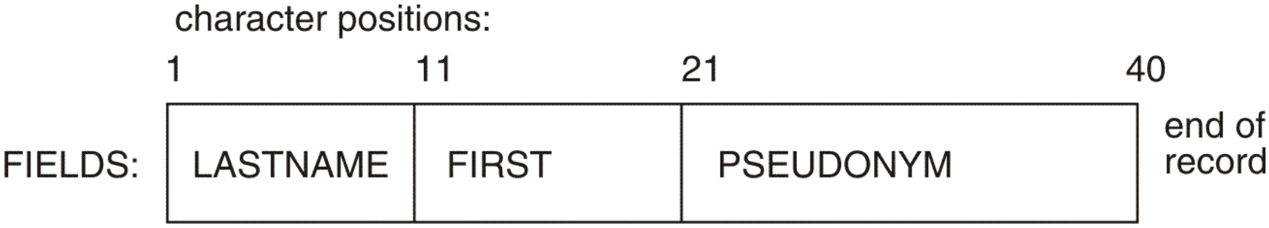
Here is a template with absolute positional patterns:

```
variable1 11 variable2 21 variable3
```

The numbers 11 and 21 are absolute positional patterns. The number 11 refers to the 11th position in the input string, 21 to the 21st position. This template:

- Puts characters 1 through 10 of the source string into `variable1`
- Puts characters 11 through 20 into `variable2`
- Puts characters 21 to the end into `variable3`.

Positional patterns are probably most useful for working with a file of records, such as:



The following example uses this record structure.

```
/* Parsing with absolute positional patterns in template        */
record.1='Clemens   Samuel    Mark Twain          '
record.2='Evans     Mary Ann  George Eliot        '
record.3='Munro     H.H.      Saki                '
do n=1 to 3
  parse var record.n lastname 11 firstname 21 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end                         /* Says 'By George!' after record 2  */
```

The source string is first split at character position 11 and at position 21. The language processor assigns characters 1 to 10 into `lastname`, characters 11 to 20 into `firstname`, and characters 21 to 40 into `pseudonym`.

The template could have been:

```
1 lastname 11 firstname 21 pseudonym
```

instead of

```
  lastname 11 firstname 21 pseudonym
```

Specifying the 1 is optional.

Optionally, you can put an equal sign before a number in a template. An equal sign is the same as no sign before a number in a template. The number refers to a particular character position in the source string. These two templates work the same:

```
lastname  11 first  21 pseudonym

lastname =11 first =21 pseudonym
```

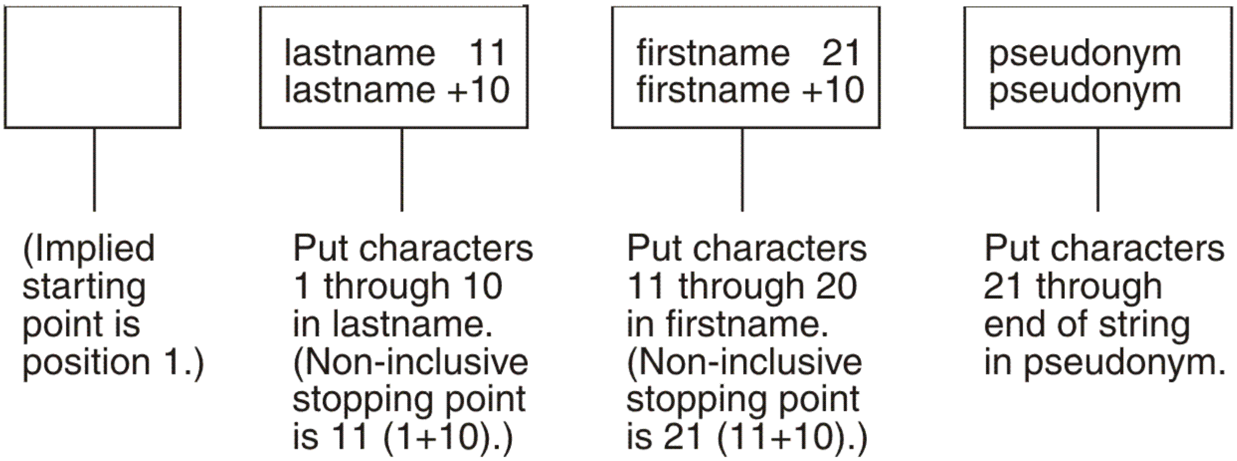
A *relative positional pattern* is a number with a plus (+) or minus (-) sign preceding it. (It can also be a variable within parentheses, with a plus (+) or minus (-) sign preceding the left parenthesis; for details see "Parsing with variable patterns" on page 163.)

The number specifies the relative character position at which to split the source string. The plus or minus indicates movement right or left, respectively, from the start of the string (for the first pattern) or from the position of the last match. The position of the last match is the first character of the last match. Here is the same example as for absolute positional patterns done with relative positional patterns:

```
/* Parsing with relative positional patterns in template         */
record.1='Clemens   Samuel    Mark Twain          '
record.2='Evans     Mary Ann  George Eliot        '
record.3='Munro     H.H.      Saki                '
do n=1 to 3
  parse var record.n lastname +10 firstname + 10 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end                                              /* same results  */
```

Blanks between the sign and the number are insignificant. Therefore, +10 and + 10 have the same meaning. Note that +0 is a valid relative positional pattern.

Absolute and relative positional patterns are interchangeable (except in the special case (see "Combining string and positional patterns: a special case" on page 166) when a string pattern precedes a variable name and a positional pattern follows the variable name). The templates from the examples of absolute and relative positional patterns give the same results.


lastname 11
lastname +10
firstname 21
firstname +10
pseudonym
pseudonym
(Implied starting point is position 1.)
Put characters 1 through 10 in lastname. (Non-inclusive stopping point is 11 (1+10).)
Put characters 11 through 20 in firstname. (Non-inclusive stopping point is 21 (11+10).)
Put characters 21 through end of string in pseudonym.

Only with positional patterns can a matching operation *back up* to an earlier position in the source string. Here is an example using absolute positional patterns:

```
/* Backing up to an earlier position (with absolute positional)  */
string='astronomers'
parse var string 2 var1 4 1 var2 2 4 var3 5 11 var4
say string 'study' var1||var2||var3||var4
/* Displays: "astronomers study stars"                           */
```

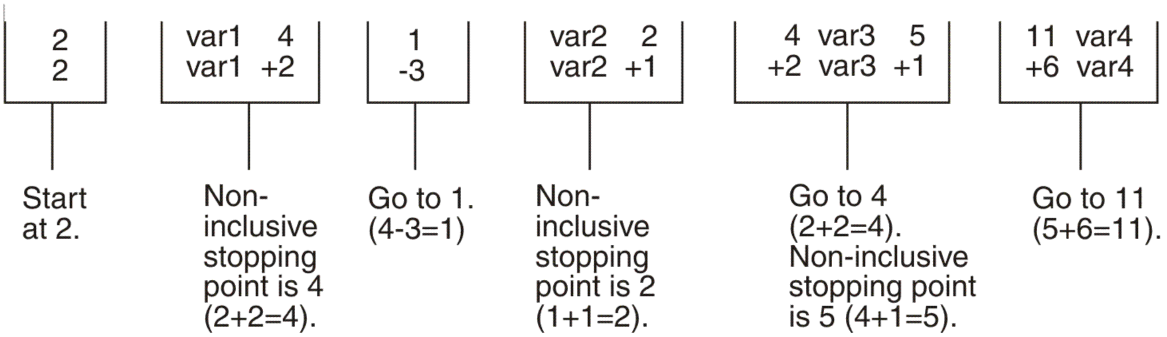
The absolute positional pattern 1 backs up to the first character in the source string.

With relative positional patterns, a number preceded by a minus sign backs up to an earlier position. Here is the same example using relative positional patterns:

```
/* Backing up to an earlier position (with relative positional)  */
string='astronomers'
parse var string 2 var1 +2 -3 var2 +1 +2 var3 +1 +6 var4
say string 'study' var1||var2||var3||var4      /* same results   */
```

In the previous example, the relative positional pattern -3 backs up to the first character in the source string.

The templates in the last two examples are equivalent.


2
2
Start at 2.
var1 4
var1 +2
Non-inclusive stopping point is 4 (2+2=4).
1
-3
Go to 1. (4-3=1)
var2 2
var2 +1
Non-inclusive stopping point is 2 (1+1=2).
4 var3 5
+2 var3 +1
Go to 4 (2+2=4). Non-inclusive stopping point is 5 (4+1=5).
11 var4
+6 var4
Go to 11 (5+6=11).

You can use templates with positional patterns to make multiple assignments:

```
/* Making multiple assignments                                    */
books='Silas Marner, Felix Holt, Daniel Deronda, Middlemarch'
parse var books 1 Eliot 1 Evans
/* Assigns the (entire) value of books to Eliot and to Evans.     */
```

Combining patterns and parsing into words

What happens when a template contains patterns that divide the source string into sections containing multiple words? String and positional patterns divide the source string into substrings. The language processor then applies a section of the template to each substring, following the rules for parsing into words.

```
/* Combining string pattern and parsing into words                */
name='    John      Q.   Public'
parse var name fn init '.' ln         /* Assigns: fn='John'       */
                                      /*          init='     Q'   */
                                      /*          ln='   Public'  */
```

The pattern divides the template into two sections:

- `fn init`
- `ln`

The matching pattern splits the source string into two substrings:

- `' John Q'`
- `' Public'`

The language processor parses these substrings into words based on the appropriate template section.

`John` had three leading blanks. All are removed because parsing into words removes leading and trailing blanks except from the last variable.

`Q` has six leading blanks. Parsing removes one word-separator blank and keeps the rest because `init` is the last variable in that section of the template.

For the substring `'  Public'`, parsing assigns the entire string into `ln` without removing any blanks. This is because `ln` is the only variable in this section of the template. (For details about treatment of blanks, see "Simple templates for parsing into words" on page 157.)

```
/* Combining positional patterns with parsing into words          */
string='R E X X'
parse var string var1 var2 4 var3 6 var4    /* Assigns: var1='R'  */
                                            /*          var2='E'  */
                                            /*          var3=' X' */
                                            /*          var4=' X' */
```

The pattern divides the template into three sections:

- `var1 var2`
- `var3`

- var4

The matching patterns split the source string into three substrings that are individually parsed into words:

- 'R E'
- ' X'
- ' X'

The variable var1 receives 'R'; var2 receives 'E'. Both var3 and var4 receive ' X' (with a blank before the X) because each is the only variable in its section of the template. (For details on treatment of blanks, see “Simple templates for parsing into words” on page 157.)

Parsing with variable patterns

You may want to specify a pattern by using the value of a variable instead of a fixed string or number. You do this by placing the name of the variable in parentheses. This is a *variable reference*. Blanks are not necessary inside or outside the parentheses, but you can add them if you wish.

The template in the next parsing instruction contains the following literal string pattern '. '.

```
parse var name fn  init '. ' ln
```

Here is how to specify that pattern as a variable string pattern:

```
strngptrn='. '
parse var name fn init (strngptrn) ln
```

If no equal, plus, or minus sign precedes the parenthesis that is before the variable name, the value of the variable is then treated as a string pattern. The variable can be one that has been set earlier in the same template.

Example:

```
/* Using a variable as a string pattern                          */
/*  The variable (delim) is set in the same template             */
SAY "Enter a date (mm/dd/yy format). =====> " /* assume 11/15/90 */
pull date
parse var date month 3 delim +1 day +2 (delim) year
            /* Sets: month='11'; delim='/'; day='15'; year='90'  */
```

If an equal, a plus, or a minus sign precedes the left parenthesis, then the value of the variable is treated as an absolute or relative positional pattern. The value of the variable must be a positive whole number or zero.

The variable can be one that has been set earlier in the same template. In the following example, the first two fields specify the starting character positions of the last two fields.

Example:

```
/* Using a variable as a positional pattern                      */
dataline = '12 26 .....Samuel ClemensMark Twain'
parse var dataline pos1 pos2 6 =(pos1) realname =(pos2) pseudonym
/* Assigns: realname='Samuel Clemens'; pseudonym='Mark Twain'    */
```

Why is the positional pattern 6 needed in the template? Remember that word parsing occurs *after* the language processor divides the source string into substrings using patterns. Therefore, the positional pattern =(pos1) cannot be correctly interpreted as =12 until *after* the language processor has split the string at column 6 and assigned the blank-delimited words 12 and 26 to pos1 and pos2, respectively.

Using UPPER

Specifying UPPER on any of the PARSE instructions converts characters to uppercase (lowercase a–z to uppercase A–Z) before parsing. The following table summarizes the effect of the parsing instructions on case.

Converts alphabetic characters to uppercase before parsing	Maintains alphabetic characters in case entered
ARG PARSE UPPER ARG	PARSE ARG
PARSE UPPER EXTERNAL	PARSE EXTERNAL
PARSE UPPER NUMERIC	PARSE NUMERIC
PULL PARSE UPPER PULL	PARSE PULL
PARSE UPPER SOURCE	PARSE SOURCE
PARSE UPPER VALUE	PARSE VALUE
PARSE UPPER VAR	PARSE VAR
PARSE UPPER VERSION	PARSE VERSION

The ARG instruction is simply a short form of PARSE UPPER ARG. The PULL instruction is simply a short form of PARSE UPPER PULL. If you do not desire uppercase translation, use PARSE ARG (instead of ARG or PARSE UPPER ARG) and use PARSE PULL (instead of PULL or PARSE UPPER PULL).

Parsing instructions summary

Remember: *All* parsing instructions assign parts of the source string into the variables named in the template. The following table summarizes where the source string comes from.

Instruction	Where the source string comes from
ARG PARSE ARG	Arguments you list when you call the program or arguments in the call to a subroutine or function.
PARSE EXTERNAL	Reads from user's terminal in TSO/E foreground, from input stream (SYSTSIN) in TSO/E background, from input stream INDD field defines in non-TSO/E address spaces.
PARSE NUMERIC	Numeric control information (from NUMERIC instruction).
PULL PARSE PULL	The string at the head of the external data queue. (If queue empty, uses default input, typically the terminal.)
PARSE SOURCE	System-supplied string giving information about the executing program.
PARSE VALUE	Expression between the keyword VALUE and the keyword WITH in the instruction.
PARSE VAR *name*	Parses the value of *name*.
PARSE VERSION	System-supplied string specifying the language, language level, and (three-word) date.

Parsing instructions examples

All examples in this section parse source strings into words.

ARG

```
/* ARG with source string named in REXX program invocation       */
/*  Program name is PALETTE.  Specify 2 primary colors (yellow,  */
/*   red, blue) on call.   Assume call is: palette red blue      */
arg var1 var2                /* Assigns: var1='RED'; var2='BLUE' */
If var1<>'RED' & var1<>'YELLOW' & var1<>'BLUE' then signal err
If var2<>'RED' & var2<>'YELLOW' & var2<>'BLUE' then signal err
total=length(var1)+length(var2)
SELECT;
  When total=7 then new='purple'
  When total=9 then new='orange'
  When total=10 then new='green'
  Otherwise new=var1                       /* entered duplicates */
END
Say new; exit                              /* Displays: "purple" */

Err:
say 'Input error--color is not "red" or "blue" or "yellow"'; exit
```

ARG converts alphabetic characters to uppercase before parsing. An example of ARG with the arguments in the CALL to a subroutine is in "Parsing multiple strings" on page 166.

PARSE ARG works the same as ARG except that PARSE ARG does not convert alphabetic characters to uppercase before parsing.

PARSE EXTERNAL

```
Say "Enter Yes or No =====> "
parse upper external answer 2 .
If answer='Y'
  then say "You said 'Yes'!"
  else say "You said 'No'!"
```

PARSE NUMERIC

```
parse numeric digits fuzz form
say digits fuzz form           /* Displays: '9 0 SCIENTIFIC'     */
                               /* (if defaults are in effect)    */
```

PARSE PULL

```
PUSH '80 7'                /* Puts data on queue                 */
parse pull fourscore seven /* Assigns: fourscore='80'; seven='7' */
SAY fourscore+seven        /* Displays: "87"                     */
```

PARSE SOURCE

```
parse source sysname .
Say sysname                          /* Displays:       "TSO"    */
```

PARSE VALUE (see "Simple templates for parsing into words" on page 157 for an example).

PARSE VAR examples are throughout the chapter, (see "Simple templates for parsing into words" on page 157 for an example).

PARSE VERSION

```
parse version . level .
say level                                    /* Displays: "3.48" */
```

PULL works the same as PARSE PULL except that PULL converts alphabetic characters to uppercase before parsing.

Advanced topics in parsing

This section includes parsing multiple strings and flow charts depicting a conceptual view of parsing.

Parsing multiple strings

Only ARG and PARSE ARG can have more than one source string. To parse *multiple strings*, you can specify multiple comma-separated templates. Here is an example:

```
parse arg template1, template2, template3
```

This instruction consists of the keywords PARSE ARG and three comma-separated templates. (For an ARG instruction, the source strings to parse come from arguments you specify when you call a program or CALL a subroutine or function.) Each comma is an instruction to the parser to move on to the next string.

Example:

```
/* Parsing multiple strings in a subroutine                    */
num='3'
musketeers="Porthos Athos Aramis D'Artagnon"
CALL Sub num,musketeers  /* Passes num and musketeers to sub     */
SAY total; say fourth /* Displays: "4" and " D'Artagnon"         */
EXIT

Sub:
parse arg subtotal, . . . fourth
total=subtotal+1
RETURN
```

Note that when a REXX program is started as a command, only one argument string is recognized. You can pass multiple argument strings for parsing:

- When one REXX program calls another REXX program with the CALL instruction or a function call.
- When programs written in other languages start a REXX program.

If there are more templates than source strings, each variable in a leftover template receives a null string. If there are more source strings than templates, the language processor ignores leftover source strings. If a template is empty (two commas in a row) or contains no variable names, parsing proceeds to the next template and source string.

Combining string and positional patterns: a special case

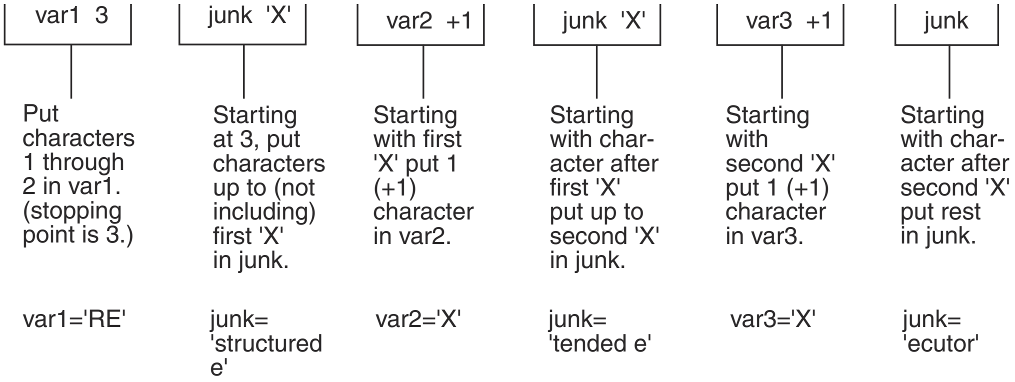
There is a special case in which absolute and relative positional patterns do not work identically. We have shown how parsing with a template containing a string pattern skips over the data in the source string that matches the pattern (see "Templates containing string patterns" on page 159). But a template containing the sequence:

- string pattern
- variable name
- *relative* positional pattern

does *not* skip over the matching data. A relative positional pattern moves relative to the first character matching a string pattern. As a result, assignment includes the data in the source string that matches the string pattern.

```
/* Template containing string pattern, then variable name, then  */
/*  relative positional pattern does not skip over any data.     */
string='REstructured eXtended eXecutor'
parse var string var1 3 junk 'X' var2 +1 junk 'X' var3 +1 junk
say var1||var2||var3 /* Concatenates variables; displays: "REXX" */
```

Here is how this template works:


var1 3
junk 'X'
var2 +1
junk 'X'
var3 +1
junk
Put characters 1 through 2 in var1. (stopping point is 3.)
Starting at 3, put characters up to (not including) first 'X' in junk.
Starting with first 'X' put 1 (+1) character in var2.
Starting with character after first 'X' put up to second 'X' in junk.
Starting with second 'X' put 1 (+1) character in var3.
Starting with character after second 'X' put rest in junk.
var1='RE'
junk= 'structured e'
var2='X'
junk= 'tended e'
var3='X'
junk= 'ecutor'

Parsing with DBCS characters

Parsing with DBCS characters generally follows the same rules as parsing with SBCS characters. Literal strings can contain DBCS characters, but numbers must be in SBCS characters. See “PARSE” on page 432 for examples of DBCS parsing.

Details of steps in parsing

The three figures that follow are to help you understand the concept of parsing. Please note that the figures do not include error cases.

The figures include terms whose definitions are as follows:

string start
is the beginning of the source string (or substring).

string end
is the end of the source string (or substring).

length
is the length of the source string.

match start
is in the source string and is the first character of the match.

match end
is in the source string. For a string pattern, it is the first character after the end of the match. For a positional pattern, it is the same as match start.

match position
is in the source string. For a string pattern, it is the first matching character. For a positional pattern, it is the position of the matching character.

token
is a distinct syntactic element in a template, such as a variable, a period, a pattern, or a comma.

value
is the numeric value of a positional pattern. This can be either a constant or the resolved value of a variable.

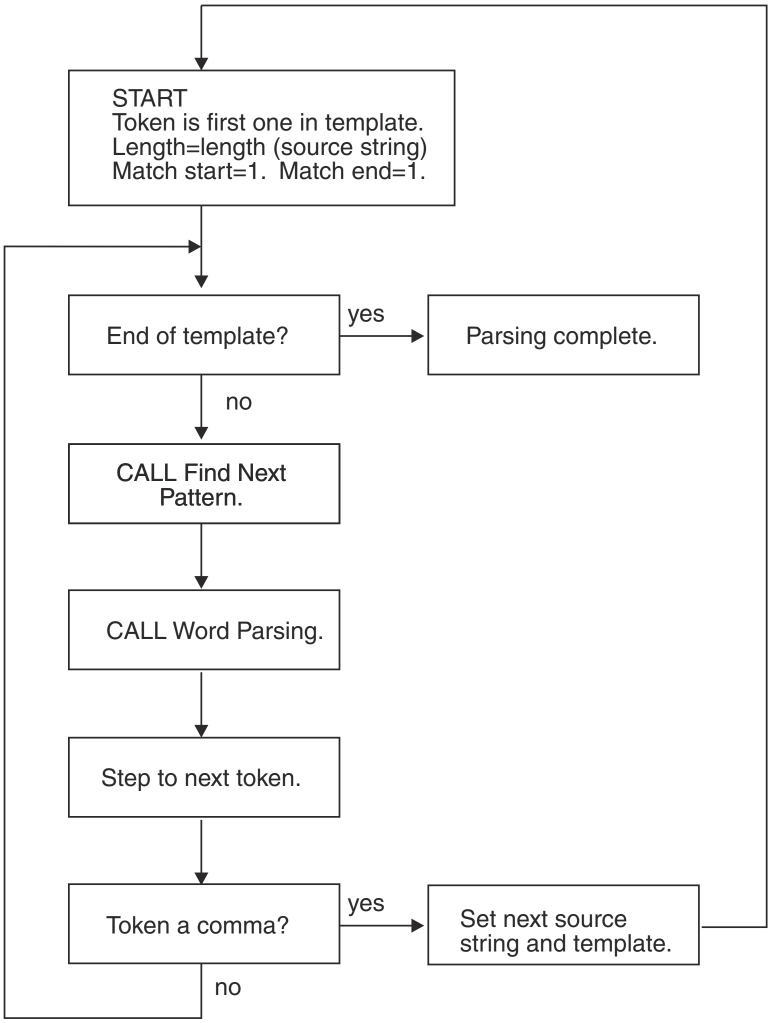

START
Token is first one in template.
Length=length (source string)
Match start=1. Match end=1.
End of template?
yes
Parsing complete.
no
CALL Find Next Pattern.
CALL Word Parsing.
Step to next token.
Token a comma?
yes
Set next source string and template.
no

Figure 10. Conceptual overview of parsing

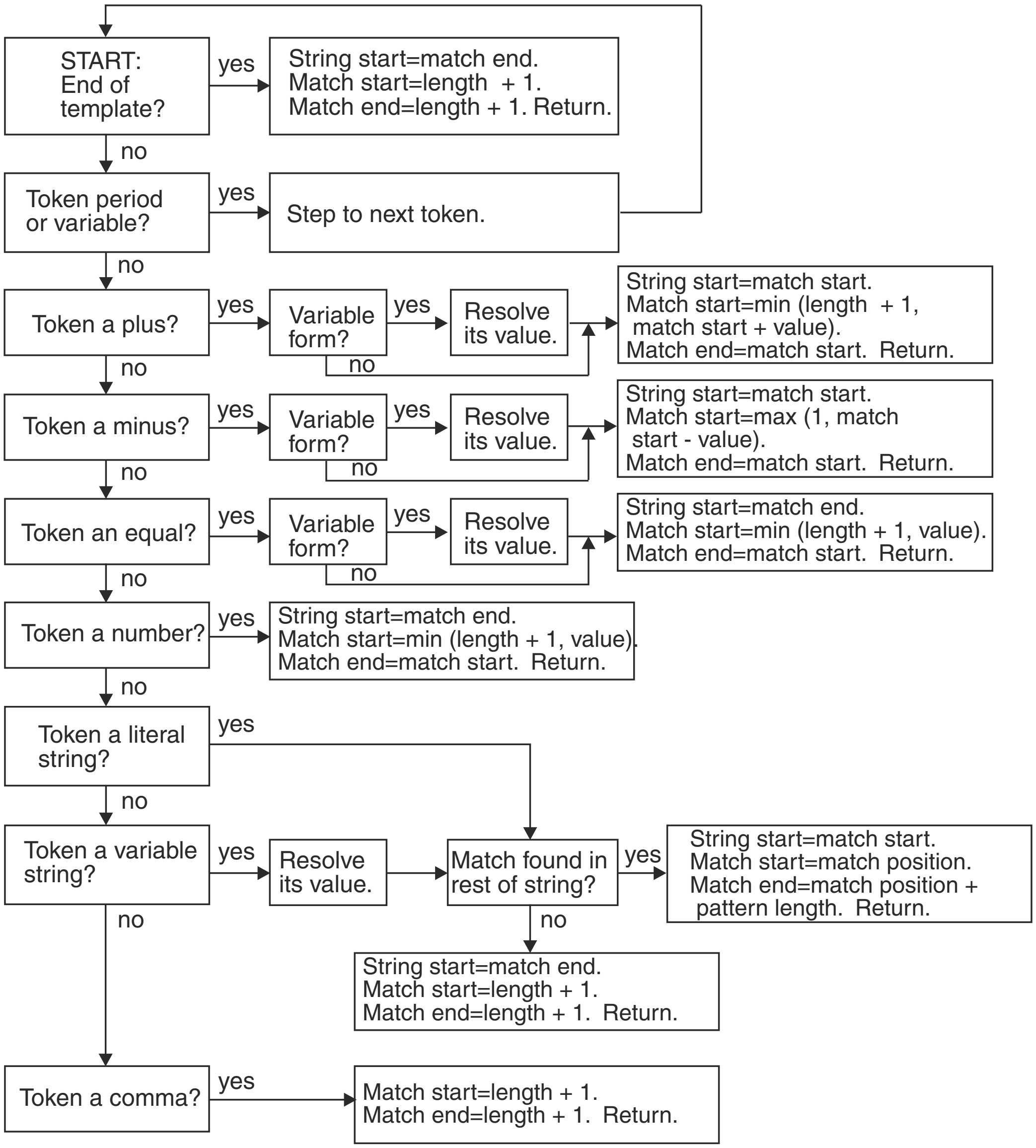
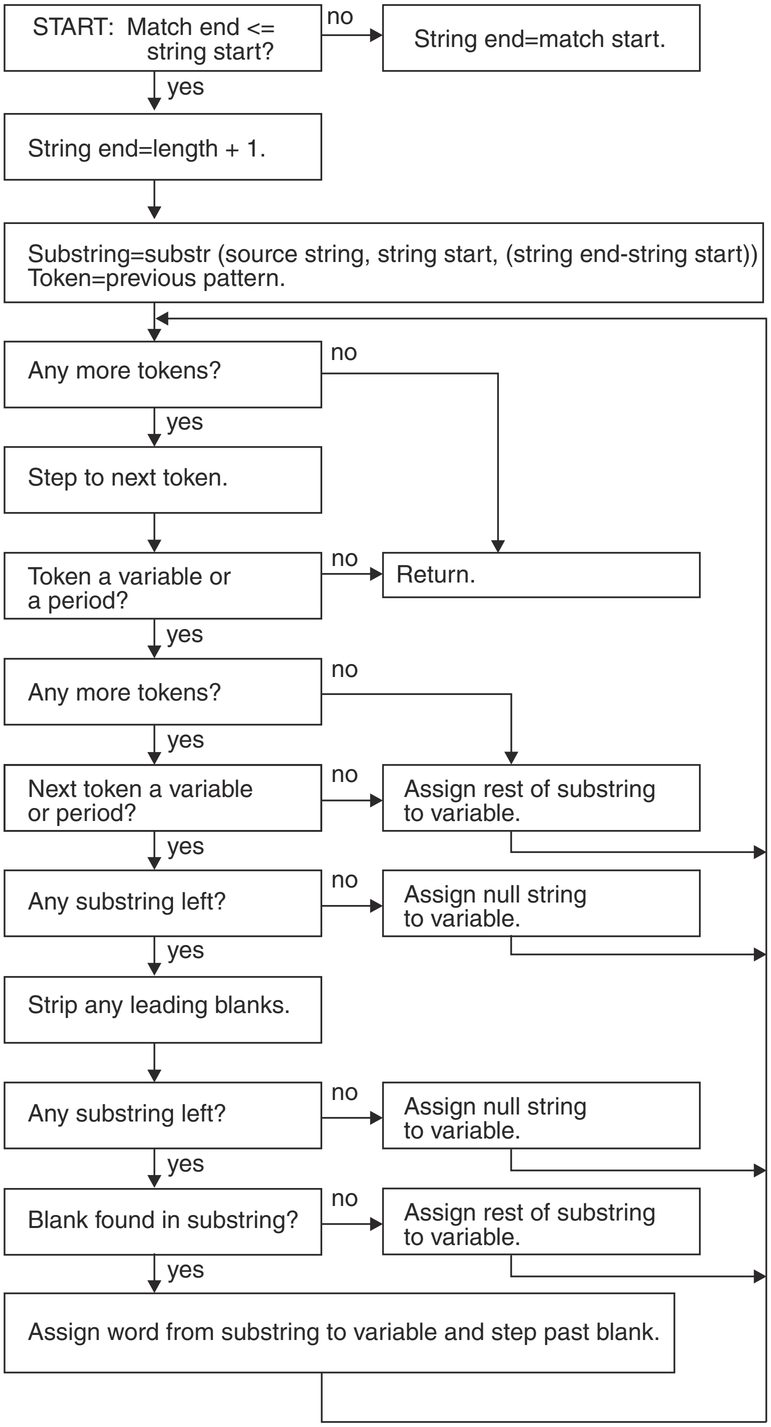


Figure 11. Conceptual view of finding next pattern



START: Match end <= string start?
no
String end=match start.
yes
String end=length + 1.
Substring=substr (source string, string start, (string end-string start))
Token=previous pattern.
Any more tokens?
no
yes
Step to next token.
Token a variable or a period?
no
Return.
yes
Any more tokens?
no
yes
Next token a variable or period?
no
Assign rest of substring to variable.
yes
Any substring left?
no
Assign null string to variable.
yes
Strip any leading blanks.
Any substring left?
no
Assign null string to variable.
yes
Blank found in substring?
no
Assign rest of substring to variable.
yes
Assign word from substring to variable and step past blank.

Figure 12. Conceptual view of word parsing

Chapter 6. Numbers and arithmetic

REXX defines the usual arithmetic operations (addition, subtraction, multiplication, and division) in as natural a way as possible. What this really means is that the rules followed are those that are conventionally taught in schools and colleges.

During the design of these facilities, however, it was found that unfortunately the rules vary considerably (indeed much more than generally appreciated) from person to person and from application to application and in ways that are not always predictable. The arithmetic described here is, therefore, a compromise that (although not the simplest) should provide acceptable results in most applications.

Introduction

Numbers (that is, character strings used as input to REXX arithmetic operations and built-in functions) can be expressed very flexibly. Leading and trailing blanks are permitted, and exponential notation can be used. Some valid numbers are:

```
    12         /* a whole number                   */
  '-76'        /* a signed whole number            */
    12.76      /* decimal places                   */
' +  0.003 '   /* blanks around the sign and so forth */
    17.        /* same as "17"                     */
      .5       /* same as "0.5"                    */
    4E9        /* exponential notation             */
     0.73e-7   /* exponential notation             */
```

In exponential notation, a number includes an exponent, a power of ten by which the number is multiplied before use. The exponent indicates how the decimal point is shifted. Thus, in the preceding examples, 4E9 is simply a short way of writing `4000000000`, and `0.73e-7` is short for `0.000000073`.

The **arithmetic operators** include addition (+), subtraction (-), multiplication (*), power (**), division (/), prefix plus (+), and prefix minus (-). In addition, there are two further division operators: integer divide (%) divides and returns the integer part; remainder (//) divides and returns the remainder.

The result of an arithmetic operation is formatted as a character string according to definite rules. The most important of these rules are as follows (see the "Definition" section for full details):

- Results are calculated up to some maximum number of significant digits (the default is 9, but you can alter this with the NUMERIC DIGITS instruction to give whatever accuracy you need). Thus, if a result requires more than 9 digits, it would usually be rounded to 9 digits. For example, the division of 2 by 3 would result in 0.666666667 (it would require an infinite number of digits for perfect accuracy).
- Except for division and power, trailing zeros are preserved (this is in contrast to most popular calculators, which remove all trailing zeros in the decimal part of results). So, for example:

  ```
  2.40 + 2    ->    4.40
  2.40 - 2    ->    0.40
  2.40 * 2    ->    4.80
  2.40 / 2    ->    1.2
  ```

  This behavior is desirable for most calculations (especially financial calculations).

  If necessary, you can remove trailing zeros with the STRIP function (see "STRIP" on page 99), or by division by 1.
- A zero result is always expressed as the single digit `0`.
- Exponential form is used for a result depending on its value and the setting of NUMERIC DIGITS (the default is 9). If the number of places needed before the decimal point exceeds the NUMERIC DIGITS

setting, or the number of places after the point exceeds twice the NUMERIC DIGITS setting, the number is expressed in exponential notation:

```
1e6 * 1e6    ->    1E+12           /* not 1000000000000 */
1 / 3E10     ->    3.33333333E-11  /* not 0.0000000000333333333 */
```

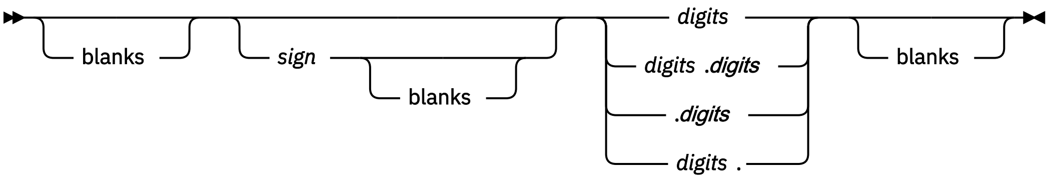
Definition

A precise definition of the arithmetic facilities of the REXX language is given here.

Numbers

A **number** in REXX is a character string that includes one or more decimal digits, with an optional decimal point. (See "Exponential notation" on page 176 for an extension of this definition.) The decimal point may be embedded in the number, or may be a prefix or suffix. The group of digits (and optional decimal point) constructed this way can have leading or trailing blanks and an optional sign (+ or -) that must come before any digits or decimal point. The sign can also have leading or trailing blanks.

Therefore, **number** is defined as:



blanks
are one or more spaces

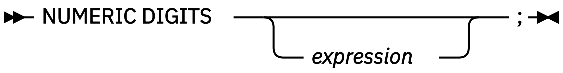
sign
is either + or -

digits
are one or more of the decimal digits 0–9.

Note that a single period alone is not a valid number.

Precision

Precision is the maximum number of significant digits that can result from an operation. This is controlled by the instruction:



The *expression* is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) to which calculations are carried out. Results are rounded to that precision, if necessary.

If you do not specify *expression* in this instruction, or if no NUMERIC DIGITS instruction has been processed since the start of a program, the default precision is used. The REXX standard for the default precision is 9.

Note that NUMERIC DIGITS can set values below the default of nine. However, use small values with care—the loss of precision and rounding thus requested affects all REXX computations, including, for example, the computation of new values for the control variable in DO loops.

Arithmetic operators

REXX arithmetic is performed by the operators +, -, *, /, %, //, and ** (add, subtract, multiply, divide, integer divide, remainder, and power), which all act on two terms, and the prefix plus and minus operators, which both act on a single term. This section describes the way in which these operations are carried out.

Before every arithmetic operation, the term or terms being operated upon have leading zeros removed (noting the position of any decimal point, and leaving only one zero if all the digits in the number are zeros). They are then truncated (if necessary) to DIGITS + 1 significant digits before being used in the computation. (The extra digit is a “guard” digit. It improves accuracy because it is inspected at the end of an operation, when a number is rounded to the required precision.) The operation is then carried out under up to double that precision, as described under the individual operations that follow. When the operation is completed, the result is rounded if necessary to the precision specified by the NUMERIC DIGITS instruction.

Rounding is done in the traditional manner. The digit to the right of the least significant digit in the result (the “guard digit”) is inspected and values of 5 through 9 are rounded up, and values of 0 through 4 are rounded down. Even/odd rounding would require the ability to calculate to arbitrary precision at all times and is, therefore, not the mechanism defined for REXX.

A conventional zero is supplied in front of the decimal point if otherwise there would be no digit before it. Significant trailing zeros are retained for addition, subtraction, and multiplication, according to the rules that follow, except that a result of zero is always expressed as the single digit 0. For division, insignificant trailing zeros are removed after rounding.

The FORMAT built-in function (see “FORMAT” on page 90) allows a number to be represented in a particular format if the standard result provided does not meet your requirements.

Arithmetic operation rules—basic operators

The basic operators (addition, subtraction, multiplication, and division) operate on numbers as follows.

Addition and subtraction

If either number is 0, the other number, rounded to NUMERIC DIGITS digits, if necessary, is used as the result (with sign adjustment as appropriate). Otherwise, the two numbers are extended on the right and left as necessary, up to a total maximum of DIGITS + 1 digits (the number with the smaller absolute value may, therefore, lose some or all of its digits on the right) and are then added or subtracted as appropriate.

Example:

```
                xxx.xxx  +  yy.yyyyy
```

becomes:

```
                xxx.xxx00
             +  0yy.yyyyy
             -------------
                zzz.zzzzz
```

The result is then rounded to the current setting of NUMERIC DIGITS if necessary (taking into account any extra “carry digit” on the left after addition, but otherwise counting from the position corresponding to the most significant digit of the terms being added or subtracted). Finally, any insignificant leading zeros are removed.

The prefix operators are evaluated using the same rules; the operations `+number` and `-number` are calculated as `0+number` and `0-number`, respectively.

Multiplication

The numbers are multiplied together ("long multiplication") resulting in a number that may be as long as the sum of the lengths of the two operands.

Example:

```
              xxx.xxx * yy.yyyyy
```

becomes:

```
              zzzzz.zzzzzzzz
```

The result is then rounded, counting from the first significant digit of the result, to the current setting of NUMERIC DIGITS.

Division

For the division:

```
yyy / xxxxx
```

the following steps are taken: First the number yyy is extended with zeros on the right until it is larger than the number xxxxx (with note being taken of the change in the power of ten that this implies). Thus, in this example, yyy might become yyy00. Traditional long division then takes place. This might be written:

```
               zzzz
            ----------
     xxxxx  | yyy00
```

The length of the result (zzzz) is such that the rightmost z is at least as far right as the rightmost digit of the (extended) y number in the example. During the division, the y number is extended further as necessary. The z number may increase up to NUMERIC DIGITS+1 digits, at which point the division stops and the result is rounded. Following completion of the division (and rounding if necessary), insignificant trailing zeros are removed.

Basic operator examples

Following are some examples that illustrate the main implications of the rules just described.

```
/* With:  Numeric digits 5 */
12+7.00      ->    19.00
1.3-1.07     ->     0.23
1.3-2.07     ->    -0.77
1.20*3       ->     3.60
7*3          ->    21
0.9*0.8      ->     0.72
1/3          ->     0.33333
2/3          ->     0.66667
5/2          ->     2.5
1/10         ->     0.1
12/12        ->     1
8.0/2        ->     4
```

With all the basic operators, the position of the decimal point in the terms being operated upon is arbitrary. The operations may be carried out as integer operations with the exponent being calculated and applied afterward. Therefore, the significant digits of a result are not in any way dependent on the position of the decimal point in either of the terms involved in the operation.

b

Arithmetic operation rules—additional operators

The operation rules for the power (**), integer divide (%), and remainder (//) operators follow.

Power

The **** (power) operator** raises a number to a power, which may be positive, negative, or 0. The power must be a whole number. (The second term in the operation must be a whole number and is rounded to DIGITS digits, if necessary, as described under “Numbers used directly by REXX” on page 178.) If negative, the absolute value of the power is used, and then the result is inverted (divided into 1). For calculating the power, the number is effectively multiplied by itself for the number of times expressed by the power, and finally trailing zeros are removed (as though the result were divided by 1).

In practice (see Note "1" on page 175 for the reasons), the power is calculated by the process of left-to-right binary reduction. For a**n: n is converted to binary, and a temporary accumulator is set to 1. If n = 0 the initial calculation is complete. (Thus, a**0 = 1 for all a, including 0**0.) Otherwise each bit (starting at the first nonzero bit) is inspected from left to right. If the current bit is 1, the accumulator is multiplied by a. If all bits have now been inspected, the initial calculation is complete; otherwise the accumulator is squared and the next bit is inspected for multiplication. When the initial calculation is complete, the temporary result is divided into 1 if the power was negative.

The multiplications and division are done under the arithmetic operation rules, using a precision of DIGITS + L + 1 digits. L is the length in digits of the integer part of the whole number n (that is, excluding any decimal part, as though the built-in function TRUNC(*n*) had been used). Finally, the result is rounded to NUMERIC DIGITS digits, if necessary, and insignificant trailing zeros are removed.

Integer division

The **% (integer divide) operator** divides two numbers and returns the integer part of the result. The result returned is defined to be that which would result from repeatedly subtracting the divisor from the dividend while the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used: the sign of the final result is the same as that which would result from regular division.

The result returned has no fractional part (that is, no decimal point or zeros following it). If the result cannot be expressed as a whole number, the operation is in error and will fail—that is, the result must not have more digits than the current setting of NUMERIC DIGITS. For example, 10000000000%3 requires 10 digits for the result (3333333333) and would, therefore, fail if NUMERIC DIGITS 9 were in effect. Note that this operator may not give the same result as truncating regular division (which could be affected by rounding).

Remainder

The **// (remainder) operator** returns the remainder from integer division and is defined as being the residue of the dividend after the operation of calculating integer division as previously described. The sign of the remainder, if nonzero, is the same as that of the original dividend.

This operation fails under the same conditions as integer division (that is, if integer division on the same two terms would fail, the remainder cannot be calculated).

Additional operator examples

Following are some examples using the power, integer divide, and remainder operators:

```
/* Again with:  Numeric digits 5 */
2**3       ->     8
2**-3      ->     0.125
1.7**8     ->    69.758
2%3        ->     0
2.1//3     ->     2.1
10%3       ->     3
10//3      ->     1
-10//3     ->    -1
10.2//1    ->     0.2
10//0.3    ->     0.1
3.6//1.3   ->     1.0
```

Note:

1. A particular algorithm for calculating powers is used, because it is efficient (though not optimal) and considerably reduces the number of actual multiplications performed. It, therefore, gives better performance than the simpler definition of repeated multiplication. Because results may differ from those of repeated multiplication, the algorithm is defined here.

2. The integer divide and remainder operators are defined so that they can be calculated as a by-product of the standard division operation. The division process is ended as soon as the integer result is available; the residue of the dividend is the remainder.

Numeric comparisons

The comparison operators are listed in “Comparison” on page 14. You can use any of these for comparing numeric strings. However, you should not use ==, \==, ¬==, >>, \>>, ¬>>, <<, \<<, and ¬<< for comparing numbers because leading and trailing blanks and leading zeros are significant with these operators.

A comparison of numeric values is effected by subtracting the two numbers (calculating the difference) and then comparing the result with 0. That is, the operation:

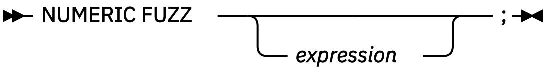
```
A ? Z
```

where ? is any numeric comparison operator, is identical with:

```
(A - Z) ? '0'
```

It is, therefore, the *difference* between two numbers, when subtracted under REXX subtraction rules, that determines their equality.

A quantity called **fuzz** affects the comparison of two numbers. This controls the amount by which two numbers may differ before being considered equal for the purpose of comparison. The FUZZ value is set by the instruction:



Here *expression* must result in a positive whole number or zero. The default is 0.

The effect of FUZZ is to temporarily reduce the value of DIGITS by the FUZZ value for each numeric comparison. That is, the numbers are subtracted under a precision of DIGITS minus FUZZ digits during the comparison. Clearly the FUZZ setting must be less than DIGITS.

Thus if DIGITS = 9 and FUZZ = 1, the comparison is carried out to 8 significant digits, just as though `NUMERIC DIGITS 8` had been put in effect for the duration of the operation.

Example:

```
Numeric digits 5
Numeric fuzz 0
say  4.9999 = 5     /* Displays "0"    */
say  4.9999 < 5     /* Displays "1"    */
Numeric fuzz 1
say  4.9999 = 5     /* Displays "1"    */
say  4.9999 < 5     /* Displays "0"    */
```

Exponential notation

The preceding description of numbers describes "pure" numbers, in the sense that the character strings that describe numbers can be very long. For example:

```
10000000000 * 10000000000
```

would give

```
100000000000000000000
```

and

```
.00000000001 * .00000000001
```

would give

```
0.0000000000000000000001
```

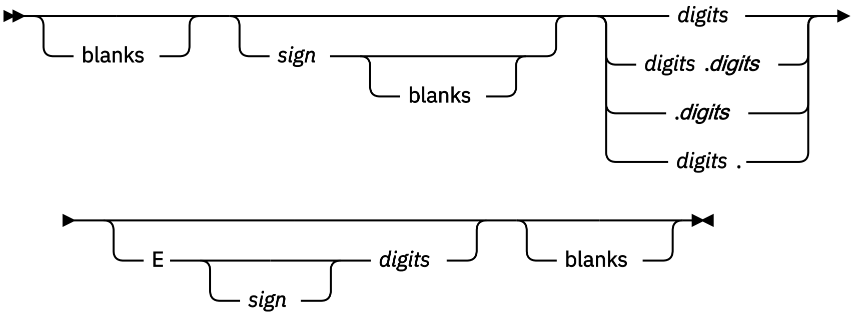
For both large and small numbers some form of exponential notation is useful, both to make long numbers more readable, and to make execution possible in extreme cases. In addition, exponential notation is used whenever the "simple" form would give misleading information.

For example:

```
numeric digits 5
say 54321*54321
```

would display 2950800000 in long form. This is clearly misleading, and so the result is expressed as 2.9508E+9 instead.

The definition of numbers is, therefore, extended as:



The integer following the E represents a power of ten that is to be applied to the number. The E can be in uppercase or lowercase.

Certain character strings are numbers even though they do not appear to be numeric to the user. Specifically, because of the format of numbers in exponential notation, strings, such as 0E123 (0 times 10 raised to the 123 power) and 1E342 (1 times 10 raised to the 342 power), are numeric. In addition, a comparison such as 0E123=0E567 gives a true result of 1 (0 is equal to 0). To prevent problems when comparing nonnumeric strings, use the strict comparison operators.

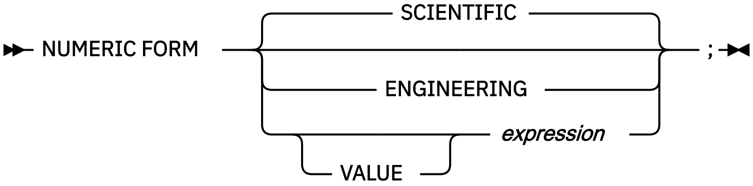
Here are some examples:

```
12E7   =    120000000          /* Displays "1" */
12E-5  =    0.00012            /* Displays "1" */
-12e4  =    -120000            /* Displays "1" */
0e123  =    0e456              /* Displays "1" */
0e123  ==   0e456              /* Displays "0" */
```

The preceding numbers are valid for input data at all times. The results of calculations are returned in either conventional or exponential form, depending on the setting of NUMERIC DIGITS. If the number of places needed before the decimal point exceeds DIGITS, or the number of places after the point exceeds twice DIGITS, exponential form is used. The exponential form REXX generates always has a sign following the E to improve readability. If the exponent is 0, then the exponential part is omitted—that is, an exponential part of E+0 is never generated.

You can explicitly convert numbers to exponential form, or force them to be displayed in long form, by using the FORMAT built-in function (see "FORMAT" on page 90).

Scientific notation is a form of exponential notation that adjusts the power of ten so a single nonzero digit appears to the left of the decimal point. **Engineering notation** is a form of exponential notation in which from one to three digits (but not simply 0) appear before the decimal point, and the power of ten is always expressed as a multiple of three. The integer part may, therefore, range from 1 through 999. You can control whether Scientific or Engineering notation is used with the instruction:



Scientific notation is the default.

```
/* after the instruction */
Numeric form scientific

123.45 * 1e11     ->     1.2345E+13

/* after the instruction */
Numeric form engineering

123.45  * 1e11    ->    12.345E+12
```

Numeric information

To determine the current settings of the NUMERIC options, use the built-in functions DIGITS, FORM, and FUZZ. These functions return the current settings of NUMERIC DIGITS, NUMERIC FORM, and NUMERIC FUZZ, respectively.

Whole numbers

Within the set of numbers REXX understands, it is useful to distinguish the subset defined as **whole numbers**. A whole number in REXX is a number that has a decimal part that is all zeros (or that has no decimal part). In addition, it must be possible to express its integer part simply as digits within the precision set by the NUMERIC DIGITS instruction. REXX would express larger numbers in exponential notation, after rounding, and, therefore, these could no longer be safely described or used as whole numbers.

Numbers used directly by REXX

As discussed, the result of any arithmetic operation is rounded (if necessary) according to the setting of NUMERIC DIGITS. Similarly, when REXX directly uses a number (which has not necessarily been involved in an arithmetic operation), the same rounding is also applied. It is just as though the number had been added to 0.

In the following cases, the number used must be a whole number, and an implementation restriction on the largest number that can be used may apply:

- The positional patterns in parsing templates (including variable positional patterns)
- The power value (right hand operand) of the power operator
- The values of *exprr* and *exprf* in the DO instruction
- The values given for DIGITS or FUZZ in the NUMERIC instruction
- Any number used in the numeric option in the TRACE instruction

Errors

Two types of errors may occur during arithmetic:

- Overflow or Underflow

  This error occurs if the exponential part of a result would exceed the range that the language processor can handle, when the result is formatted according to the current settings of NUMERIC DIGITS and NUMERIC FORM. The language defines a minimum capability for the exponential part, namely the largest number that can be expressed as an exact integer in default precision. Because the default precision is 9, TSO/E supports exponents in the range -999999999 through 999999999.

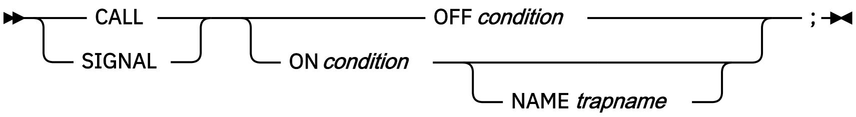
Because this allows for (very) large exponents, overflow or underflow is treated as a syntax error.

- Insufficient storage

  Storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail because of lack of storage. This is considered a terminating error as usual, rather than an arithmetic error.

Chapter 7. Conditions and condition traps

A **condition** is a specified event or state that CALL ON or SIGNAL ON can trap. A condition trap can modify the flow of execution in a REXX program. Condition traps are turned on or off using the ON or OFF subkeywords of the SIGNAL and CALL instructions (see "CALL" on page 43 and "SIGNAL" on page 66).



condition and *trapname* are single symbols that are taken as constants. Following one of these instructions, a condition trap is set to either ON (enabled) or OFF (disabled). The initial setting for all condition traps is OFF.

If a condition trap is enabled and the specified *condition* occurs, control passes to the routine or label *trapname* if you have specified *trapname*. Otherwise, control passes to the routine or label *condition*. CALL or SIGNAL is used, depending on whether the most recent trap for the condition was set using CALL ON or SIGNAL ON, respectively.

Note: If you use CALL, the *trapname* can be an internal label, a built-in function, or an external routine. If you use SIGNAL, the *trapname* can be only an internal label.

The conditions and their corresponding events that can be trapped are:

ERROR

raised if a command indicates an error condition upon return. It is also raised if any command indicates failure and neither CALL ON FAILURE nor SIGNAL ON FAILURE is active. The condition is raised at the end of the clause that called the command but is ignored if the ERROR condition trap is already in the delayed state. The **delayed state** is the state of a condition trap when the condition has been raised but the trap has not yet been reset to the enabled (ON) or disabled (OFF) state.

In TSO/E, SIGNAL ON ERROR traps all positive return codes, and negative return codes only if CALL ON FAILURE and SIGNAL ON FAILURE are not set.

Note: In TSO/E, a command is not only a TSO/E command processor. See "Host commands and host command environments" on page 23 for a definition of *host commands*.

FAILURE

raised if a command indicates a failure condition upon return. The condition is raised at the end of the clause that called the command but is ignored if the FAILURE condition trap is already in the delayed state.

In TSO/E, CALL ON FAILURE and SIGNAL ON FAILURE trap all negative return codes from commands.

HALT

raised if an external attempt is made to interrupt and end execution of the program. The condition is usually raised at the end of the clause that was being processed when the external interruption occurred.

For example, the TSO/E REXX immediate command HI (Halt Interpretation) or the EXECUTIL HI command raises a halt condition. The HE (Halt Execution) immediate command does not raise a halt condition. See "Interrupting execution and controlling tracing" on page 235.

NOVALUE

raised if an uninitialized variable is used:

- As a term in an expression
- As the *name* following the VAR subkeyword of a PARSE instruction
- As a variable reference in a parsing template, a PROCEDURE instruction, or a DROP instruction.

Note: SIGNAL ON NOVALUE can trap any uninitialized variables except tails in compound variables.

```
/* The following does not raise NOVALUE. */
signal on novalue
a.=0
say a.z
say 'NOVALUE is not raised.'
exit

novalue:
say 'NOVALUE is raised.'
```

You can specify this condition only for SIGNAL ON.

SYNTAX

raised if any language processing error is detected while the program is running. This includes all kinds of processing errors, including true syntax errors and “runtime” errors, such as attempting an arithmetic operation on nonnumeric terms. You can specify this condition only for SIGNAL ON.

Any ON or OFF reference to a condition trap replaces the previous state (ON, OFF, or DELAY, and any *trapname*) of that condition trap. Thus, a CALL ON HALT replaces any current SIGNAL ON HALT (and a SIGNAL ON HALT replaces any current CALL ON HALT), a CALL ON or SIGNAL ON with a new trap name replaces any previous trap name, any OFF reference disables the trap for CALL or SIGNAL, and so on.

Action taken when a condition is not trapped

When a condition trap is currently disabled (OFF) and the specified condition occurs, the default action depends on the condition:

- For HALT and SYNTAX, the processing of the program ends, and a message (see *z/OS TSO/E Messages*) describing the nature of the event that occurred usually indicates the condition.
- For all other conditions, the condition is ignored and its state remains OFF.

Action taken when a condition is trapped

When a condition trap is currently enabled (ON) and the specified condition occurs, instead of the usual flow of control, a CALL *trapname* or SIGNAL *trapname* instruction is processed automatically. You can specify the *trapname* after the NAME subkeyword of the CALL ON or SIGNAL ON instruction. If you do not specify a *trapname*, the name of the condition itself (ERROR, FAILURE, HALT, NOVALUE, or SYNTAX) is used.

For example, the instruction `call on error` enables the condition trap for the ERROR condition. If the condition occurred, then a call to the routine identified by the name ERROR is made. The instruction `call on error name commanderror` would enable the trap and call the routine COMMANDERROR if the condition occurred.

The sequence of events, after a condition has been trapped, varies depending on whether a SIGNAL or CALL is processed:

- If the action taken is a SIGNAL, execution of the current instruction ceases immediately, the condition is disabled (set to OFF), and the SIGNAL takes place in exactly the same way as usual (see “SIGNAL” on page 66).

  If any new occurrence of the condition is to be trapped, a new CALL ON or SIGNAL ON instruction for the condition is required to re-enable it when the label is reached. For example, if SIGNAL ON SYNTAX is enabled when a SYNTAX condition occurs, then, if the SIGNAL ON SYNTAX label name is not found, a usual syntax error termination occurs.
- If the action taken is a CALL (which can occur only at a clause boundary), the CALL is made in the usual way (see “CALL” on page 43) except that the call does not affect the special variable RESULT. If the routine should RETURN any data, then the returned character string is ignored.

  Because these conditions (ERROR, FAILURE, and HALT) can arise during execution of an INTERPRET instruction, execution of the INTERPRET may be interrupted and later resumed if CALL ON was used.

As the condition is raised, and before the CALL is made, the condition trap is put into a delayed state. This state persists until the RETURN from the CALL, or until an explicit CALL (or SIGNAL) ON (or OFF) is made for the condition. This delayed state prevents a premature condition trap at the start of the routine called to process a condition trap. When a condition trap is in the delayed state it remains enabled, but if the condition is raised again, it is either ignored (for ERROR or FAILURE) or (for the other conditions) any action (including the updating of the condition information) is delayed until one of the following events occurs:

1. A CALL ON or SIGNAL ON, for the delayed condition, is processed. In this case a CALL or SIGNAL takes place immediately after the new CALL ON or SIGNAL ON instruction has been processed.
2. A CALL OFF or SIGNAL OFF, for the delayed condition, is processed. In this case the condition trap is disabled and the default action for the condition occurs at the end of the CALL OFF or SIGNAL OFF instruction.
3. A RETURN is made from the subroutine. In this case the condition trap is no longer delayed and the subroutine is called again immediately.

On RETURN from the CALL, the original flow of execution is resumed (that is, the flow is not affected by the CALL).

Note:

1. You must be extra careful when you write a syntax trap routine. Where possible, put the routine near the beginning of the program. This is necessary because the trap routine label might not be found if there are certain scanning errors, such as a missing ending comment. Also, the trap routine should not contain any statements that might cause more of the program in error to be scanned. Examples of this are calls to built-in functions with no quotation marks around the name. If the built-in function name is in uppercase and is enclosed in quotation marks, REXX goes directly to the function, rather than searching for an internal label.
2. In all cases, the condition is raised immediately upon detection. If SIGNAL ON traps the condition, the current instruction is ended, if necessary. Therefore, the instruction during which an event occurs may be only partly processed. For example, if SYNTAX is raised during the evaluation of the expression in an assignment, the assignment does not take place. Note that the CALL for ERROR, FAILURE, and HALT traps can occur only at clause boundaries. If these conditions arise in the middle of an INTERPRET instruction, execution of INTERPRET may be interrupted and later resumed. Similarly, other instructions, for example, DO or SELECT, may be temporarily interrupted by a CALL at a clause boundary.
3. The state (ON, OFF, or DELAY, and any *trapname*) of each condition trap is saved on entry to a subroutine and is then restored on RETURN. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller. See the CALL instruction ("CALL" on page 43) for details of other information that is saved during a subroutine call.
4. The state of condition traps is not affected when an external routine is called by a CALL, even if the external routine is a REXX program. On entry to any REXX program, all condition traps have an initial setting of OFF.
5. While user input is processed during interactive tracing, all condition traps are temporarily set OFF. This prevents any unexpected transfer of control—for example, should the user accidentally use an uninitialized variable while SIGNAL ON NOVALUE is active. For the same reason, a syntax error during interactive tracing does not cause exit from the program but is trapped specially and then ignored after a message is given.
6. The system interface detects certain execution errors either before execution of the program starts or after the program has ended. SIGNAL ON SYNTAX cannot trap these errors.

Note that a **label** is a clause consisting of a single symbol followed by a colon. Any number of successive clauses can be labels; therefore, multiple labels are allowed before another type of clause.

Condition information

When any condition is trapped and causes a SIGNAL or CALL, this becomes the current trapped condition, and certain condition information associated with it is recorded. You can inspect this information by using the CONDITION built-in function (see "CONDITION" on page 82).

The condition information includes:

- The name of the current trapped condition
- The name of the instruction processed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition
- Any descriptive string associated with that condition

The current condition information is replaced when control is passed to a label as the result of a condition trap (CALL ON or SIGNAL ON). Condition information is saved and restored across subroutine or function calls, including one because of a CALL ON trap. Therefore, a routine called by a CALL ON can access the appropriate condition information. Any previous condition information is still available after the routine returns.

Descriptive strings

The descriptive string varies, depending on the condition trapped.

ERROR
The string that was processed and resulted in the error condition.

FAILURE
The string that was processed and resulted in the failure condition.

HALT
Any string associated with the halt request. This can be the null string if no string was provided.

NOVALUE
The derived name of the variable whose attempted reference caused the NOVALUE condition. The NOVALUE condition trap can be enabled only using SIGNAL ON.

SYNTAX
Any string the language processor associated with the error. This can be the null string if you did not provide a specific string. Note that the special variables RC and SIGL provide information about the nature and position of the processing error. You can enable the SYNTAX condition trap only by using SIGNAL ON.

Special variables

A special variable is one that may be set automatically during processing of a REXX program. There are three special variables: RC, RESULT, and SIGL. None of these has an initial value, but the program may alter them. (For information about RESULT, see "RETURN" on page 64.)

The special variable RC

For ERROR and FAILURE, the REXX special variable RC is set to the command return code, as usual, before control is transferred to the condition label. The return code may be the return code from a TSO/E command processor or a routine (such as, a CLIST, REXX exec, program, and so on) that caused the ERROR or FAILURE condition. The return code may also be a -3, which indicates that the command could not be found. For more information about issuing commands and their return codes, see "Host commands and host command environments" on page 23.

For SIGNAL ON SYNTAX, RC is set to the syntax error number. If you use a SIGNAL ON SYNTAX error trap to trap syntax error conditions, RC will be set to the syntax error number before control is passed to the SYNTAX trap. The syntax error number *nn* is a numeric identifier in the range 0-99 that corresponds to TSO/E REXX error message IRX00nnI. See *z/OS TSO/E Messages* for a full description of message IRX00nnI.

You can also use the REXX ERRORTEXT(nn) function to display error text associated with error number *nn*. For example, if the user received error 8, then the following statemement

```
say ERRORTEXT(8)
```

will display the following error text:

```
Unexpected THEN or ELSE
```

This corresponds to the full error message:

```
IRX0008I Error running execname line nn: Unexpected THEN or ELSE
```

For more information, see "ERRORTEXT" on page 89.

The special variable SIGL

Following any transfer of control because of a CALL or SIGNAL, the program line number of the clause causing the transfer of control is stored in the special variable SIGL. Where the transfer of control is because of a condition trap, the line number assigned to SIGL is that of the last clause processed (at the current subroutine level) before the CALL or SIGNAL took place. This is especially useful for SIGNAL ON SYNTAX when the number of the line in error can be used, for example, to control a text editor. Typically, code following the SYNTAX label may PARSE SOURCE to find the source of the data, then call an editor to edit the source file positioned at the line in error. Note that in this case you may have to run the program again before any changes made in the editor can take effect.

Alternatively, SIGL can be used to help determine the cause of an error (such as the occasional failure of a function call) as in the following example:

```
signal on syntax
a = a + 1      /* This is to create a syntax error */
say 'SYNTAX error not raised'
exit

/* Standard handler for SIGNAL ON SYNTAX */
syntax:
  say 'REXX error' rc 'in line' sigl':' "ERRORTEXT"(rc)
  say "SOURCELINE"(sigl)
  trace ?r; nop
```

This code first displays the error code, line number, and error message. It then displays the line in error, and finally drops into debug mode to let you inspect the values of the variables used at the line in error.

Chapter 8. Using REXX in different address spaces

TSO/E provides support for the REXX programming language in any MVS address space. You can run REXX execs in the TSO/E address space and in any non-TSO/E address space, such as CICS TS or IMS.

The REXX language consists of keyword instructions and built-in functions that you use in a REXX exec. The keyword instructions and built-in functions are described in Chapter 3, "Keyword instructions," on page 41 and Chapter 4, "Functions," on page 73, respectively.

TSO/E also provides TSO/E external functions and REXX commands you can use in a REXX exec. The functions are described in "TSO/E external functions" on page 109. The TSO/E REXX commands provide additional services that let you:

- Control I/O processing to and from data sets
- Perform data stack requests
- Change characteristics that control how a REXX exec runs
- Check for the existence of a specific host command environment.

Chapter 10, "TSO/E REXX commands," on page 201 describes the commands.

In an exec, you can use any of the keyword instructions and built-in functions regardless of whether the exec runs in a TSO/E or non-TSO/E address space. There are, however, differences in the TSO/E external functions, commands, and programming services you can use in an exec depending on whether the exec will run in a TSO/E address space or in a non-TSO/E address space. For example, you can use the TSO/E external function SETLANG in an exec that runs in any MVS address space. However, you can use the LISTDSI external function only in execs that run in a TSO/E address space. The following topics describe the services you can use in execs that run in TSO/E and non-TSO/E address spaces:

- "Writing execs that run in Non-TSO/E address spaces" on page 189
- "Writing execs that run in the TSO/E address space" on page 191.

TSO/E provides the TSO/E environment service, IKJTSOEV, that lets you create a TSO/E environment in a non-TSO/E address space. If you use IKJTSOEV and then run a REXX exec in the TSO/E environment that is created, the exec can contain TSO/E external functions, commands, and services that an exec running in a TSO/E address space can use. That is, the TSO host command environment (ADDRESS TSO) is available to the exec. *z/OS TSO/E Programming Services* describes the TSO/E environment service and the different considerations for running REXX execs within the environment.

TSO/E REXX is the implementation of the SAA Procedures Language on the MVS system. By using the keyword instructions and functions that are defined for the SAA Procedures Language, you can write REXX programs that can run in any of the supported SAA environments. See *SAA Common Programming Interface REXX Level 2 Reference* for more information.

Additional REXX support

In addition to the keyword instructions, built-in functions, and TSO/E external functions and REXX commands, TSO/E provides ***programming services*** you can use to interface with REXX and the language processor and ***customizing services*** that let you customize REXX processing and how system services are accessed and used.

TSO/E REXX programming services

The REXX programming services that TSO/E provides in addition to REXX language support are:

IRXEXCOM – Variable Access
The variable access routine IRXEXCOM lets you access and manipulate the current generation of REXX variables. Unauthorized commands and programs can invoke IRXEXCOM to inspect, set, and drop REXX variables. "Variable access routine - IRXEXCOM" on page 274 describes IRXEXCOM.

IRXSUBCM – Maintain Host Command Environments

The IRXSUBCM routine is a programming interface to the *host command environment table*. The table contains the names of the environments and routines that handle the processing of host commands. You can use IRXSUBCM to add, change, and delete entries in the table and to query entries. "Maintain entries in the host command environment table - IRXSUBCM" on page 280 describes the IRXSUBCM routine.

IRXIC – Trace and Execution Control

The trace and execution control routine IRXIC is an interface to the immediate commands HI, HT, RT, TS, and TE. A program can invoke IRXIC to use one of these commands to affect the processing and tracing of REXX execs. "Trace and execution control routine - IRXIC" on page 284 describes the routine.

IRXRLT – Get Result

You can use the *get result* routine, IRXRLT, to get the result from a REXX exec that was invoked with the IRXEXEC routine. If you write an external function or subroutine that is link-edited into a load module, you can use IRXRLT to obtain storage to return the result to the calling exec. The IRXRLT routine also lets a compiler runtime processor obtain an evaluation block to handle the result from a compiled REXX exec. "Get result routine - IRXRLT" on page 286 describes the IRXRLT routine.

IRXJCL and IRXEXEC – Exec Processing

You can use the IRXJCL and IRXEXEC routines to invoke a REXX exec in any address space. The two routines are programming interfaces to the language processor. You can run an exec in MVS batch by specifying IRXJCL as the program name on the JCL EXEC statement. You can invoke either IRXJCL or IRXEXEC from an application program, including a REXX exec, in any address space to invoke a REXX exec. "Exec processing routines - IRXJCL and IRXEXEC" on page 245 describes the IRXJCL and IRXEXEC routines.

External Functions and Subroutines, and Function Packages

You can write your own external functions and subroutines to extend the programming capabilities of the REXX language. You can write external functions or subroutines in REXX. You can also write external functions or subroutines in any programming language that supports the system-dependent interfaces that the language processor uses to invoke the function or subroutine.

You can also group frequently used external functions and subroutines into a *package*, which allows for quick access to the packaged functions and subroutines. If you want to include an external function or subroutine in a function package, the function or subroutine must be link-edited into a load module. "External functions and subroutines, and function packages" on page 263 describes the system-dependent interfaces for writing external functions and subroutines and how to define function packages.

IRXSAY – SAY Instruction Routine

The SAY instruction routine, IRXSAY, lets you write a character string to the same output stream as the REXX SAY keyword instruction. "SAY instruction routine - IRXSAY" on page 292 describes the IRXSAY routine.

IRXHLT – Halt Condition Routine

The halt condition routine, IRXHLT, lets you query or reset the halt condition. "Halt condition routine - IRXHLT" on page 295 describes the IRXHLT routine.

IRXTXT – Text Retrieval Routine

The text retrieval routine, IRXTXT, lets you retrieve the same text that the TSO/E REXX interpreter uses for the ERRORTEXT built-in function and for certain options of the DATE built-in function. For example, using IRXTXT, a program can retrieve the name of a month or the text of a syntax error message. "Text retrieval routine - IRXTXT" on page 297 describes the IRXTXT routine.

IRXLIN – LINESIZE Function Routine

The LINESIZE function routine, IRXLIN, lets you retrieve the same value that the LINESIZE built-in function returns. "LINESIZE function routine - IRXLIN" on page 302 describes the IRXLIN routine.

TSO/E REXX customizing services

In addition to the programming support to write REXX execs and REXX programming services that allow you to interface with REXX and the language processor, TSO/E also provides services you can use to customize REXX processing. Many services let you change how an exec is processed and how the language processor interfaces with the system to access and use system services, such as storage and I/O. Customization services for REXX processing include the following:

Environment Characteristics
TSO/E provides various routines and services that allow you to customize the environment in which the language processor processes a REXX exec. This environment is known as the *language processor environment* and defines various characteristics relating to how execs are processed and how system services are accessed and used. TSO/E provides default environment characteristics that you can change and also provides a routine you can use to define your own environment.

Replaceable Routines
When a REXX exec runs, various system services are used, such as services for loading and freeing an exec, I/O, obtaining and freeing storage, and data stack requests. TSO/E provides routines that handle these types of system services. The routines are known as *replaceable routines* because you can provide your own routine that either replaces the system routine or that performs pre-processing and then calls the system routine.

Exit Routines
You can provide exit routines to customize various aspects of REXX processing.

Information about the different ways in which you can customize REXX processing are described in Chapters 13 - 16.

Writing execs that run in Non-TSO/E address spaces

As described above, you can run REXX execs in any MVS address space (both TSO/E and non-TSO/E). Execs that run in TSO/E can use several TSO/E external functions, commands, and programming services that are not available to execs that run in a non-TSO/E address space. "Writing execs that run in the TSO/E address space" on page 191 describes writing execs for TSO/E.

If you write a REXX exec that will run in a non-TSO/E address space, you can use the following in the exec:

- All keyword instructions that are described in Chapter 3, "Keyword instructions," on page 41
- All built-in functions that are described in Chapter 4, "Functions," on page 73.
- The TSO/E external functions SETLANG and STORAGE. See "TSO/E external functions" on page 109 for more information.
- The following TSO/E REXX commands:
  - MAKEBUF - to create a buffer on the data stack
  - DROPBUF - to drop (discard) a buffer that was previously created on the data stack with the MAKEBUF command
  - NEWSTACK - to create a new data stack and effectively isolate the current data stack that the exec is using
  - DELSTACK - to delete the most current data stack that was created with the NEWSTACK command
  - QBUF - to query how many buffers are currently on the active data stack
  - QELEM - to query how many elements are on the data stack above the most recently created buffer
  - QSTACK - to query the number of data stacks that are currently in existence
  - EXECIO - to read data from and write data to data sets. Using EXECIO, you can read data from and write data to the data stack or stem variables.
  - TS (Trace Start) - to start tracing REXX execs. Tracing lets you control exec processing and debug problems.
  - TE (Trace End) - to end tracing of REXX execs

  - SUBCOM - to determine whether a particular host command environment is available for the processing of host commands.

  The commands are described in Chapter 10, “TSO/E REXX commands,” on page 201.

- Invoking an exec

  You can invoke another REXX exec from an exec using the following instructions (the examples assume that the current host command environment is MVS):

  ```
  "execname p1 p2 ..."

  "EX execname p1 p2 ..."

  "EXEC execname p1 p2 ..."
  ```

  See “Commands to external environments” on page 22 about using host commands in a REXX exec.

- Linking to and attaching programs

  You can use the LINK, LINKMVS, and LINKPGM host command environments to link to unauthorized programs. For example:

  ```
  ADDRESS LINK "program p1 p2 ..."
  ```

  You can use the ATTACH, ATTCHMVS, and ATTCHPGM host command environments to attach unauthorized programs. For example:

  ```
  ADDRESS ATTACH "program p1 p2 ..."
  ```

  For more information about linking to and attaching programs, see “Host command environments for linking to and attaching programs” on page 30.

- TSO/E REXX programming services.

  In any address space, you can use the REXX programming services, such as IRXEXEC and IRXJCL, IRXEXCOM, and IRXIC. The services are described in Chapter 12, “TSO/E REXX programming services,” on page 239.

Running an exec in a Non-TSO/E address space

You can invoke a REXX exec in a non-TSO/E address space using the IRXJCL and IRXEXEC routines, which are programming interfaces to the language processor.

To execute an exec in MVS batch, use the IRXJCL routine. In the JCL, specify IRXJCL as the program name (PGM=) on the JCL EXEC statement. On the EXEC statement, specify the member name of the exec and the argument in the PARM field. Specify the name of the data set that contains the member on a DD statement. For example:

```
//STEP1    EXEC PGM=IRXJCL,PARM='PAYEXEC week hours'
//SYSEXEC  DD DSN=USERID.REXX.EXEC,DISP=SHR
```

You can also invoke IRXJCL from a program (for example, a PL/I program) to invoke a REXX exec.

You can invoke the IRXEXEC routine from a program to invoke a REXX exec. “Exec processing routines - IRXJCL and IRXEXEC” on page 245 describes IRXJCL and IRXEXEC in more detail and provides several examples.

If you want to invoke an exec from another exec that is running in a non-TSO/E address space, use one of the following instructions (the examples assume that the current host command environment is *not* MVS):

```
ADDRESS MVS "execname p1 p2 ..."

ADDRESS MVS "EX execname p1 p2 ..."

ADDRESS MVS "EXEC execname p1 p2 ..."
```

See “Host commands and host command environments” on page 23 for more information about the different environments for issuing host commands.

Writing execs that run in the TSO/E address space

If you write a REXX exec that will run in the TSO/E address space, there are additional TSO/E external functions and TSO/E commands and services you can use that are not available to execs that run in a non-TSO/E address space. For execs that run in the TSO/E address space, you can use the following:

- All keyword instructions that are described in Chapter 3, “Keyword instructions,” on page 41.
- All built-in functions that are described in Chapter 4, “Functions,” on page 73.
- All of the TSO/E external functions, which are described in “TSO/E external functions” on page 109.

  You can use the SETLANG and STORAGE external functions in execs that run in any address space (TSO/E and non-TSO/E). However, you can use the other TSO/E external functions only in execs that run in the TSO/E address space.
- The following TSO/E REXX commands:
  - MAKEBUF - to create a buffer on the data stack
  - DROPBUF - to drop (discard) a buffer that was previously created on the data stack with the MAKEBUF command
  - NEWSTACK - to create a new data stack and effectively isolate the current data stack that the exec is using
  - DELSTACK - to delete the most current data stack that was created with the NEWSTACK command
  - QBUF - to query how many buffers are currently on the active data stack
  - QELEM - to query how many elements are on the data stack above the most recently created buffer
  - QSTACK - to query the number of data stacks that are currently in existence
  - EXECIO - to read data from and write data to data sets. Using EXECIO, you can read data from and write data to the data stack or stem variables.
  - SUBCOM - to determine whether a particular host command environment is available for the processing of host commands
  - EXECUTIL - to change various characteristics that control how a REXX exec is processed. You can use EXECUTIL in an exec or CLIST, and from TSO/E READY mode and ISPF.
  - Immediate commands, which are:
    - HE (Halt Execution) - halt execution of the exec
    - HI (Halt Interpretation) - halt interpretation of the exec
    - TS (Trace Start) - start tracing of the exec
    - TE (Trace End) - end tracing of the exec
    - HT (Halt Typing) - suppress terminal output that the exec generates
    - RT (Resume Typing) - resume terminal output that was previously suppressed.

    You can use the TS and TE immediate commands in a REXX exec to start and end tracing. You can use any of the immediate commands if an exec is running in TSO/E and you press the attention interruption key. When you enter attention mode, you can enter an immediate command. The commands are described in Chapter 10, “TSO/E REXX commands,” on page 201.
- Invoking an exec

  You can invoke another REXX exec using the TSO/E EXEC command processor. For more information about the EXEC command, see *z/OS TSO/E Command Reference*.
- Linking to and attaching programs

  You can use the LINK, LINKMVS, and LINKPGM host command environments to link to unauthorized programs. For example:

```
ADDRESS LINK "program p1 p2 ..."
```

You can use the ATTACH, ATTCHMVS, and ATTCHPGM host command environments to attach unauthorized programs. For example:

```
ADDRESS ATTACH "program p1 p2 ..."
```

For more information about linking to and attaching programs, see "Host command environments for linking to and attaching programs" on page 30.

- Interactive System Productivity Facility (ISPF)

  You can invoke REXX execs from ISPF. You can also write ISPF dialogs in the REXX programming language. If an exec runs in ISPF, it can use ISPF services that are not available to execs that are invoked from TSO/E READY mode. In an exec, you can use the ISPEXEC and ISREDIT host command environments to use ISPF services. For example, to use the ISPF SELECT service, use:

  ```
  ADDRESS ISPEXEC 'SELECT service'
  ```

  You can use ISPF services only after ISPF has been invoked.
- TSO/E commands

  You can use any TSO/E command in a REXX exec that runs in the TSO/E address space. That is, from ADDRESS TSO, you can issue any unauthorized and authorized TSO/E command. For example, the exec can issue the ALLOCATE, TEST, PRINTDS, FREE, SEND, and LISTBC commands. *z/OS TSO/E Command Reference* and *z/OS TSO/E System Programming Command Reference* describe the syntax of TSO/E commands.
- TSO/E programming services

  If your REXX exec runs in the TSO/E address space, you can use various TSO/E service routines. For example, your exec can call a module that invokes a TSO/E programming service, such as the parse service routine (IKJPARS); TSO/E I/O service routines, such as PUTLINE and PUTGET; message handling routine (IKJEFF02); and the dynamic allocation interface routine (DAIR). These TSO/E programming services are described in *z/OS TSO/E Programming Services*.
- TSO/E REXX programming services

  In any address space, you can use the TSO/E REXX programming services, such as IRXEXEC and IRXJCL, IRXEXCOM, and IRXIC. The services are described in Chapter 12, "TSO/E REXX programming services," on page 239.
- Interaction with CLISTs.

  In TSO/E, REXX execs can invoke CLISTs and can also be invoked by CLISTs. CLIST is a command language and is described in *z/OS TSO/E CLISTs*.

Running an exec in the TSO/E address space

You can invoke a REXX exec in the TSO/E address space in several ways. To invoke an exec in TSO/E foreground, use the TSO/E EXEC command processor to either implicitly or explicitly invoke the exec. To invoke an exec explicitly, use the EXEC command, specifying the name of the data set and member to be executed. For example:

```
EXEC 'USERID.MYREXX.EXEC(TEST4)'EXEC
```

To invoke an exec implicitly by just specifying the member name of an exec, place the exec in a data set allocated to your SYSEXEC DD or SYSPROC DD. SYSEXEC contains only REXX execs, but SYSPROC can contain both REXX execs and CLISTs. To distinguish REXX execs from CLISTs, execs that are placed in SYSPROC should always start with a comment with the word "REXX" in line one, such as : /* REXX */. So if TEST1 is a REXX exec in a SYSEXEC or SYSPROC data set, you might invoke it implicitly with:

```
%TEST1
```

For more information about how to invoke an exec in TSO/E foreground, see *z/OS TSO/E REXX User's Guide*.

You can run a REXX exec in TSO/E background. In the JCL, specify IKJEFT01 as the program name (PGM=) on the JCL EXEC statement. On the EXEC statement, specify the member name of the exec and any arguments in the PARM field. For example, to execute an exec that is called TEST4 that is in data set USERID.MYREXX.EXEC, use the following JCL:

```
//TSOBATCH  EXEC PGM=IKJEFT01,DYNAMNBR=30,REGION=4096K,PARM='TEST4'
//SYSEXEC   DD   DSN=USERID.MYREXX.EXEC,DISP=SHR
```

You can also invoke an exec implicitly or explicitly in the input stream of the SYSTSIN DD statement.

```
//TSOBATCH  EXEC PGM=IKJEFT01,DYNAMNBR=30,REGION=4096K
//SYSEXEC   DD   DSN=USERID.MYREXX.EXEC,DISP=SHR
//SYSTSPRT  DD   SYSOUT=A
//SYSTSIN   DD   *
  %TEST4
     or
  EXEC           'USERID.MYREXX.EXEC(TEST4)'EXEC
/*
//
```

See *z/OS TSO/E REXX User's Guide* for more information about invoking execs.

From a program that is written in a high-level programming language, you can use the TSO service facility to invoke the TSO/E EXEC command to process a REXX exec. *z/OS TSO/E Programming Services* describes the TSO service facility in detail.

You can also invoke a REXX exec from an application program by using the exec processing routines IRXJCL and IRXEXEC. Although IRXJCL and IRXEXEC are primarily used in non-TSO/E address spaces, they are programming interfaces to the language processor that you can use to run an exec in any address space, including TSO/E. For example, in an assembler or PL/I program, you could invoke IRXJCL or IRXEXEC to process a REXX exec.

The IRXEXEC routine gives you more flexibility in processing an exec. For example, if you want to preload an exec in storage and then process the preloaded exec, you can use IRXEXEC. "Exec processing routines - IRXJCL and IRXEXEC" on page 245 describes the IRXJCL and IRXEXEC interfaces in detail.

Note: You cannot invoke a REXX exec as authorized in either the foreground or the background.

Summary of writing execs for different address spaces

Table 13 on page 193 summarizes the REXX keyword instructions, built- in functions, TSO/E external functions, TSO/E REXX commands, and other services you can use for execs that run in TSO/E and non-TSO/E address spaces. An *X* in the TSO/E or non-TSO/E columns indicates that the entry can be used in REXX execs that run in that address space.

You can use the TSO/E environment service, IKJTSOEV, to create a TSO/E environment in a non-TSO/E address space. If you run a REXX exec in the TSO/E environment you created, the exec can contain TSO/E commands, external functions, and services that an exec running in a TSO/E address space can use. For more information about the TSO/E environment service and the different considerations for running REXX execs within the environment, see *z/OS TSO/E Programming Services*.

Table 13. Summary of using instructions, functions, commands, and services

Instruction, function, command, service	**TSO/E**	**Non-TSO/E**
Keyword instructions (see Chapter 3, "Keyword instructions," on page 41)	X	X
Built-in functions (see "Built-in functions" on page 77)	X	X
TSO/E external functions (see "TSO/E external functions" on page 109)		
GETMSG	X	

Table 13. Summary of using instructions, functions, commands, and services (continued)

Instruction, function, command, service	TSO/E	Non-TSO/E
LISTDSI	X	
MSG	X	
OUTTRAP	X	
PROMPT	X	
SETLANG	X	X
STORAGE	X	X
SYSDSN	X	
SYSVAR	X	
TSO/E REXX commands (see Chapter 10, “TSO/E REXX commands,” on page 201)		
DELSTACK	X	X
DROPBUF	X	X
EXECIO	X	X
EXECUTIL	X	
HE (from attention mode only)	X	
HI (from attention mode only)	X	
HT (from attention mode only)	X	
MAKEBUF	X	X
NEWSTACK	X	X
QBUF	X	X
QELEM	X	X
QSTACK	X	X
RT (from attention mode only)	X	
SUBCOM	X	X
TE	X	X
TS	X	X
Miscellaneous services		
Invoking another exec	X	X
Linking to programs	X	X
Attaching programs	X	X
ISPF services	X	
TSO/E commands, such as ALLOCATE and PRINTDS	X	
TSO/E service routines, such as DAIR and IKJPARS	X	
TSO/E REXX programming services, such as IRXJCL, IRXEXEC, and IRXEXCOM (see Chapter 12, “TSO/E REXX programming services,” on page 239)	X	X

Table 13. Summary of using instructions, functions, commands, and services (continued)

Instruction, function, command, service	TSO/E	Non-TSO/E
Interacting with TSO/E CLISTs	X	
Issuing MVS system and subsystem commands during an extended MCS console session	X	
SAA CPI Communications calls	X	X
APPC/MVS calls	X	X

Chapter 9. Reserved keywords, special variables, and command names

You can use keywords as ordinary symbols in many situations where there is no ambiguity. The precise rules are given here.

There are three special variables: RC, RESULT, and SIGL.

TSO/E provides several TSO/E REXX commands whose names are reserved.

This chapter describes the reserved keywords, special variables, and reserved command names.

Reserved keywords

The free syntax of REXX implies that some symbols are reserved for the language processor environment's use in certain contexts.

Within particular instructions, some symbols may be reserved to separate the parts of the instruction. These symbols are referred to as keywords. Examples of REXX keywords are the WHILE in a DO instruction, and the THEN (which acts as a clause terminator in this case) following an IF or WHEN clause.

Apart from these cases, only simple symbols that are the first token in a clause and that are not followed by an "=" or ":" are checked to see if they are instruction keywords. You can use the symbols freely elsewhere in clauses without their being taken to be keywords.

It is not, however, suggested for users to execute host commands or subcommands with the same name as REXX keywords (QUEUE, for example). This can create problems for programmers whose REXX programs might be used for some time and in circumstances outside their control, and who want to make the program absolutely "watertight."

In this case, a REXX program may be written with (at least) the first words in command lines enclosed in quotation marks.

Example:

```
'LISTDS' ds_name
```

This also has the advantage of being more efficient, and with this style, you can use the SIGNAL ON NOVALUE condition to check the integrity of an exec.

In TSO/E, single quotation marks are often used in TSO/E commands, for example, to enclose the name of a fully qualified data set. In any REXX execs that run in TSO/E, you may want to enclose an entire host command in double quotation marks. This ensures that the language processor processes the expression as a host command. For example:

```
"ALLOCATE DA('prefix.proga.exec') FILE(SYSEXEC) SHR REUSE"
```

Special variables

There are three special variables that the language processor can set automatically:

RC

is set to the return code from any executed host command (or subcommand). Following the SIGNAL events SYNTAX, ERROR, and FAILURE, RC is set to the code appropriate to the event: the syntax error number or the command return code. RC is unchanged following a NOVALUE or HALT event.

Note: Host commands issued manually from debug mode do not cause the value of RC to change.

The special variable RC can also be set to a -3 if the host command could not be found. See "Host commands and host command environments" on page 23 for information about issuing commands from an exec.

The TSO/E REXX commands also return a value in the special variable RC. Some of the commands return the result from the command. For example, the QBUF command returns the number of buffers currently on the data stack in the special variable RC. The commands are described in Chapter 10, "TSO/E REXX commands," on page 201.

RESULT
is set by a RETURN instruction in a subroutine that has been called, if the RETURN instruction specifies an expression. If the RETURN instruction has no expression, RESULT is dropped (becomes uninitialized.)

SIGL
contains the line number of the clause currently executing when the last transfer of control to a label took place. (A SIGNAL, a CALL, an internal function invocation, or a trapped error condition could cause this.)

None of these variables has an initial value. You can alter them, just as with any other variable, and they can be accessed using the variable access routine IRXEXCOM (see "Variable access routine - IRXEXCOM" on page 274). The PROCEDURE and DROP instructions also affect these variables in the usual way.

Certain other information is always available to a REXX program. This includes the name by which the program was invoked and the source of the program (which is available using the PARSE SOURCE instruction—see "PARSE" on page 57). The data that PARSE SOURCE returns is:

1. The character string TSO
2. The call type (command, function, or subroutine)
3. Name of the exec in uppercase
4. Name of the DD from which the exec was loaded, if known
5. Name of the data set from which the exec was loaded, if known
6. Name of the exec as invoked (that is, not folded to uppercase)
7. Initial (default) host command environment
8. Name of the address space in uppercase
9. Eight character user token

In addition, PARSE VERSION (see "PARSE" on page 57) makes available the version and date of the language processor code that is running. The built-in functions TRACE and ADDRESS return the current trace setting and host command environment name, respectively.

Finally, you can obtain the current settings of the NUMERIC function using the DIGITS, FORM, and FUZZ built-in functions.

Reserved command names

TSO/E provides TSO/E REXX commands that you can use for REXX processing. The commands are described in Chapter 10, "TSO/E REXX commands," on page 201. The names of these commands are reserved for use by TSO/E; do not use these names for names of your REXX execs, CLISTs, or load modules. The reserved command names are:

- DELSTACK
- DROPBUF
- EXECIO
- EXECUTIL
- HE
- HI

- HT
- MAKEBUF
- NEWSTACK
- QBUF
- QELEM
- QSTACK
- RT
- SUBCOM
- TE
- TS

Chapter 10. TSO/E REXX commands

TSO/E provides TSO/E REXX commands to perform different services, such as I/O and data stack requests. The TSO/E REXX commands are not the same as TSO/E command processors, such as ALLOCATE and PRINTDS. In general, you can only use these commands in REXX execs (in any address space), not in CLISTs or from TSO/E READY mode. The exceptions are the EXECUTIL command and the *immediate commands* HE, HI, HT, RT, TE, and TS.

You can use the EXECUTIL command in the TSO/E address space only. In general, you can use EXECUTIL in an exec or a CLIST, from TSO/E READY mode, or from ISPF. See "EXECUTIL" on page 216 for the description of the EXECUTIL command that describes the different operands and any exceptions about using them.

You can use the TS (Trace Start) and TE (Trace End) immediate commands in an exec that runs in any address space. In the TSO/E address space, you can use any of the immediate commands (HE, HI, HT, RT, TE, and TS) if you are executing a REXX exec and press the attention interrupt key. When you enter attention mode, you can enter one of the immediate commands.

The TSO/E REXX commands perform services, such as:

- Controlling I/O processing of information to and from data sets (EXECIO)
- Performing data stack services (MAKEBUF, DROPBUF, QBUF, QELEM, NEWSTACK, DELSTACK, QSTACK)
- Changing characteristics that control the execution of an exec (EXECUTIL and the immediate commands)
- Checking for the existence of a host command environment (SUBCOM).

Restriction: The names of the TSO/E REXX commands are reserved for use by TSO/E. You should not use these names for names of your REXX execs, CLISTs, or load modules.

Environment Customization Considerations: If you customize REXX processing using the initialization routine IRXINIT, you can initialize a language processor environment that is not integrated into TSO/E (see "Types of environments - integrated and not integrated into TSO/E" on page 316). Most of the TSO/E REXX commands can be used in any type of language processor environment. The EXECUTIL command can be used only if the environment is integrated into TSO/E. You can use the immediate commands from attention mode only if the environment is integrated into TSO/E. You can use the TS and TE immediate commands in a REXX exec that executes in any type of language processor environment (integrated or not integrated into TSO/E). Chapter 13, "TSO/E REXX customizing services," on page 305 describes customization and language processor environments in more detail.

In this chapter, examples are provided that show how to use the TSO/E REXX commands. The examples may include data set names. When an example includes a data set name that is not enclosed in either single quotation marks or double quotation marks, the prefix is added to the beginning of the data set name to form the final, fully qualified data set name. If the data set name is enclosed within quotation marks (single or double), it is considered to be full qualified, and the data set name is used exactly as specified. In the examples, the user ID is the prefix.

DELSTACK

```
►► DELSTACK ►◄
```

deletes the most recently created data stack that was created by the NEWSTACK command, and all elements on it. If a new data stack was not created, DELSTACK removes all the elements from the original data stack.

The DELSTACK command can be used in REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

The exec that creates a new data stack with the NEWSTACK command can delete the data stack with the DELSTACK command, or an external function or subroutine that is written in REXX and that is called by that exec can issue a DELSTACK command to delete the data stack.

Examples

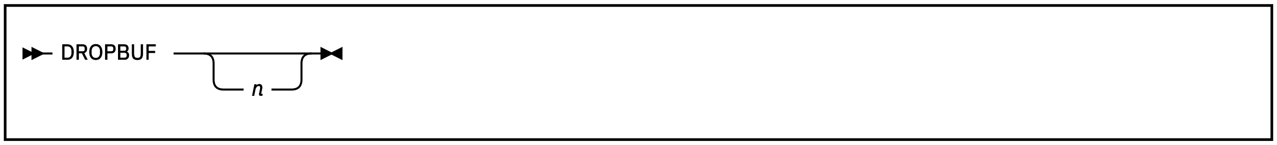
1. To create a new data stack for a called routine and delete the data stack when the routine returns, use the NEWSTACK and DELSTACK commands as follows:

```
   ⋮
"NEWSTACK"   /* data stack 2 created */
CALL sub1
"DELSTACK"   /* data stack 2 deleted */
   ⋮
EXIT

sub1:
PUSH ...
QUEUE ...
PULL ...
RETURN
```

2. After creating multiple new data stacks, to find out how many data stacks were created and delete all but the original data stack, use the NEWSTACK, QSTACK, and DELSTACK commands as follows:

```
"NEWSTACK"   /* data stack 2 created */
   ⋮
"NEWSTACK"   /* data stack 3 created */
   ⋮
"NEWSTACK"   /* data stack 4 created */
"QSTACK"
times = RC-1 /* set times to the number of new data stacks created */
DO times     /* delete all but the original data stack */
  "DELSTACK" /* delete one data stack */
END
```

DROPBUF



removes the most recently created data stack buffer that was created with the MAKEBUF command, and all elements on the data stack in the buffer. To remove a specific data stack buffer and all buffers created after it, issue the DROPBUF command with the number (n) of the buffer.

The DROPBUF command can be issued from REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

Operand: The operand for the DROPBUF command is:

n

specifies the number of the first data stack buffer you want to drop. DROPBUF removes the specified buffer and all buffers created after it. Any elements that were placed on the data stack after the specified buffer was created are also removed. If *n* is not specified, only the most recently created buffer and its elements are removed.

The data stack initially contains one buffer, which is known as buffer 0. This buffer is never removed, as it is not created by MAKEBUF. If you issue DROPBUF 0, all buffers that were created on the data stack with the MAKEBUF command and all elements that were put on the data stack are removed. DROPBUF 0 effectively clears the data stack including the elements on buffer 0.

If processing was not successful, the DROPBUF command sets one of the following return codes in the REXX special variable RC.

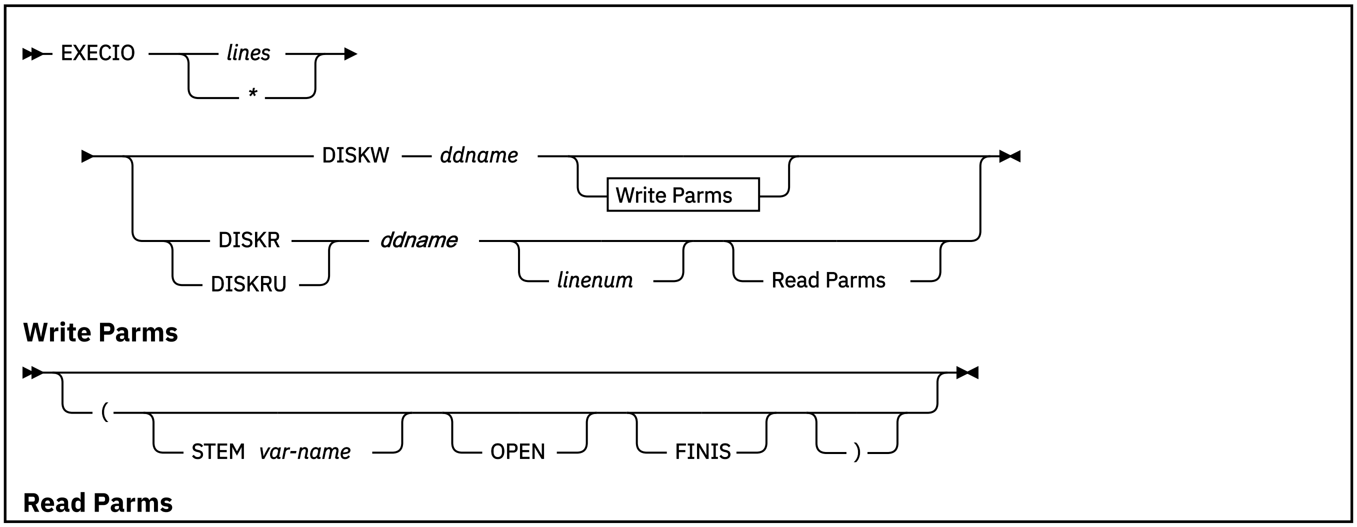
Return code	Meaning
0	DROPBUF was successful.
1	An invalid number *n* was specified. For example, *n* was A1.
2	The specified buffer does not exist. For example, you get a return code of 2 if you try to drop a buffer that does not exist.

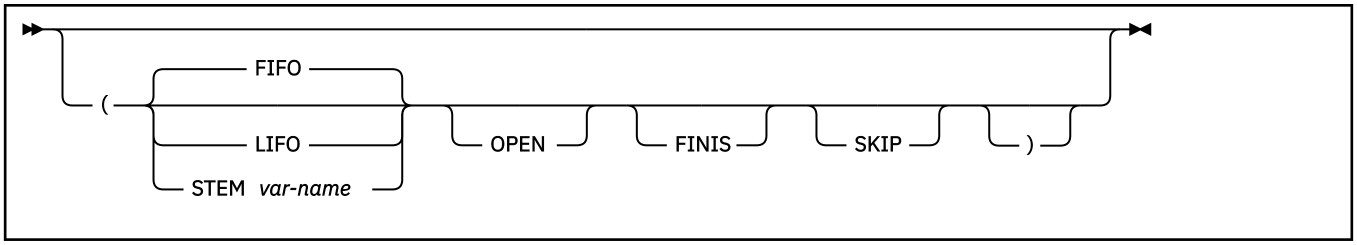
Example

A subroutine (sub2) in a REXX exec (execc) issues the MAKEBUF command to create four buffers. Before the subroutine returns, it removes buffers two and above and all elements within the buffers.

```
/*  REXX program  */
execc:
       ⋮
     CALL sub2
       ⋮

exit
sub2:
   "MAKEBUF"      /* buffer 1 created */
   QUEUE A
   "MAKEBUF"      /* buffer 2 created */
   QUEUE B
   QUEUE C
   "MAKEBUF"      /* buffer 3 created */
   QUEUE D
   "MAKEBUF"      /* buffer 4 created */
   QUEUE E
   QUEUE F
       ⋮
   "DROPBUF 2"    /* buffers 2 and above deleted */
   RETURN
```

EXECIO





Controls the input and output (I/O) of information to and from a data set. Information can be read from a data set to the data stack for serialized processing or to a list of variables for random processing. Information from the data stack or a list of variables can be written to a data set.

The EXECIO command can be used in REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

You can use the EXECIO command to do various types of I/O tasks, such as copy information to and from a data set to add, delete, or update the information.

Note: The EXECIO command does not support I/O on files that are allocated to data sets with track overflow. However, if a data set is allocated with track overflow, that attribute is reset by DFSMS before any I/O is performed. The user might see a message like:

```
    IEC137I TRACK OVERFLOW RESET FOR <ddname>
```

and then the I/O is performed.

An I/O data set must be either sequential or a single member of a PDS. Before the EXECIO command can perform I/O to or from the data set, the data set must be allocated to a file that is specified on the EXECIO command. The EXECIO command does not perform the allocation.

Both fixed and fixed standard record formats with or without blocking are supported. Also, variable record formats, including spanned records, with or without blocking are supported. In addition, data sets with undefined record formats are supported. That is, all of the following record formats (RECFM) are allowed:

```
     F, FB, FS, FBS, V, VB, VS, VBS, and U
```

The data set might also optionally include ANSI or MACHINE carriage control characters. With carriage control included in the RECFM, the following additional record formats are also supported:

```
     FA, FM, FBA, FBM, FSA, FSM, FBSA, FBSM,
     VA, VM, VBA, VBM, VSA, VSM, VBSA, VBSM
```

EXECIO does not support read for update (DISKRU) for files that are allocated with RECFM=VS or VBS, nor for files allocated to a member of a PDSE Program Library. EXECIO does not support write (DISKW) for files that are allocated to a member of a PDSE Program Library.

When performing I/O with a system data set that is available to multiple users, allocate the data set as OLD, before issuing the EXECIO command, to have exclusive use of the data set.

When you use EXECIO, you must ensure that you use quotation marks around any operands, such as DISKW, STEM, FINIS, or LIFO. Using quotation marks prevents the possibility of the operands being substituted as variables. For example, if you assign the variable *stem* to a value in the exec and then issue EXECIO with the STEM option, if STEM is not enclosed in quotation marks, it is substituted with its assigned value.

Operands for Reading from a Data Set: The operands for the EXECIO command to read from a data set are as follows:

lines
The number of lines to be processed. This operand can be a specific decimal number or an arbitrary number that is indicated by *. When the operand is * and EXECIO is reading from a data set, input is read until EXECIO reaches the end of the data set.

If you specify a value of zero, no I/O operations are performed unless you also specify either OPEN, FINIS, or both OPEN and FINIS.

- If you specify OPEN and the data set is closed, EXECIO opens the data set but does not read any lines. If you specify OPEN and the data set is open, EXECIO does not read any lines.

  In either case, if you also specify a nonzero value for the *linenum* operand, EXECIO sets the current record number to the record number indicated by the *linenum* operand.

  Note: By default, when a file is opened, the current record number is set to the first record (record 1). The current record number is the number of the next record EXECIO will read. However, if you use a nonzero *linenum* value with the OPEN operand, EXECIO sets the current record number to the record number indicated by *linenum*.
- If you specify FINIS and the data set is open, EXECIO does not read any lines, but EXECIO closes the data set. If you specify FINIS and the data set is not already opened, EXECIO does not open the data set and then close it.
- If you specify both OPEN and FINIS, EXECIO processes the OPEN first as described above. EXECIO then processes the FINIS as described above.

DISKR

Opens a data set for input (if it is not already open) and reads the specified number of lines from the data set and places them on the data stack. If the STEM operand is specified, the lines are placed in a list of variables instead of on the data stack.

While a data set is open for input, you cannot write information back to the same data set.

The data set is not automatically closed unless:

- The task, under which the data set was opened, ends.
- The last language processor environment associated with the task, under which the data set was opened, is terminated (see “Initialization and termination of a language processor environment” on page 306 for information about language processor environments)

DISKRU

Opens a data set for update (if it is not already open) and reads the specified number of lines from the data set and places them on the data stack. If the STEM operand is specified, the lines are placed in a list of variables instead of on the data stack.

While a data set is open for update, the last record read can be changed and then written back to the data set with a corresponding EXECIO DISKW command. Typically, you open a data set for update when you want to modify information in the data set.

However, note that data sets with spanned record format (that is, VS or VBS) cannot be opened in update mode.

The data set is not automatically closed unless:

- The task, under which the data set was opened, ends.
- The last language processor environment associated with the task, under which the data set was opened, is terminated.

After a data set is open for update (by issuing a DISKRU as the first operation against the data set), you can use either DISKR or DISKRU to fetch subsequent records for update.

ddname

The name of the file to which the sequential data set or member of the PDS was allocated. You must allocate the file before you can issue EXECIO. For example, you can allocate a file using the ALLOCATE command in the TSO/E address space only or a JCL DD statement.

linenum

The line number in the data set at which EXECIO is to begin reading. When a data set is closed and reopened as a result of specifying a record number earlier than the current record number, the file is open for:

- Input, if DISKR is specified.
- Update, if DISKRU is specified.

When a data set is open for input or update, the current record number is the number of the next record to be read. When *linenum* specifies a record number earlier than the current record number in an open data set, the data set must be closed and reopened to reposition the current record number at *linenum*. When this situation occurs and the data set was not opened at the same task level as that of the executing exec, attempting to close the data set at a different task level results in an EXECIO error. The *linenum* operand must not be used in this case.

Specifying a value of zero for *linenum* is equivalent to not specifying the *linenum* operand. In either case, EXECIO begins reading the file as follows:

- If the file was already opened, EXECIO begins reading with the line following the last line that was read.
- If the file was opened, EXECIO begins reading with the first line of the file.

FINIS

Close the data set after the EXECIO command completes. A data set can be closed only if it was opened at the same task level as the exec issuing the EXECIO command.

You can use FINIS with a *lines* value of 0 to have EXECIO close an open data set without first reading a record.

Because the EXEC command (when issued from TSO/E READY mode) is attached by the TSO/E terminal monitor program (TMP), data sets that are opened by a REXX exec are typically closed automatically when the top level exec ends. However, good programming practice would be to explicitly close all data sets when finished with them.

OPEN

Opens the specified data set if it is not already open. You can use OPEN with a *lines* value of 0 to have EXECIO do one of the following:

- Open a data set without reading any records.
- Set the current record number (that is, the number of the next record EXECIO will read) to the record number indicated by the *linenum* operand, if you specify a value for *linenum*.

STEM *var-name*

The stem of the set of variables into which information is to be placed. To place information in compound variables, which allow for easy indexing, the *var-name* should end with a period. For example,

```
MYVAR.
```

When *var-name* does not end with a period, the variable names are appended with numbers and can be accessed in a loop such as:

```
   "EXECIO * DISKR MYINDD (FINIS STEM MYVAR"
   DO i = 1 to MYVAR0
      this_line = VALUE('MYVAR'||i)
   END
```

In the first example above, the list of compound variables has the stem `MYVAR.` and lines of information (records) from the data set are placed in variables `MYVAR.1`, `MYVAR.2`, `MYVAR.3`, and so forth. The number of variables in the list is placed in `MYVAR.0`

Thus if 10 lines of information were read into the MYVAR variables, `MYVAR.0` contains the number 10, indicating that 10 records are read. Furthermore, `MYVAR.1` contains record 1, `MYVAR.2` contains record 2, and so forth, up to `MYVAR.10` which contains record 10. All stem variables beyond `MYVAR.10` (that is, `MYVAR.11`, `MYVAR.12`, and so on) are residual and contain the values that they held before entering the EXECIO command.

To avoid confusion whether a residual stem variable value is meaningful, you might want to clear the entire stem variable before entering the EXECIO command. To clear all compound variables whose names begin with that stem, you can either:

- Use the DROP instruction as follows to set all possible compound variables whose names begin with that stem to their uninitialized values:

  ```
  DROP MYVAR.
  ```

- Set all possible compound variables whose names begin with that stem to nulls as follows:

  ```
  MYVAR. = ''
  ```

See example “12” on page 213 that shows the usage of the EXECIO command with stem variables.

LIFO
Places information about the data stack in LIFO (last in first out) order.

FIFO
Places information about the data stack in FIFO (first in first out) order. FIFO is the default when LIFO, FIFO, and STEM are not specified.

SKIP
Reads the specified number of lines but does not place them on the data stack or in variables. When the number of lines is *, EXECIO skips to the end of the data set.

Operands for Writing to a Data Set: The operands for the EXECIO command that write to a data set are as follows:

lines
The number of lines to be written. This operand can be a specific decimal number or an arbitrary number indicated by *. If you specify a value of zero, or if you specify * and it is determined that there are no lines available to be written, no I/O operations are performed unless you also specify either OPEN, FINIS, or both OPEN and FINIS.

- If you specify OPEN and the data set is closed, EXECIO opens the data set but does not write any lines. If you specify OPEN and the data set is open, EXECIO does not write any lines.
- If you specify FINIS and the data set is open, EXECIO does not write any lines, but EXECIO closes the data set. If you specify FINIS and the data set is not already opened, EXECIO does not open the data set and then close it.
- If you specify both OPEN and FINIS, EXECIO processes the OPEN first as described above. EXECIO then processes the FINIS as described above.

When EXECIO writes an arbitrary number of lines from the data stack, it stops only when it reaches a null line. If there is no null line on the data stack in an interactive TSO/E address space, EXECIO waits for input from the terminal and stops only when it receives a null line. See note below.

When EXECIO writes an arbitrary number of lines from a list of compound variables, it stops when it reaches a null value or an uninitialized variable (one that displays its own name).

The 0th variable has no effect on controlling the number of lines that are written from variables.

Note: EXECIO running in TSO/E background or in a non-TSO/E address space has the same use of the data stack as an exec that runs in the TSO/E foreground. If an EXECIO * DISKW ... command is executing in the background or in a non-TSO/E address space and the data stack becomes empty before a null line is found (which would terminate EXECIO), EXECIO goes to the input stream as defined by the INDD field in the module name table (see “Module name table” on page 326). The system default is SYSTSIN. When end of file is reached, EXECIO ends.

DISKW
Opens a data set for output (if it is not already open) and writes the specified number of lines to the data set. The lines can be written from the data stack or, if the STEM operand is specified, from a list of variables.

You can use the DISKW operand to write information to a different data set from the one opened for input, or to update, one line at a time, the same data set opened for update.

When a data set is open for update, you can use DISKW to rewrite the last record read. The *lines* value should be 1 when doing an update. For *lines* values greater than 1, each write updates the same record.

The data set is not automatically closed unless:

- The task, under which the data set was opened, ends.
- The last language processor environment associated with the task, under which the data set was opened, is terminated.

Note:

1. The length of an updated line is set to the length of the line it replaces. When an updated line is longer than the line it replaces, information that extends beyond the replaced line is truncated. When information is shorter than the replaced line, the line is padded with blanks to attain the original line length.
2. When using EXECIO to write to more than one member of the same PDS, only one member of the PDS should be open at a time for output.
3. Do not use the MOD attribute when allocating a member of a PDS to which you want to append information. You can use MOD only when appending information to a sequential data set. To append information to a member of a PDS, rewrite the member with the additional records added.

ddname

The name of the file to which the sequential data set or member of the PDS was allocated. You must allocate the file before you issue the EXECIO command.

FINIS

Close the data set after the EXECIO command completes. A data set can be closed only if it was opened at the same task level as the exec issuing the EXECIO command.

You can use FINIS with a *lines* value of 0 to have EXECIO close an open data set without first writing a record.

Because the EXEC command (when issued from TSO/E READY mode) is attached by the TMP, data sets opened by a REXX exec are typically closed automatically when the top level exec ends. However, good programming practice would be to explicitly close all data sets when finished with them.

OPEN

Opens the specified data set if it is not already open. You can use OPEN with a *lines* value of 0 to have EXECIO open the data set without writing any records. If you specify OPEN with a lines value of * and it is determined that there are no lines to be written, the data set still opens.

STEM *var-name*

The stem of the list of variables from which information is to be written. To write information from compound variables, which allow for indexing, the *var-name* should end with a period, `MYVAR.`, for example. When three lines are written to the data set, they are taken from `MYVAR.1`, `MYVAR.2`, `MYVAR.3`. When * is specified as the number of lines to write, the EXECIO command stops writing information to the data set when it finds a null line or an uninitialized compound variable. In this case, if the list contained 10 compound variables, the EXECIO command stops at `MYVAR.11`.

The 0th variable has no effect on controlling the number of lines that are written from variables.

When *var-name* does not end with a period, the variable names must be appended with consecutive numbers, such as `MYVAR1`, `MYVAR2`, `MYVAR3`.

See example “12” on page 213 that shows the usage of the EXECIO command with stem variables.

Closing Data Sets: If you specify FINIS on the EXECIO command, the data set is closed after EXECIO completes processing. If you do not specify FINIS, the data set is closed when one of the following occurs:

- The task, under which the data set was opened, is terminated, or
- The last language processor environment associated with the task, under which the data set was opened, is terminated (even if the task itself is not terminated).

In general, if you use the TSO/E EXEC command to invoke a REXX exec, any data sets that the exec opens are closed when the top level exec completes. For example, suppose you are executing an exec (top level exec) that invokes another exec. The second exec uses EXECIO to open a data set and then returns control to the first exec without closing the data set. The data set is still open when the top level exec regains control. The top level exec can then read the same data set continuing from the point where the nested exec finished EXECIO processing. When the original exec (top level exec) ends, the data set is automatically closed.

Figure 13 on page 209 is an example of two execs that show how a data set remains open. The first (top level) exec, EXEC1, allocates a file and then calls EXEC2. The second exec (EXEC2) opens the file, reads the first three records, and then returns control to EXEC1. Note that EXEC2 does not specify FINIS on the EXECIO command, so the file remains open.

When the first exec EXEC1 regains control, it issues EXECIO and gets the fourth record because the file is still open. If EXEC2 had specified FINIS on the EXECIO command, EXEC1 would have read the first record. In the example, both execs run at the same task level.

```
                 FIRST EXEC ---- EXEC1

/*   REXX exec (EXEC1) invokes another exec (EXEC2) to open a         */
/*   file.  EXEC1 then continues reading the same file.               */
say 'Executing the first exec EXEC1'
"ALLOC FI(INPUTDD) DA(MYINPUT) SHR REUSE"   /* Allocate input file    */
/*                                                                    */
/*   Now invoke the second exec (EXEC2) to open the INPUTDD file.     */
/*   The exec uses a call to invoke the second exec.  You can         */
/*   also use the TSO/E EXEC command, which would have the            */
/*   same result.                                                     */
/*   If EXEC2 opens a file and does not close the file before         */
/*   returning control to EXEC1, the file remains open when           */
/*   control is returned to EXEC1.                                    */
/*                                                                    */
say 'Invoking the second exec EXEC2'
call exec2                            /* Call EXEC2 to open file      */
say 'Now back from the second exec EXEC2.  Issue another EXECIO.'
"EXECIO 1 DISKR INPUTDD (STEM X."     /* EXECIO reads record 4        */
say x.1
say 'Now close the file'
"EXECIO 0 DISKR INPUTDD (FINIS"      /* Close file so it can be freed */
"FREE FI(INPUTDD)"
EXIT 0


                 SECOND EXEC ---- EXEC2

/*   REXX exec (EXEC2) opens the file INPUTDD, reads 3 records, and   */
/*   then returns to the invoking exec (EXEC1).  The exec (EXEC2)     */
/*   returns control to EXEC1 without closing the INPUTDD file.       */
/*                                                                    */
say "Now in the second exec EXEC2"
DO I = 1 to 3                       /* Read & display first 3 records */
   "EXECIO 1 DISKR INPUTDD (STEM Y."
   say y.1
END
Say 'Leaving second exec EXEC2.  Three records were read from file.'
RETURN 0
```

Figure 13. Example of closing data sets with EXECIO

Return Codes: After the EXECIO command runs, it sets the REXX special variable RC to one of the following return codes:

Return code	Meaning
0	Normal completion of requested operation.
1	Data was truncated during DISKW operation.
2	End-of-file reached before the specified number of lines were read during a DISKR or DISKRU operation. This does not occur if * is used for number of lines because the remainder of the file is always read.
4	During a DISKR or DISKRU operation, an empty data set was found in a concatenation of data sets. The file was not successfully opened and no data was returned.
20	Severe error. EXECIO completed unsuccessfully and a message is issued.

Examples

1. This example copies an entire existing sequential data set named prefix.MY.INPUT into a member of an existing PDS named DEPT5.MEMO(MAR22), and uses the ddnames DATAIN and DATAOUT respectively.

```
"ALLOC DA(MY.INPUT) F(DATAIN) SHR REUSE"
"ALLOC DA('DEPT5.MEMO(MAR22)') F(DATAOUT) OLD"
"NEWSTACK" /* Create a new data stack for input only */

"EXECIO * DISKR DATAIN (FINIS"
QUEUE ''    /* Add a null line to indicate the end of information */
"EXECIO * DISKW DATAOUT (FINIS"

"DELSTACK" /* Delete the new data stack */
"FREE F(DATAIN DATAOUT)"
```

2. This example copies an arbitrary number of lines from existing sequential data set prefix.TOTAL.DATA into a list of compound variables with the stem DATA., and uses the ddname INPUTDD:

```
ARG lines
"ALLOC DA(TOTAL.DATA) F(INPUTDD) SHR REUSE"
"EXECIO" lines "DISKR INPUTDD (STEM DATA."
SAY data.0 'records were read.'
```

3. To update the second line in data set DEPT5.EMPLOYEE.LIST in file UPDATEDD, allocate the data set as OLD to guarantee exclusive update.

```
"ALLOC DA('DEPT5.EMPLOYEE.LIST') F(UPDATEDD) OLD"
"EXECIO 1 DISKRU UPDATEDD 2"
PULL line
PUSH 'Crandall, Amy        AMY       5500'
"EXECIO 1 DISKW UPDATEDD (FINIS"
"FREE F(UPDATEDD)"
```

4. The following example scans each line of a data set whose name and size is specified by the user. The user is given the option of changing each line as it appears. If there is no change to the line, the user presses the Enter key to indicate that there is no change. If there is a change to the line, the user types the entire line with the change and the new line is returned to the data set.

```
PARSE ARG name numlines   /* Get data set name and size from user */

"ALLOC DA("name") F(UPDATEDD) OLD"
eof = 'NO'                /* Initialize end-of-file flag */

DO i = 1 to numlines WHILE eof = no
  "EXECIO 1 DISKRU UPDATEDD "  /* Queue the next line on the stack */
  IF RC = 2 THEN               /* Return code indicates end-of-file */
     eof = 'YES'
  ELSE
    DO
      PARSE PULL line
      SAY 'Please make changes to the following line.'
      SAY 'If you have no changes, press ENTER.'
      SAY line
      PARSE PULL newline
      IF newline = '' THEN NOP
      ELSE
```

```
       DO
          PUSH newline
          "EXECIO 1 DISKW UPDATEDD"
       END
    END
END
"EXECIO 0 DISKW UPDATEDD (FINIS"
```

5. This example reads from the data set allocated to INDD to find the first occurrence of the string "Jones". Upper and lowercase distinctions are ignored. The example demonstrates how to read and search one record at a time. For better performance, you can read all records to the data stack or to a list of variables, search them, and then return the updated records.

```
done = 'no'

DO WHILE done = 'no'
  "EXECIO 1 DISKR INDD"
   IF RC = 0 THEN          /*  Record was read */
     DO
       PULL record
       lineno = lineno + 1   /*  Count the record */
       IF INDEX(record,'JONES') ¬= 0 THEN
         DO
           SAY 'Found in record' lineno
           done = 'yes'
           SAY 'Record = ' record
         END
       ELSE NOP
     END
   ELSE
     done = 'yes'
END
"EXECIO 0 DISKR INDD (FINIS"
EXIT 0
```

6. This exec copies records from data set prefix.MY.INPUT to the end of data set prefix.MY.OUTPUT. Neither data set has been allocated to a ddname. It assumes that the input data set has no null lines.

```
"ALLOC DA(MY.INPUT) F(INDD) SHR REUSE"
"ALLOC DA(MY.OUTPUT) F(OUTDD) MOD REUSE"

SAY 'Copying ...'

"EXECIO * DISKR INDD (FINIS"
QUEUE ''   /* Insert a null line at the end to indicate end of file */
"EXECIO * DISKW OUTDD (FINIS"

SAY 'Copy complete.'
"FREE F(INDD OUTDD)"

EXIT 0
```

7. This exec reads five records from the data set allocated to MYINDD starting with the third record. It strips trailing blanks from the records, and then writes any record that is longer than 20 characters. The file is not closed when the exec is finished.

```
"EXECIO 5 DISKR MYINDD 3"

DO i = 1 to 5
  PARSE PULL line
  stripline = STRIP(line,t)
  len = LENGTH(stripline)

  IF len > 20 THEN
    SAY 'Line' stripline 'is long.'
  ELSE NOP
END

/* The file is still open for processing */

EXIT 0
```

8. This exec reads the first 100 records (or until EOF) of the data set allocated to INVNTORY. Records are placed on the data stack in LIFO order. A message is issued that gives the result of the EXECIO operation.

```
eofflag = 2             /* Return code to indicate end of file */

"EXECIO 100 DISKR INVNTORY (LIFO"
return_code = RC

IF return_code = eofflag THEN
   SAY 'Premature end of file.'
ELSE
   SAY '100 Records read.'

EXIT return_code
```

9. This exec erases any existing data from the data set FRED.WORKSET.FILE by opening the data set and then closing it without writing any records. By doing this, EXECIO just writes an end-of-file marker, which erases any existing records in the data set.

   In this example, the data set from which you are erasing records must not be allocated with a disposition of MOD. If you allocate the data set with a disposition of MOD, the EXECIO OPEN followed by the EXECIO FINIS results in EXECIO just rewriting the existing end-of-file marker.

```
"ALLOCATE DA('FRED.WORKSET.FILE') F(OUTDD) OLD REUSE"

"EXECIO 0 DISKW OUTDD (OPEN"    /* Open the OUTDD file for writing,
                                   but do not write a record       */

"EXECIO 0 DISKW OUTDD (FINIS"   /* Close the OUTDD file.  This
                                   basically completes the erasing of
                                   any existing records from the
                                   OUTDD file.                     */
```

   Note that in this example, the `EXECIO … (OPEN` command followed by the `EXECIO … (FINIS` command is equivalent to:

```
"EXECIO 0 DISKW OUTDD (OPEN FINIS"
```

   Also, note that the above `EXECIO 0 DISKW` command is equivalent to using `EXECIO * DISKW` when there are no lines to write. That is, the above `EXECIO 0` is equivalent to:

```
push  ''           /* Place null on stack to terminate EXECIO*/
"EXECIO * DISKW OUTDD (OPEN FINIS"
```

10. This exec opens the data set MY.INVNTORY without reading any records. The exec then uses a main loop to read records from the data set and process the records.

```
"ALLOCATE DA('MY.INVNTORY') F(INDD) SHR REUSE"
"ALLOCATE DA('MY.AVAIL.FILE') F(OUTDD) OLD REUSE"

"EXECIO 0 DISKR INDD (OPEN"         /* Open INDD file for input, but
                                       do not read any records       */

eof = 'NO'                          /* Initialize end-of-file flag   */
avail_count = 0                     /* Initialize counter            */

DO WHILE eof = 'NO'                 /* Loop until the EOF of input
                                       file                          */
  "EXECIO 1 DISKR INDD (STEM LINE." /* Read a line                   */
  IF RC = 2 THEN                    /* If end of file is reached,    */
    eof = 'YES'                     /* set the end-of-file (eof)
                                       flag                          */
  ELSE                              /* Otherwise, a record is read   */
    DO
      IF INDEX(line.1,'AVAILABLE') ¬ = 0 THEN /* Look for records
                                                 marked "available"     */
        DO                          /* "Available" record found      */

          "EXECIO 1 DISKW OUTDD (STEM LINE."  /* Write record to "available"
                                                 file                   */
          avail_count = avail_count + 1 /* Increment "available"
                                           counter                   */
        END
    END
END

"EXECIO 0 DISKR INDD (FINIS"    /* Close INDD file that is currently
                                   open                               */
"EXECIO 0 DISKW OUTDD (FINIS"   /* Close OUTDD file if file is cur-
                                   rently open.  If the OUTDD file is
                                   not open, the EXECIO command has
                                   no effect.                         */

EXIT 0
```

11. This exec opens the data set MY.WRKFILE and sets the current record number to record 8 so that the next EXECIO DISKR command begins reading at the eighth record.

```
"ALLOC DA('MY.WRKFILE') F(INDD) SHR REUSE"

"EXECIO 0 DISKR INDD 8 (OPEN" /* Open INDD file for input and set
                                 current record number to 8.       */

CALL READ_NEXT_RECORD         /* Call subroutine to read record on
                                 to the data stack.  The next
                                 record EXECIO reads is record 8
                                 because the previous EXEC IO set
                                 the current record number to 8.   */
⋮

"EXECIO 0 DISKR INDD (FINIS"  /* Close the INDD file.              */
```

12. This exec uses EXECIO to successively append the records from 'sample1.data' and then from 'sample2.data' to the end of the data set 'all.sample.data'. It illustrates the effect of residual data in STEM variables. Data set 'sample1.data' contains 20 records. Data set 'sample2.data' contains 10 records.

```
"ALLOC FI(MYINDD1) DA('SAMPLE1.DATA') SHR REUSE"  /* input file 1    */
"ALLOC FI(MYINDD2) DA('SAMPLE2.DATA') SHR REUSE"  /* input file 2    */

"ALLOC FI(MYOUTDD) DA('ALL.SAMPLE.DATA') MOD REUSE" /* output append
                                  file                               */

/*******************************************************************/
/* Read all records from 'sample1.data' and append them to the     */
/* end of 'all.sample.data'.                                       */
/*******************************************************************/

exec_RC = 0                    /* Initialize exec return code      */

"EXECIO * DISKR MYINDD1 (STEM NEWVAR. FINIS" /* Read all records   */

if rc = 0 then                 /* If read was successful           */
  do
  /*****************************************************************/
  /* At this point, newvar.0 should be 20, indicating 20 records   */
  /* have been read. Stem variables newvar.1, newvar.2, ... through*/
  /* newvar.20 will contain the 20 records that were read.         */
  /*****************************************************************/

    say "-----------------------------------------------------"
    say newvar.0 "records have been read from 'sample1.data': "
    say
    do i = 1 to newvar.0       /* Loop through all records         */
      say newvar.i             /* Display the ith record           */
    end

    "EXECIO" newvar.0 "DISKW MYOUTDD (STEM NEWVAR." /* Write exactly
                                  the number of records read       */
    if rc = 0 then             /* If write was successful          */
      do
        say
        say newvar.0 "records were written to 'all.sample.data'"
      end
    else
      do
        exec_RC = RC           /* Save exec return code            */
        say
        say "Error during 1st EXECIO ... DISKW, return code is " RC
        say
      end
  end
```

```
else
  do
    exec_RC = RC               /* Save exec return code            */
    say
    say "Error during 1st EXECIO ... DISKR, return code is " RC
    say
  end

  If exec_RC = 0 then          /* If no errors so far... continue  */
    do
    /***************************************************************/
    /* At this time, the stem variables newvar.0 through newvar.20 */
    /* will contain residual data from the previous EXECIO. We     */
    /* issue the "DROP newvar." instruction to clear these residual*/
    /* values from the stem.                                       */
    /***************************************************************/
    DROP newvar.               /* Set all stems variables to their */
                                  uninitialized state              */
    /***************************************************************/
    /* Read all records from 'sample2.data' and append them to the */
    /* end of 'all.sample.data'.                                   */
    /***************************************************************/
    "EXECIO * DISKR MYINDD2 (STEM NEWVAR. FINIS" /*Read all records*/
     if rc = 0 then             /* If read was successful          */
      do
      /*************************************************************/
      /* At this point, newvar.0 should be 10, indicating 10       */
      /* records have been read. Stem variables newvar.1, newvar.2,*/
      /* ... through newvar.10 will contain the 10 records. If we  */
      /* had not cleared the stem newvar. with the previous DROP   */
      /* instruction, variables newvar.11 through newvar.20 would  */
      /* still contain records 11 through 20 from the first data   */
      /* set. However, we would know that these values were not    */
      /* read by the last EXECIO DISKR since the current newvar.0  */
      /* variable indicates that only 10 records were read by      */
      /* that last EXECIO.                                         */
      /*************************************************************/
```

```
        say
        say
        say "-----------------------------------------------------"
        say newvar.0 "records have been read from 'sample2.data': "
        say
        do i = 1 to newvar.0   /* Loop through all records         */
          say newvar.i         /* Display the ith record           */
        end

        "EXECIO" newvar.0 "DISKW MYOUTDD (STEM NEWVAR." /* Write
                               exactly the number of records read  */
        if rc = 0 then         /* If write was successful          */
         do
           say
           say newvar.0 "records were written to 'all.sample.data'"
         end
        else
          do
            exec_RC = RC       /* Save exec return code            */
            say
            say "Error during 2nd EXECIO ...DISKW, return code is " RC
            say
          end
      end
    else
      do
        exec_RC = RC           /* Save exec return code            */
        say
        say "Error during 2nd EXECIO ... DISKR, return code is " RC
        say
      end
  end

"EXECIO 0 DISKW MYOUTDD (FINIS"   /* Close output file             */

"FREE FI(MYINDD1)"
"FREE FI(MYINDD2)"
```

```
"FREE FI(MYOUTDD)"
 exit 0
```

13. Example of reading records from an input RECFM=VS file and writing them to a new file having RECFM=VBS, then verifying that the copy is good.

    Note: Assuming input LRECL <=240.

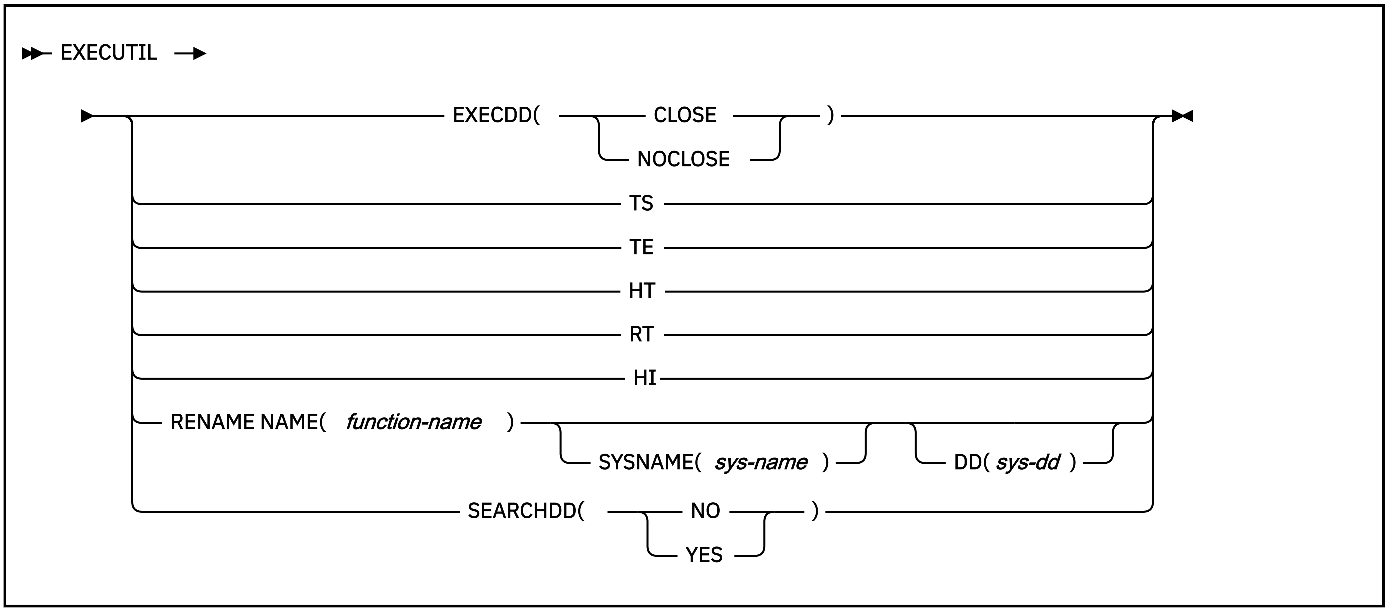
```
/*   REXX */
"ALLOC FI(INVS) DA('userid.test.vs') SHR REUSE"
ALLOCRC = RC
"ALLOC FI(OUTVBS) DA('userid.test.newvbs') SPACE(1) TRACKS " ,
   " LRECL(240) BLKSIZE(80) RECFM(V B S) DSORG(PS) NEW REUSE"
ALLOCRC = MAX(RC, ALLOCRC)

execio_rc = 0                         /* Initialize                    */
error = 0                             /* Initialize                    */
IF ALLOCRC = 0 THEN
  do
    /******************************************************************/
    /* When spanned records are read, each logical record is the      */
    /* collection of all spanned segments of that record on DASD.     */
    /******************************************************************/
    "execio * DISKR INVS (STEM inrec. FINIS"  /* Read all records     */
    if rc /= 0 then
      error = 1                       /* Read Error occurred          */
  end
ELSE
  do
    say 'File allocation error ...'
    error = 1                         /* Error occurred               */
  end
IF error = 0 then                     /* If no errors                 */
  DO
    "execio "inrec.0" DISKW OUTVBS (STEM inrec. FINIS"    /*   Write all
                                     records read to the new file     */

    if rc=0 then
      do
        say 'Output to new VBS file completed successfully'
        say 'Number of records copied ===> ' inrec.0
      end
    else
      do
        say 'Error writing to new VBS file '
        error = 1                      /* Error occurred               */
      end
 END
/**********************************************************************/
/* Confirm the new output VBS file matches the input.                 */
/**********************************************************************/
if error = 0 then                         /* No error so far          */
  do
    "execio * DISKR OUTVBS (STEM inrec2.  FINIS"  /* Read back
                                            the newly created file    */
    if rc = 0 & inrec.0 = inrec2.0 then   /* If read back OK          */
      do                                  /* Compare                  */
        match_cnt = 0
        do i=1 to inrec2.0
          if inrec2.i == inrec.i then
            match_cnt = match_cnt+1
        end
        if match_cnt = inrec2.0 then
          say 'New output VBS file matches input file - successful'
        else
          do
            say 'New output VBS does not match input - error.'
            say 'Num of records different ==> 'inrec2.0 -match_cnt
          end
        else
          say 'Unable to verify that new output VBS matches input'
      end
"FREE FI(INVS,OUTVBS)"
exit 0
```

14. Use EXECIO to read a member of a RECFM=U file, and change the first occurrence of the string 'TSO REXX' within each record to 'TSOEREXX' before rewriting the record. If a record is not changed, it need not be rewritten.

Note: If a record is not changed, it need not be rewritten.

```
/*  REXX */
/*--------------------------------------------------------------------*/
/*  Alloc my Load Lib data set having RECFM=U BLKSIZE=32000 LRECL=0   */
/*--------------------------------------------------------------------*/
/* Note that you cannot replace a PDS Load Library by a PDSE          */
/* Program Library in this example.  EXECIO can be used to read       */
/* (DISKR) a file allocated to a member in a PDSE Program Library     */
/* but it cannot update such a member, because EXECIO cannot          */
/* open such a file for update (DISKRU) nor for write (DISKW).        */
/*--------------------------------------------------------------------*/
"ALLOC FI(INOUTDD) DA('apar2.my.load(mymem)') SHR REUSE"

readcnt = 0                           /* Initialize rec read cntr     */
updtcnt = 0                           /* Initialize rec update cntr   */
error = 0                             /* Initialize flag              */
EOF = 0                               /* Initialize flag              */
do while (EoF=0 & error=0)            /* Loop while more records and
                                         no error
  "execio 1 DISKRU INOUTDD (STEM inrec."     /* Read a record         */
  if rc = 0 then                      /* If read ok                   */
    do                                /* Replace 1st occurrence of
                                      'TSO REXX' in record by 'TSOEREXX'
                                      and write it back               */
      readcnt = readcnt + 1         /* Records read                   */
      z = POS('TSO REXX', inrec.1,1)     /* Find target within rec    */
      if z /= 0 then                     /* If found, replace it      */
        do
          inrec.1 = SUBSTR(inrec.1,1,z-1)||'TSOEREXX'|| ,
                 SUBSTR(inrec.1,z+LENGTH('TSOEREXX'))     /*Replace it*/
          "execio 1 DISKW INOUTDD (STEM inrec." /* Rewrite the update
                                       made to the last record read   */
          if rc > 0 then              /* If error       */
            error=1                   /* Indicate error */
          else
            updtcnt = updtcnt + 1     /* Incr update count            */
        end
      else                            /* Else nothing changed, nothing
                                         to rewrite                   */
        NOP                           /* Continue with next record    */
    end
  else                                /* Else non-0 RC                */
    if rc=2 then                      /* if end-of-file               */
      EoF=1                           /* Indicate end-of-file         */
    else
      error=1                         /* Else read error              */
end                                   /* End do while                 */
"execio 0 DISKW INOUTDD (FINIS"       /* Close the file               */
if error = 1 then
  say '*** Error occurred while updating file '
else
  say updtcnt' of 'readcnt' records were changed'
"FREE FI(INOUTDD)"
exit 0
```

EXECUTIL



lets you change various characteristics that control how an exec processes in the TSO/E address space. You can use EXECUTIL:

- In a REXX exec that runs in a TSO/E address space
- From TSO/E READY mode
- From ISPF — the ISPF command line or the ISPF option that lets you enter a TSO/E command or CLIST
- In a CLIST. You can use EXECUTIL in a CLIST to affect exec processing. However, it has no effect on CLIST processing

You can also use EXECUTIL with the HI, HT, RT, TS, and TE operands from a program that is written in a high-level programming language by using the TSO service facility. From READY mode or ISPF, the HI, HT, and RT operands are not applicable because an exec is not currently running.

Use EXECUTIL to:

- Specify whether the system exec library (the default is SYSEXEC) is to be closed after the exec is located or is to remain open
- Start and end tracing of an exec
- Halt the interpretation of an exec
- Suppress and resume terminal output from an exec
- Change entries in a function package directory
- Specify whether the system exec library (the default is SYSEXEC) is to be searched in addition to SYSPROC.

Additional Considerations for Using EXECUTIL:

- All of the EXECUTIL operands are mutually exclusive, that is, you can only specify one of the operands on the command.
- The HI, HT, RT, TS, and TE operands on the EXECUTIL command are also, by themselves, *immediate commands*. Immediate commands are commands you can issue from the terminal if an exec is running in TSO/E and you press the attention interrupt key and enter attention mode. When you enter attention mode, you can enter an immediate command. Note that HE (Halt Execution) is an immediate command, but HE is not a valid operand on the EXECUTIL command.

  You can also use the TSO/E REXX commands TS (Trace Start) and TE (Trace End) in a REXX exec that runs in any address space (TSO/E and non-TSO/E). For information about the TS command, see "TS" on page 232. For information about the TE command, see "TE" on page 231.

- In general, EXECUTIL works on a language processor environment basis. That is, EXECUTIL affects only the current environment in which EXECUTIL is issued. For example, if you are in split screen in ISPF and issue EXECUTIL TS from the second ISPF screen to start tracing, only execs that are invoked from that ISPF screen are traced. If you invoke an exec from the first ISPF screen, the exec is not traced.

  Using the EXECDD and SEARCHDD operands may affect subsequent language processor environments that are created. For example, if you issue EXECUTIL SEARCHDD from TSO/E READY mode and then invoke ISPF, the new search order defined by EXECUTIL SEARCHDD may be in effect for the ISPF session also. This depends on whether your installation has provided its own parameters modules IRXTSPRM and IRXISPRM and the values specified in the load module.

EXECDD(CLOSE) or EXECDD(NOCLOSE)

Specifies whether the system exec library is to be closed after the system locates the exec but before the exec runs.

CLOSE causes the system exec library, whose default name is SYSEXEC, to be closed after the exec is located but before the exec runs. You can change this condition by issuing the EXECUTIL EXECDD(NOCLOSE) command.

NOCLOSE causes the system exec library to remain open. This is the default condition and can be changed by issuing the EXECUTIL EXECDD(CLOSE) command. The selected option remains in effect until it is changed by the appropriate EXECUTIL command, or until the current environment is terminated.

Note:

1. The EXECDD operand affects the ddname specified in the LOADDD field in the module name table. The default is SYSEXEC. "Module name table" on page 326 describes the table.
2. If you specify EXECDD(CLOSE), the exec library (DD specified in the LOADDD field) is closed immediately after an exec is loaded.
3. Specify EXECDD(CLOSE) or EXECDD(NOCLOSE) before running any execs out of the SYSEXEC file. If you attempt to use EXECDD(CLOSE) or EXECDD(NOCLOSE) after SYSEXEC has already been opened, you may not get the expected result because the SYSEXEC file must be closed at the same MVS task level at which it was opened.

Any libraries defined using the ALTLIB command are not affected by the EXECDD operand. SYSPROC is also not affected.

TS

Use TS (Trace Start) to start tracing execs. Tracing lets you interactively control the processing of an exec and debug problems. For more information about the interactive debug facility, see Chapter 11, "Debug aids," on page 233.

If you issue EXECUTIL TS from READY mode or ISPF, tracing is started for the next exec you invoke. Tracing is then in effect for that exec and any other execs it calls. Tracing ends:

- When the original exec completes
- If one of the invoked execs specifies EXECUTIL TE
- If one of the invoked execs calls a CLIST, which specifies EXECUTIL TE
- If you enter attention mode while an exec is running and issue the TE immediate command.

If you use EXECUTIL TS in an exec, tracing is started for all execs that are running. This includes the current exec that contains EXECUTIL TS, any execs it invokes, and any execs that were running when the current exec was invoked. Tracing remains active until all execs that are currently running complete or an exec or CLIST contains EXECUTIL TE.

For example, suppose exec A calls exec B, which then calls exec C. If exec B contains the EXECUTIL TS command, tracing is started for exec B and remains in effect for both exec C and exec A. Tracing ends when exec A completes. However, if one of the execs contains EXECUTIL TE, tracing ends for all of the execs.

If you use EXECUTIL TS in a CLIST, tracing is started for all execs that are running, that is, for any exec the CLIST invokes or execs that were running when the CLIST was invoked. Tracing ends when the CLIST and all execs that are currently running complete or if an exec or CLIST contains EXECUTIL TE. For example, suppose an exec calls a CLIST and the CLIST contains the EXECUTIL TS command. When control returns to the exec that invoked the CLIST, that exec is traced.

You can use EXECUTIL TS from a program by using the TSO service facility. For example, suppose an exec calls a program and the program encounters an error. The program can invoke EXECUTIL TS using the TSO service facility to start tracing all execs that are currently running.

You can also press the attention interrupt key, enter attention mode, and then enter TS to start tracing or TE to end tracing. You can also use the TS command (see "TS" on page 232) and TE command (see "TE" on page 231) in an exec.

TE

Use TE (Trace End) to end tracing execs. The TE operand is not really applicable in READY mode because an exec is not currently running. However, if you issued EXECUTIL TS to trace the next exec you invoke and then issued EXECUTIL TE, the next exec you invoke is not traced.

If you use EXECUTIL TE in an exec or CLIST, tracing is ended for all execs that are currently running. This includes execs that were running when the exec or CLIST was invoked and execs that the exec or CLIST calls. For example, suppose exec A calls CLIST B, which then calls exec C. If tracing was on and CLIST B contains EXECUTIL TE, tracing is ended and execs C and A are not traced.

You can use EXECUTIL TE from a program by using the TSO service facility. For example, suppose tracing has been started and an exec calls a program. The program can invoke EXECUTIL TE using the TSO service facility to end tracing of all execs that are currently running.

You can also press the attention interrupt key, enter attention mode, and then enter TE to end tracing. You can also use the TE immediate command in an exec (see "TE" on page 231).

HT

Use HT (Halt Typing) to suppress terminal output generated by an exec. The exec continues running. HT suppresses any output generated by REXX instructions or functions (for example, the SAY instruction) and REXX informational messages. REXX error messages are still displayed. Normal terminal output resumes when the exec completes. You can also use EXECUTIL RT to resume terminal output.

HT has no effect on CLISTs or commands. If an exec invokes a CLIST and the CLIST generates terminal output, the output is displayed. If an exec invokes a command, the command displays messages.

Use the HT operand in either an exec or CLIST. You can also use EXECUTIL HT from a program by using the TSO service facility. If the program invokes EXECUTIL HT, terminal output from all execs that are currently running is suppressed. EXECUTIL HT is not applicable from READY mode or ISPF because no execs are currently running.

If you use EXECUTIL HT in an exec, output is suppressed for all execs that are running. This includes the current exec that contains EXECUTIL HT, any execs the exec invokes, and any execs that were running when the current exec was invoked. Output is suppressed until all execs that are currently running complete or an exec or CLIST contains EXECUTIL RT.

If you use EXECUTIL HT in a CLIST, output is suppressed for all execs that are running, that is, for any exec the CLIST invokes or execs that were running when the CLIST was invoked. Terminal output resumes when the CLIST and all execs that are currently running complete or if an exec or CLIST contains EXECUTIL RT.

For example, suppose exec A calls CLIST B, which then calls exec C. If the CLIST contains EXECUTIL HT, output is suppressed for both exec A and exec C.

If you use EXECUTIL HT and want to display terminal output using the SAY instruction, you must use EXECUTIL RT before the SAY instruction to resume terminal output.

RT

Use RT (Resume Typing) to resume terminal output that was previously suppressed. Use the RT operand in either an exec or CLIST. You can also use EXECUTIL RT from a program by using the TSO service facility. If the program invokes EXECUTIL RT, terminal output from all execs that are currently running is resumed. EXECUTIL RT is not applicable from READY mode or ISPF because no execs are currently running.

If you use EXECUTIL RT in an exec or CLIST, typing is resumed for all execs that are running.

HI

Use HI (Halt Interpretation) to halt the interpretation of all execs that are currently running in the language processor environment. From either an exec or a CLIST, EXECUTIL HI halts the interpretation of all execs that are currently running. If an exec calls a CLIST and the CLIST contains EXECUTIL HI, the exec that invoked the CLIST stops processing.

EXECUTIL HI is not applicable from READY mode or ISPF because no execs are currently running.

You can use EXECUTIL HI from a program by using the TSO service facility. If the program invokes EXECUTIL HI, the interpretation of all execs that are currently running is halted.

If an exec enables the halt condition trap and the exec includes the EXECUTIL HI command, the interpretation of the current exec and all execs the current exec invokes is halted. However, any execs that were running when the current exec was invoked are not halted. These execs continue running. For example, suppose exec A calls exec B and exec B specifies EXECUTIL HI and also contains a SIGNAL ON HALT instruction (with a HALT: label). When EXECUTIL HI is processed, control is given to the HALT subroutine. When the subroutine completes, exec A continues processing at the statement that follows the call to exec B. For more information, see Chapter 7, “Conditions and condition traps,” on page 181.

RENAME

Use EXECUTIL RENAME to change entries in a function package directory. A function package directory contains information about the functions and subroutines that make up a function package. See “External functions and subroutines, and function packages” on page 263 for more information.

A function package directory contains the following fields for each function and subroutine:

- Func-name -- the name of the external function or subroutine that is used in an exec.
- Addr -- the address, in storage, of the entry point of the function or subroutine code.
- Sys-name -- the name of the entry point in a load module that corresponds to the code that is called for the function or subroutine.
- Sys-dd -- the name of the DD from which the function or subroutine code is loaded.

You can use EXECUTIL RENAME with the SYSNAME and DD operands to change an entry in a function package directory as follows:

- Use the SYSNAME operand to change the *sys-name* of the function or subroutine in the function package directory. When an exec invokes the function or subroutine, the routine with the new sys-name is invoked.
- Use EXECUTIL RENAME NAME(function-name) without the SYSNAME and DD operands to flag the directory entry as null. This causes the search for the function or subroutine to continue because a null entry is bypassed. The system will then search for a load module and/or an exec. See “Search order” on page 74 for the complete search order.

EXECUTIL RENAME clears the *addr* field in the function package directory to X'00'. When you change an entry, the name of the external function or subroutine is not changed, but the code that the function or subroutine invokes is replaced.

You can use EXECUTIL RENAME to change an entry so that different code is used.

NAME(function-name)

Specifies the name of the external function or subroutine that is used in an exec. This is also the name in the *func-name* field in the directory entry.

SYSNAME(sys-name)

Specifies the name of the entry point in a load module that corresponds to the package code that is called for the function or subroutine. If SYSNAME is omitted, the *sys-name* field in the package directory is set to blanks.

DD(sys-dd)

Specifies the name of the DD from which the package code is loaded. If DD is omitted, the *sys-dd* field in the package directory is set to blanks.

SEARCHDD(YES/NO)

Specifies whether the system exec library (the default is SYSEXEC) should be searched when execs are implicitly invoked. YES indicates that the system exec library (SYSEXEC) is searched, and if the exec is not found, SYSPROC is then searched. NO indicates that SYSPROC only is searched.

EXECUTIL SEARCHDD lets you dynamically change the search order. The new search order remains in effect until you issue EXECUTIL SEARCHDD again, the language processor environment terminates, or you use ALTLIB. Subsequently created environments inherit the same search order unless explicitly changed by the invoked parameters module.

ALTLIB affects how EXECUTIL operates to determine the search order. If you use the ALTLIB command to indicate that user-level, application-level, or system-level libraries are to be searched, ALTLIB operates on an application basis. For more information about the ALTLIB command, see *z/OS TSO/E Command Reference*.

EXECUTIL SEARCHDD generally affects the current language processor environment in which it is invoked. For example, if you are in split screen in ISPF and issue EXECUTIL SEARCHDD from the second ISPF screen to change the search order, the changed search order affects execs invoked from that ISPF screen. If you invoke an exec from the first ISPF screen, the changed search order is not in effect. However, if you issue EXECUTIL SEARCHDD from TSO/E READY mode, when you invoke ISPF, the new search order may also be in effect for ISPF. This depends on whether your installation has provided its own parameters modules IRXTSPRM and IRXISPRM and the values specified in the load module.

Return Codes: EXECUTIL returns the following return codes.

Return code	Meaning
0	Processing successful.
12	Processing unsuccessful. An error message has been issued.

Examples

1. Your installation uses both SYSEXEC and SYSPROC to store REXX execs and CLISTs. All of the execs you work with are stored in SYSEXEC and your CLISTs are stored in SYSPROC. Currently, your system searches SYSEXEC and SYSPROC and you do not use ALTLIB.

   You want to work with CLISTs only and do not need to search SYSEXEC. To change the search order and have the system search SYSPROC only, use the following command:

   ```
   EXECUTIL SEARCHDD(NO)
   ```

2. You are updating a REXX exec and including a new internal subroutine. You want to trace the subroutine to test for any problems. In your exec, include EXECUTIL TS at the beginning of your subroutine and EXECUTIL TE when the subroutine returns control to the main program. For example:

   ```
   /*  REXX program  */
   MAINRTN:
   ⋮
   CALL SUBRTN
   "EXECUTIL TE"
   ⋮
   EXIT
   /*  Subroutine follows  */
   SUBRTN:
   "EXECUTIL TS"
   ```

```
⋮
RETURN
```

3. You want to invoke an exec and trace it. The exec does not contain EXECUTIL TS or the TRACE instruction. Instead of editing the exec and including EXECUTIL TS or a TRACE instruction, you can enter the following from TSO/E READY mode:

   ```
   EXECUTIL TS
   ```

   When you invoke the exec, the exec is traced. When the exec completes processing, tracing is off.

4. Suppose an external function called PARTIAL is part of a function package. You have written your own function called PARTIAL or a new version of the external function PARTIAL and want to execute your new PARTIAL function instead of the one in the function package. Your new PARTIAL function may be an exec or may be stored in a load module. You must flag the entry for the PARTIAL function in the function package directory as null in order for the search to continue to execute your new PARTIAL function. To flag the PARTIAL entry in the function package directory as null, use the following command:

   ```
   EXECUTIL RENAME NAME(PARTIAL)
   ```

   When you execute the function PARTIAL, the null entry for PARTIAL in the function package directory is bypassed. The system will continue to search for a load module and/or exec that is called PARTIAL.

HE

```
▶▶— HE —▶◀
```

HE (Halt Execution) is an immediate command you can use to halt the execution of a REXX exec. The HE immediate command is available only if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter HE in response to the REXX attention prompting message, IRX0920I.

HE does not set the halt condition, which is set by the HI (Halt Interpretation) immediate command. If you need to halt the execution of an exec, it is suggested that you use the HI immediate command whenever possible. HE is useful if an exec is processing an external function or subroutine written in a programming language other than REXX and the function or subroutine goes into a loop.

For more information about how to use the HE immediate command, see Chapter 11, “Debug aids,” on page 233.

Example

You are running an exec in TSO/E. The exec invokes an external subroutine and the subroutine goes into a loop. To halt execution of the exec, press the attention interrupt key. The system issues the REXX attention prompting message that asks you to enter either a null line to continue or an immediate command. Enter HE to halt execution.

HI

```
▶▶— HI—▶◀
```

HI (Halt Interpretation) is an immediate command you can use to halt the interpretation of all currently executing execs. The HI immediate command is available only if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter HI in response to the REXX attention prompting message, IRX0920I.

After you enter HI, exec processing ends or control passes to a routine or label if the halt condition trap has been turned on in the exec. For example, if the exec contains a SIGNAL ON HALT instruction and exec processing is interrupted by HI, control passes to the HALT: label in the exec. See Chapter 7, “Conditions and condition traps,” on page 181 for information about the halt condition.

Example

You are running an exec in TSO/E that is in an infinite loop. To halt interpretation of the exec, press the attention interrupt key. The system issues the REXX attention prompting message that asks you to enter either a null line to continue or an immediate command. Enter HI to halt interpretation.

HT

```
►►— HT —►◄
```

HT (Halt Typing) is an immediate command you can use to suppress terminal output that an exec generates. The HT immediate command is available only if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter HT in response to the REXX attention prompting message, IRX0920I.

After you enter HT, the exec that is running continues processing, but the only output that is displayed at the terminal is output from TSO/E commands that the exec issues. All other output from the exec is suppressed.

Example

You are running an exec in TSO/E that calls an internal subroutine to display a line of output from a loop that repeats many times. Before the exec calls the subroutine, the exec displays a message that lets you press the attention interrupt key and then suppress the output by entering HT. When the loop is completed, the subroutine issues EXECUTIL RT to redisplay output.

```
/*  REXX program  */
⋮
SAY 'To suppress the output that will be displayed,'
SAY 'press the attention interrupt key and'
SAY 'enter HT.'
CALL printout
⋮
EXIT

printout:
DO i = 1 to 10000
⋮
SAY 'The outcome is' ....
END
"EXECUTIL RT"
RETURN
```

Immediate Commands

Immediate commands are commands you can use if you are running a REXX exec in TSO/E and you press the attention interrupt key to enter attention mode. When you enter attention mode, the system displays the REXX attention prompting message, IRX0920I. In response to the message, you can enter an immediate command. The immediate commands are:

- HE – Halt Execution

- HI – Halt Interpretation
- HT – Halt Typing
- RT – Resume Typing
- TE – Trace End
- TS – Trace Start

TE and TS are also TSO/E REXX commands you can use in a REXX exec that runs in any address space. That is, TE and TS are available from the TSO and MVS host command environments.

Except for HE, when you enter an immediate command from attention mode in TSO/E, the system processes the command as soon as control returns to the exec but before the next statement in the exec is interpreted. For the HE immediate command, the system processes the command before control returns to the exec.

For information about the syntax of each immediate command, see the description of the command in this chapter.

MAKEBUF

```
►►— MAKEBUF —►◄
```

Use the MAKEBUF command to create a new buffer on the data stack. The MAKEBUF command can be issued from REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

Initially, the data stack contains one buffer, which is known as buffer 0. You create additional buffers using the MAKEBUF command. MAKEBUF returns the number of the buffer it creates in the REXX special variable RC. For example, the first time an exec issues MAKEBUF, it creates the first buffer and returns a 1 in the special variable RC. The second time MAKEBUF is used, it creates another buffer and returns a 2 in the special variable RC.

To remove buffers from the data stack that were created with the MAKEBUF command, use the DROPBUF command (see "DROPBUF" on page 202).

After the MAKEBUF command executes, it sets the REXX special variable RC to the number of the buffer it created.

Return code	Meaning
1	A single additional buffer after the original buffer 0 now exists on the data stack.
2	A second additional buffer after the original buffer 0 now exists on the data stack.
3	A third additional buffer after the original buffer 0 now exists on the data stack.
n	An *n*th additional buffer after the original buffer 0 now exists on the data stack.

Example

An exec (execa) places two elements, elem1 and elem2, on the data stack. The exec calls a subroutine (sub3) that also places an element, elem3, on the data stack. The exec (execa) and the subroutine (sub3) each create a buffer on the data stack so they do not share their data stack information. Before the subroutine returns, it uses the DROPBUF command to remove the buffer it created.

```
/*  REXX program to ...  */
execa:
⋮
  "MAKEBUF"                 /* buffer created */
  SAY 'The number of buffers created is' RC   /* RC = 1 */
  PUSH elem1
  PUSH elem2
  CALL sub3
```

```
⋮
exit
sub3:
   "MAKEBUF"                /* second buffer created */
   PUSH elem3
⋮
   "DROPBUF"                /* second buffer created is deleted */
⋮
   RETURN
```

NEWSTACK

```
►►─ NEWSTACK ─►◄
```

creates a new data stack and basically hides or isolates the current data stack. Elements on the previous data stack cannot be accessed until a DELSTACK command is issued to delete the new data stack and any elements remaining in it.

The NEWSTACK command can be used in REXX execs that execute in both the TSO/E address space and non-TSO/E address spaces.

After an exec issues the NEWSTACK command, any element that is placed on the data stack with a PUSH or QUEUE instruction is placed on the new data stack. If an exec calls a routine (function or subroutine) after the NEWSTACK command is issued, that routine also uses the new data stack and cannot access elements on the previous data stack, unless it issues a DELSTACK command. If you issue a NEWSTACK command, you must issue a corresponding DELSTACK command to delete the data stack that NEWSTACK created.

When there are no more elements on the new data stack, PULL obtains information from the terminal (TSO/E address space) or the input stream (non-TSO/E address space), even though elements remain in the previous data stack (in non-TSO/E address spaces, the default input stream is SYSTSIN). To access elements on the previous data stack, issue a DELSTACK command. If a new data stack was not created, DELSTACK removes all elements from the original data stack.

For information about the PULL instruction, see “PULL” on page 62.

Multiple new data stacks can be created, but only elements on the most recently created data stack are accessible. To find out how many data stacks have been created, use the QSTACK command. To find the number of elements on the most recently created stack, use the QUEUED () built-in function.

If multiple language processor environments are chained together in a non-TSO/E address space and a new data stack is created with the NEWSTACK command, the new data stack is available only to execs that execute in the language processor environment in which the new data stack was created. The other environments in the chain cannot access the new data stack.

Examples

1. To protect elements placed on the data stack from a subroutine that might also use the data stack, you can use the NEWSTACK and DELSTACK commands as follows:

   ```
   PUSH element1
   PUSH element2
   ⋮
   "NEWSTACK"     /* data stack 2 created */
   CALL sub
   "DELSTACK"     /* data stack 2 deleted */
   ⋮
   PULL stackelem
   ⋮
   PULL stackelem
   EXIT
   ```

2. To put elements on the data stack and prevent the elements from being used as prompts for a TSO/E command, use the NEWSTACK command as follows:

```
"PROFILE PROMPT"
x = PROMPT("ON")
PUSH elem1
PUSH elem2
"NEWSTACK"     /* data stack 2 created */
"ALLOCATE"     /* prompts the user at the terminal for input. */
⋮
"DELSTACK"     /* data stack 2 deleted */
```

3. To use MVS batch to execute an exec named ABC, which is a member in USERID.MYREXX.EXEC, use program IRXJCL and include the exec name after the PARM parameter on the EXEC statement.

   ```
   //MVSBATCH EXEC  PGM=IRXJCL,
   //               PARM='ABC'
   //SYSTSPRT DD    DSN=USERID.IRXJCL.OUTPUT,DISP=OLD
   //SYSEXEC  DD    DSN=USERID.MYREXX.EXEC,DISP=SHR
   ```

   Exec ABC creates a new data stack and then put two elements on the new data stack for module MODULE3.

   ```
   "NEWSTACK"                /* data stack 2 created */
   PUSH elem1
   PUSH elem2
   ADDRESS LINK "module3"
   ⋮
   "DELSTACK"                /* data stack 2 deleted */
   ⋮
   ```

QBUF

```
►► QBUF ►◄
```

queries the number of buffers that were created on the data stack with the MAKEBUF command. The QBUF command returns the number of buffers in the REXX special variable RC. If you have not issued MAKEBUF to create any buffers on the data stack, QBUF sets the special variable RC to 0. In this case, 0 is the number of the buffer that is contained in every data stack.

You can use the QBUF command in REXX execs that run in both the TSO/E address space and non-TSO/E address spaces.

QBUF returns the current number of data stack buffers created by an exec and by other routines (functions and subroutines) the exec calls. You can issue QBUF from the calling exec or from a called routine. For example, if an exec issues two MAKEBUF commands and then calls a routine that issues another MAKEBUF command, QBUF returns 3 in the REXX special variable RC.

The following table shows how QBUF sets the REXX special variable RC.

Return code	Meaning
0	Only buffer 0 exists on the data stack
1	One additional buffer exists on the data stack
2	Two additional buffers exist on the data stack
n	*n* additional buffers exist on the data stack

Examples

1. If an exec creates two buffers on the data stack using the MAKEBUF command, deletes one buffer using the DROPBUF command, and then issues the QBUF command, RC is set to 1.

   ```
   "MAKEBUF"            /* buffer created */
   ⋮
   "MAKEBUF"            /* second buffer created */
   ```

```
⋮
"DROPBUF"            /* second buffer created is deleted */
"QBUF"
SAY 'The number of buffers created is' RC      /* RC = 1 */
```

2. Suppose an exec uses MAKEBUF to create a buffer and then calls a routine that also issues MAKEBUF. The called routine then calls another routine that issues two MAKEBUF commands to create two buffers. If either of the called routines or the original exec issues the QBUF command, QBUF sets the REXX special variable RC to 4.

```
"DROPBUF 0"     /* delete any buffers MAKEBUF created */
"MAKEBUF"                     /* create one buffer    */
SAY 'Buffers created = ' RC               /*  RC = 1  */
CALL sub1
"QBUF"
SAY 'Buffers created = ' RC               /*  RC = 4  */
EXIT

sub1:
"MAKEBUF"                     /* create second buffer */
SAY 'Buffers created = ' RC               /*  RC = 2  */
CALL sub2
"QBUF"
SAY 'Buffers created = ' RC               /*  RC = 4  */
RETURN

sub2:
"MAKEBUF"                     /* create third buffer  */
SAY 'Buffers created = ' RC               /*  RC = 3  */
⋮
"MAKEBUF"                     /* create fourth buffer */
SAY 'Buffers created = ' RC               /*  RC = 4  */
RETURN
```

QELEM

▶▶— QELEM —▶◀

queries the number of data stack elements that are in the most recently created data stack buffer (that is, in the buffer that was created by the MAKEBUF command). The number of elements is returned in the REXX special variable RC. When MAKEBUF has not been issued to create a buffer, QELEM returns the number 0 in the special variable RC, regardless of the number of elements on the data stack. Thus when QBUF returns 0, QELEM also returns 0.

The QELEM command can be issued from REXX execs that execute in both the TSO/E address space and in non-TSO/E address spaces.

QELEM only returns the number of elements in a buffer that was explicitly created using the MAKEBUF command. You can use QELEM to coordinate the use of MAKEBUF. Knowing how many elements are in a data stack buffer can also be useful before an exec issues the DROPBUF command, because DROPBUF removes the most recently created buffer and all elements in it.

The QELEM command returns the number of elements in the most recently created buffer. The QUEUED built-in function (see "QUEUED" on page 96) returns the total number of elements in the data stack, not including buffers.

After the QELEM command processes, the REXX special variable RC contains one of the following return codes:

Return code	Meaning
0	Either the MAKEBUF command has not been issued or the buffer that was most recently created by MAKEBUF contains no elements.
1	MAKEBUF has been issued and there is one element in the current buffer.

Return code	Meaning
2	MAKEBUF has been issued and there are two elements in the current buffer.
3	MAKEBUF has been issued and there are three elements in the current buffer.
n	MAKEBUF has been issued and there are *n* elements in the current buffer.

Examples

1. If an exec creates a buffer on the data stack with the MAKEBUF command and then puts three elements on the data stack, the QELEM command returns the number 3.

```
"MAKEBUF"          /* buffer created */
PUSH one
PUSH two
PUSH three
"QELEM"
SAY 'The number of elements in the buffer is' RC  /* RC = 3 */
```

2. Suppose an exec creates a buffer on the data stack, puts two elements on the data stack, creates another buffer, and then puts one element on the data stack. If the exec issues the QELEM command, QELEM returns the number 1. The QUEUED function, however, which returns the total number of elements on the data stack, returns the number 3.

```
"MAKEBUF"          /* buffer created */
QUEUE one
PUSH two
"MAKEBUF"          /* second buffer created */
PUSH one
"QELEM"
SAY 'The number of elements in the most recent buffer is' RC /* 1 */
SAY 'The total number of elements is' QUEUED()  /* returns 3 */
```

3. To check whether a data stack buffer contains elements before you remove the buffer, use the result from QELEM and QBUF in an IF...THEN...ELSE instruction.

```
"MAKEBUF"
PUSH a
"QELEM"
NUMELEM = RC       /* Assigns value of RC to variable NUMELEM */
"QBUF"
NUMBUF  = RC       /* Assigns value of RC to variable NUMBUF  */
IF (NUMELEM = 0) & (numbuf > 0) THEN
   "DROPBUF"
ELSE               /* Deletes most recently created buffer    */
   DO NUMELEM
      PULL elem
      SAY  elem
   END
```

QSTACK

```
►►─ QSTACK ─►◄
```

queries the number of data stacks in existence for an exec that is running. QSTACK returns the number of data stacks in the REXX special variable RC. The value QSTACK returns indicates the total number of data stacks, including the original data stack. If you have not issued a NEWSTACK command to create a new data stack, QSTACK returns 1 in the special variable RC for the original data stack.

You can use the QSTACK command in REXX execs that run in both the TSO/E address space and in non-TSO/E address spaces.

QSTACK returns the current number of data stacks created by an exec and by other routines (functions and subroutines) the exec calls. You can issue QSTACK from the calling exec or from a called routine.

For example, if an exec issues one NEWSTACK command and then calls a routine that issues another NEWSTACK command, and none of the new data stacks are deleted with the DELSTACK command, QSTACK returns 3 in the REXX special variable RC.

The following table shows how QSTACK sets the REXX special variable RC.

Return code	Meaning
0	No data stack exists. See "Data stack routine" on page 415.
1	Only the original data stack exists
2	One new data stack and the original data stack exist
3	Two new data stacks and the original data stack exist
n	*n - 1* new data stacks and the original data stack exist

Examples

1. Suppose an exec creates two new data stacks using the NEWSTACK command and then deletes one data stack using the DELSTACK command. If the exec issues the QSTACK command, QSTACK returns 2 in the REXX special variable RC.

```
"NEWSTACK"    /* data stack 2 created */
⋮
"NEWSTACK"    /* data stack 3 created */
⋮
"DELSTACK"    /* data stack 3 deleted */
"QSTACK"
SAY 'The number of data stacks is' RC   /* RC = 2 */
```

2. Suppose an exec creates one new data stack and then calls a routine that also creates a new data stack. The called routine then calls another routine that creates two new data stacks. When either of the called routines or the original exec issues the QSTACK command, QSTACK returns 5 in the REXX special variable RC. The data stack that is active is data stack 5.

```
"NEWSTACK"  /* data stack 2 created */
CALL sub1
"QSTACK"
SAY 'Data stacks =' RC  /* RC = 5 */
EXIT

sub1:
"NEWSTACK"  /* data stack 3 created */
CALL sub2
"QSTACK"
SAY 'Data stacks =' RC   /* RC = 5 */
RETURN

sub2:
"NEWSTACK"  /* data stack 4 created */
⋮
"NEWSTACK"  /* data stack 5 created */
"QSTACK"
SAY 'Data stacks =' RC  /* RC = 5 */
RETURN
```

RT

►► RT ►◄

RT (Resume Typing) is an immediate command you can use to resume terminal output that was previously suppressed. The RT immediate command is available only if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter RT in response to the REXX attention prompting message, IRX0920I. Terminal output that the exec generated after you issued the HT command and before you issued the RT command is lost.

Example

You are running an exec in TSO/E and have suppressed typing with the HT command. You now want terminal output from the exec to display at your terminal.

To resume typing, press the attention interrupt key. The system issues the REXX attention prompting message that asks you to enter either a null line to continue or an immediate command. Enter RT to resume typing.

SUBCOM

```
►► SUBCOM envname ►◄
```

queries the existence of a specified host command environment. SUBCOM searches the host command environment table for the named environment and sets the REXX special variable RC to 0 or 1. When RC contains 0, the environment exists. When RC contains 1, the environment does not exist.

You can use the SUBCOM command in REXX execs that run in any address space, TSO/E or non-TSO/E. That is, SUBCOM is available from the TSO and MVS host command environments.

Before an exec runs, a default host command environment is defined to process the commands that the exec issues. You can use the ADDRESS keyword instruction (see “ADDRESS” on page 41) to change the environment to another environment as long as the environment is defined in the host command environment table. Use the SUBCOM command to determine whether the environment is defined in the host command environment table for the current language processor environment. You can use the ADDRESS built-in function to determine the name of the environment to which host commands are currently being submitted (see “ADDRESS” on page 79).

Operand: The one operand for the SUBCOM command is:

envname
the name of the host command environment for which SUBCOM is to search.

When you invoke an exec from TSO/E, the following default host command environments are available:

- TSO (the default environment)
- CONSOLE
- CPICOMM
- LU62
- MVS
- LINK
- ATTACH
- LINKPGM
- ATTCHPGM
- LINKMVS
- ATTCHMVS

When you run an exec in a non-TSO/E address space, the following default host command environments are available:

- MVS (the default environment)
- CPICOMM
- LU62

- LINK
- ATTACH
- LINKPGM
- ATTCHPGM
- LINKMVS
- ATTCHMVS

When you invoke an exec from ISPF, the following default host command environments are available:

- TSO (the default environment)
- CONSOLE
- ISPEXEC
- ISREDIT
- CPICOMM
- LU62
- MVS
- LINK
- ATTACH
- LINKPGM
- ATTCHPGM
- LINKMVS
- ATTCHMVS

The SUBCOM command sets the REXX special variable RC to indicate the existence of the specified environment.

RC value	Description
0	The host command environment exists.
1	The host command environment does not exist.

Example

To check whether the ISPEXEC environment is available before using the ADDRESS instruction to change the environment, use the SUBCOM command as follows:

```
"SUBCOM ispexec"
IF RC = 0 THEN
  ADDRESS ispexec
ELSE NOP
```

TE

▶▶— TE —▶◀

TE (Trace End) is an immediate command you can use to end tracing REXX execs. The TE immediate command is available if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter TE in response to the REXX attention prompting message, IRX0920I. The exec continues processing, but tracing is off.

TE is also a TSO/E REXX command you can use in a REXX exec that runs in any address space. That is, TE is available from the TSO and MVS host command environments.

If you are running in interactive debug, you can also use TE without entering attention mode to end tracing.

Example

You have an exec that calls an internal subroutine. The subroutine is not processing correctly and you want to trace it. At the beginning of the subroutine, you can insert a TS command to start tracing. At the end of the subroutine, before the RETURN instruction, insert the TE command to end tracing before control returns to the main exec.

TS

```
►►─ TS ─►◄
```

TS (Trace Start) is an immediate command you can use to start tracing REXX execs. Tracing lets you control the execution of an exec and debug problems. The TS immediate command is available if an exec is running in TSO/E and you press the attention interrupt key to enter attention mode. You can enter TS in response to the REXX attention prompting message, IRX0920I. The exec continues processing and tracing is started.

TS is also a TSO/E REXX command you can use in a REXX exec that runs in any address space. That is, TS is available from the TSO and MVS host command environments.

In TSO/E foreground, trace output is written to the terminal. In TSO/E background, trace output is written to the output stream, SYSTSPRT. In non-TSO/E address spaces, trace output is written to the output stream as defined by the OUTDD field in the module name table (see page “Module name table” on page 326). The system default is SYSTSPRT.

To end tracing, you can use the TRACE OFF instruction or the TE immediate command. You can also use TE in the exec to stop tracing at a specific point. If you are running in interactive debug, you can use TE without entering attention mode to end tracing.

For more information about tracing, see the TRACE instruction on page “TRACE” on page 67 and Chapter 11, “Debug aids,” on page 233.

Example

You are running an exec in TSO/E and the exec is not processing correctly. To start tracing the exec, press the attention interrupt key. The system issues the REXX attention prompting message that asks you to enter either a null line to continue or an immediate command. Enter TS to start tracing.

Chapter 11. Debug aids

In addition to the TRACE instruction, (see "TRACE" on page 67) there are the following debug aids:

- The interactive debug facility
- The TSO/E REXX immediate commands:
  - HE — Halt Execution
  - HI — Halt Interpretation
  - TS — Trace Start
  - TE — Trace End

  You can use the immediate commands if a REXX exec is running in the TSO/E address space and you press the attention interrupt key. In attention mode, you can enter HE, HI, TS, or TE. You can also use the TS and TE immediate commands in a REXX exec that runs in any address space. That is, TS and TE are available from both ADDRESS MVS and ADDRESS TSO.
- The TSO/E REXX command EXECUTIL with the following operands:
  - HI — Halt Interpretation
  - TS — Trace Start
  - TE — Trace End

  You can use the EXECUTIL command in an exec that runs in the TSO/E address space. You can also use EXECUTIL from TSO/E READY mode and ISPF and in a TSO/E CLIST. You can use EXECUTIL with the HI, TS, or TE operands in a program written in a high-level programming language using the TSO service facility. See "EXECUTIL" on page 216 for more information.
- The trace and execution control routine IRXIC. You can invoke IRXIC from a REXX exec or any program that runs in any address space to use the following TSO/E REXX immediate commands:
  - HI — Halt Interpretation
  - TS — Trace Start
  - TE — Trace End
  - HT — Halt Typing
  - RT — Resume Typing

  See "Trace and execution control routine - IRXIC" on page 284 for more information.

Interactive debugging of programs

The debug facility permits interactively controlled execution of a REXX exec.

Changing the TRACE action to one with a prefix **?** (for example, TRACE ?A or the TRACE built-in function) turns on interactive debug and indicates to the user that interactive debug is active. You can interactively debug REXX execs in the TSO/E address space from your terminal session.

Further TRACE instructions in the exec are ignored, and the language processor pauses after nearly all instructions that are traced at the terminal (see the following for exceptions). When the language processor pauses, three debug actions are available:

1. **Entering a null line** (with no characters, including no blanks) makes the language processor continue execution until the next pause for debug input. Repeatedly entering a null line, therefore, steps from pause point to pause point. For TRACE ?A, for example, this is equivalent to single-stepping through the exec.

2. **Entering an equal sign (=)**, with no blanks, makes the language processor re-execute the clause last traced. For example: if an IF clause is about to take the wrong branch, you can change the value of the variable(s) on which it depends, and then re-execute it.

   After the clause has been re-executed, the language processor pauses again.

3. **Anything else entered** is treated as a **line** of one or more clauses, and processed immediately (that is, as though DO; line ; END; had been inserted in the exec). The same rules apply as in the INTERPRET instruction (for example, DO-END constructs must be complete). If an instruction has a syntax error in it, a standard message is displayed and you are prompted for input again. Similarly, all the other SIGNAL conditions are disabled while the string is processed to prevent unintentional transfer of control.

   During execution of the string, no tracing takes place, except that nonzero return codes from host commands are displayed. Host commands are always executed (that is, they are not affected by the prefix **!** on TRACE instructions), but the variable RC is not set.

   After the string has been processed, the language processor pauses again for further debug input, unless a TRACE instruction was entered. In this latter case, the language processor immediately alters the tracing action (if necessary) and then continues executing until the next pause point (if any). Therefore, to alter the tracing action (from All to Results, for example) and then re-execute the instruction, you must use the built-in function TRACE (see “TRACE” on page 102). For example, CALL TRACE I changes the trace action to "I" and allows re-execution of the statement after which the pause was made. Interactive debug is turned off, when it is in effect, if a TRACE instruction uses a prefix, or at any time, when a TRACE O or TRACE with no options is entered.

   You can use the numeric form of the TRACE instruction to allow sections of the exec to be executed without pause for debug input. TRACE n (that is, positive result) allows execution to continue, skipping the next n pauses (when interactive debug is or becomes active). TRACE -n (that is, negative result) allows execution to continue without pause and with tracing inhibited for n clauses that would otherwise be traced.

The trace action selected by a TRACE instruction is saved and restored across subroutine calls. This means that if you are stepping through an exec (for example, after using TRACE ?R to trace Results) and then enter a subroutine in which you have no interest, you can enter TRACE O to turn tracing off. No further instructions in the subroutine are traced, but on return to the caller, tracing is restored.

Similarly, if you are interested only in a subroutine, you can put a TRACE ?R instruction at its start. Having traced the routine, the original status of tracing is restored and, therefore, (if tracing was off on entry to the subroutine) tracing (and interactive debug) is turned off until the next entry to the subroutine.

You can switch tracing on (without modifying an exec) using the command EXECUTIL TS. You can also switch tracing on or off asynchronously, (that is, while an exec is running) using the TS and TE immediate commands. See “Interrupting exec processing” on page 235 for the description of these facilities.

Because you can execute any instructions in interactive debug, you have considerable control over execution.

Some examples:

```
Say expr     /* displays the result of evaluating the         */
             /* expression.                                    */

name=expr    /* alters the value of a variable.               */

Trace O      /* (or Trace with no options) turns off          */
             /* interactive debug and all tracing.             */

Trace ?A     /* turns off interactive debug but continues     */
             /* tracing all clauses.                           */

Trace L      /* makes the language processor pause at labels  */
             /* only.  This is similar to the traditional     */
             /* "breakpoint" function, except that you        */
             /* do not have to know the exact name and        */
             /* spelling of the labels in the exec.           */

exit         /* terminates execution of the exec.             */
```

```
Do i=1 to 10; say stem.i; end /* displays ten elements of the */
                              /* array stem.                  */
```

Exceptions: Some clauses cannot safely be re-executed, and therefore, the language processor does not pause after them, even if they are traced. These are:

- Any repetitive DO clause, on the second or subsequent time around the loop
- All END clauses (not a useful place to pause in any case)
- All THEN, ELSE, OTHERWISE, or null clauses
- All RETURN and EXIT clauses
- All SIGNAL and CALL clauses (the language processor pauses after the target label has been traced)
- Any clause that raises a condition that CALL ON or SIGNAL ON traps (the pause takes place after the target label for the CALL or SIGNAL has been traced)
- Any clause that causes a syntax error (These can be trapped by SIGNAL ON SYNTAX, but cannot be re-executed.)

Interrupting execution and controlling tracing

The following topics describe how you can interrupt the processing of a REXX exec and how you can start and stop tracing an exec.

Interrupting exec processing

You can interrupt the language processor during processing in several ways:

- In the TSO/E address space, you can use the HI (Halt Interpretation) immediate command or the EXECUTIL HI command to halt the interpretation of execs. HI and EXECUTIL HI cause the interpretation of all REXX execs that are currently running to be halted, as though a halt condition had been raised. This is especially useful when an exec gets into a loop and you want to end processing.

  If an exec is running, you can press the attention interrupt key and enter attention mode. In attention mode, you can enter HI to halt the interpretation of the exec.

  You can use EXECUTIL with the HI operand in a REXX exec. You can also use EXECUTIL HI in a TSO/E CLIST or in a program that is written in a high-level programming language using the TSO service facility.

  When an HI interrupt halts the interpretation of an exec, the data stack is cleared. You can trap an HI interrupt by enabling the halt condition using either the CALL ON or SIGNAL ON instruction (see Chapter 7, “Conditions and condition traps,” on page 181).
- In any address space (TSO/E and non-TSO/E), you can call the trace and execution control routine, IRXIC, to invoke the HI immediate command and halt the interpretation of all REXX execs that are currently running. You can invoke IRXIC from an exec or other program in any address spaces.
- In the TSO/E address space, you can use the HE (Halt Execution) immediate command to halt the execution of an exec. If an exec is running, you can press the attention interrupt key and enter attention mode. In attention mode, you can enter HE to halt the exec.

From attention mode, the HI immediate command is processed as soon as control returns to the exec, but before the next statement in the exec is interpreted. For the HE immediate command, the system processes the command before control returns to the exec.

If the exec is processing an external function or subroutine written in a programming language other than REXX or the exec is processing a host command, when you halt exec interpretation using HI, the halt is not processed until the function, subroutine, or command returns to the calling exec. That is, the function, subroutine, or command completes processing before exec processing is interrupted.

The HE immediate command is useful if an exec invokes an external function or subroutine that is written in a programming language other than REXX and the function or subroutine cannot return to the invoking exec (for example, because it goes into a loop). HE is also useful for certain host commands that may hang and cannot return to the exec, for example, the commands available under ADDRESS MVS. In these cases, the HI immediate command cannot halt the exec because HI is not processed until the function, subroutine, or command returns to the exec. However, the HE immediate command is processed immediately and halts the exec.

For more information, see "Using the HE immediate command to halt an exec" on page 236.

Considerations for interrupting exec processing

If you are running a REXX exec in TSO/E and press the attention interrupt key to interrupt exec processing, there are several considerations of which you should be aware.

- Considerations for interrupting a host command that is running in a REXX exec.

  Unless a command provides its own attention processing, if a host command is processing and you press the attention interrupt key, the language processor terminates the command and returns a value of -1 in the REXX special variable RC. In this case, the language processor does not display a message that lets you enter an immediate command, such as TS (Trace Start) or HI (Halt Interpretation).

- Considerations for interrupting a REXX exec that is running under ISPF.

  When the language processor gives control to an ISPF or ISPF/PDF service (for example, the SELECT service) and you press the attention interrupt key, attention processing is under the control of ISPF. For example, if ISPF is processing a command using the SELECT service and you press the attention interrupt key, ISPF displays a message that the command was terminated and then terminates the screen. In this case, the language processor does not display a message that lets you enter an immediate command, such as TS (Trace Start) or HI (Halt Interpretation) and ISPF sets the REXX special variable RC.

  Note that when ISPF is active and the language processor is in control, whether or not the language processor displays the message that allows you to enter an immediate command depends on how ISPF was started. For example, if ISPF is started using the ISPSTART command with the TEST operand, ISPF attention processing is disabled and, therefore, the language processor environment's attention processing is also disabled.

Using the HE immediate command to halt an exec

In the TSO/E address space, you can use the HE (Halt Execution) immediate command to halt the execution of a REXX exec. You can use the HE immediate command only if you are running an exec in TSO/E and you press the attention interrupt key and enter attention mode. When you enter attention mode, the system displays the REXX attention prompting message, IRX0920I. You can enter HE in response to the message.

If you need to stop the processing of a REXX exec, it is suggested that you use the HI immediate command instead of HE whenever possible.

Note that unlike the other immediate commands, HE is not a valid operand on the EXECUTIL command, nor does the trace and execution control routine, IRXIC, support the HE command.

If you have nested execs and use the HE immediate command, HE works differently for execs you invoke from TSO/E READY mode compared to execs you invoke from ISPF. As an example, suppose you have an exec (EXECA) that calls another exec (EXECB). While the EXECB exec is running, you enter attention mode and enter the HE immediate command to halt execution. The HE immediate command works as follows:

- If you invoked the EXECA exec from ISPF, the HE immediate command halts the execution of both the EXECB exec and the EXECA exec.
- If you invoked the EXECA exec from TSO/E READY, the HE immediate command halts the execution of the currently running exec, which is EXECB. The top-level exec (EXECA) may or may not be halted depending on how the EXECA exec invoked EXECB.
  - If EXECA invoked EXECB using the TSO/E EXEC command, the HE immediate command does not halt the execution of EXECA. For example, suppose EXECA used the following command to invoke EXECB:

    ```
    ADDRESS TSO "EXEC 'winston.workds.rexx(execb)' exec"
    ```

  When you enter HE while the EXECB exec is running, the EXECB exec is halted and control returns to EXECA. In this case, the TSO/E EXEC command terminates and the REXX special variable RC is set to 12. The EXECA exec continues processing at the clause following the TSO/E EXEC command.

  - If EXECA invoked EXECB using either a subroutine call (CALL EXECB) or a function call (X = EXECB(arg)), the following occurs. The EXECB exec is halted and control returns to the calling exec, EXECA. In this case, EXECB is prematurely halted and the calling exec (EXECA) raises the SYNTAX condition because the function or subroutine failed.

If you use the HE immediate command and you halt the execution of an external function, external subroutine, or a host command, note the following. The function, subroutine, or command does not regain control to perform its normal cleanup processing. Therefore, its resources could be left in an inconsistent state. If the function, subroutine, or command requires cleanup processing, it should be covered by its own recovery ESTAE, which performs any required cleanup and then percolates.

Starting and stopping tracing

The following describes how to start and stop tracing an exec.

You can start tracing REXX execs in several ways:

- You can use the TRACE instruction to start tracing. For more information, see "TRACE" on page 67.
- In the TSO/E address space, you can use the TS (Trace Start) immediate command or the EXECUTIL TS command to start tracing. If an exec is running and you press the attention interrupt key, after you enter attention mode, you can enter TS to start tracing.

  You can use EXECUTIL with the TS operand in a REXX exec. You can also use EXECUTIL TS in a TSO/E CLIST or in a program that is written in a high-level programming language by using the TSO service facility.

  TS or EXECUTIL TS puts the REXX exec into normal interactive debug. You can then execute REXX instructions; for example, to display variables or EXIT. Interactive debug is helpful if an exec is looping. You can inspect the exec and step through the execution before deciding whether or not to continue execution.
- In any address space (TSO/E and non-TSO/E), you can use the TS (Trace Start) immediate command in a REXX exec to start tracing. The trace output is written to the:
  - Terminal (TSO/E foreground)
  - Output stream SYSTSPRT (TSO/E background)
  - Output stream, which is usually SYSTSPRT (non-TSO/E address space)

  In any address space, you can call the trace and execution control routine IRXIC to invoke the TS immediate command. You can invoke IRXIC from an exec or other program in any address space.

You can end tracing in several ways:

- You can use the TRACE OFF instruction to end tracing. For more information, see "TRACE" on page 67.
- In the TSO/E address space, you can use the TE (Trace End) immediate command or the EXECUTIL TE command to end tracing. If an exec is running and you press the attention interrupt key, after you enter attention mode, you can enter TE to end tracing.

  You can use EXECUTIL with the TE operand in a REXX exec. You can also use EXECUTIL TE in a TSO/E CLIST or in a program that is written in a high-level programming language by using the TSO service facility.

  TE or EXECUTIL TE has the effect of executing a TRACE O instruction. The commands are useful if you want to end tracing when you are not in interactive debug.
- In any address space (TSO/E and non-TSO/E), you can use the TE (Trace End) immediate command in a REXX exec to end tracing.

  In any address space, you can call the trace and execution control routine IRXIC to invoke the TE immediate command. You can invoke IRXIC from an exec or other program in any address spaces.

For more information about the HI, TS, and TE immediate commands and the EXECUTIL command, see Chapter 10, “TSO/E REXX commands,” on page 201.

For more information about the trace and execution control routine IRXIC, see “Trace and execution control routine - IRXIC” on page 284.

Chapter 12. TSO/E REXX programming services

In addition to the REXX language instructions and built-in functions, and the TSO/E external functions and REXX commands that are provided for writing REXX execs, TSO/E provides programming services for REXX processing. Some programming services are routines that let you interface with REXX and the language processor.

In addition to the TSO/E REXX programming services that are described in this chapter, TSO/E also provides various routines that let you customize REXX processing. These are described beginning in Chapter 13, "TSO/E REXX customizing services," on page 305. TSO/E also provides *replaceable routines* that handle system services. The routines are described in Chapter 16, "Replaceable routines and exits," on page 389. Whenever you invoke a TSO/E REXX routine, there are general conventions relating to registers that are passed on the call, parameter lists, and return codes the routines return. "General considerations for calling TSO/E REXX routines" on page 240 highlights several major considerations about calling REXX routines.

The REXX programming services TSO/E provides are summarized below and are described in detail in the individual topics in this chapter.

IRXJCL and IRXEXEC Routines: IRXJCL and IRXEXEC are two routines that you can use to run a REXX exec in any MVS address space. Both IRXEXEC and IRXJCL are programming interfaces to the language processor.

You can use IRXJCL to run a REXX exec in MVS batch by specifying IRXJCL as the program name (PGM=) on the JCL EXEC statement. You can also invoke IRXJCL from a REXX exec or a program in any address space to run a REXX exec.

You can invoke IRXEXEC from a REXX exec or a program in any address space to run a REXX exec. Using IRXEXEC instead of the IRXJCL routine or, in TSO/E, the EXEC command processor to invoke an exec provides more flexibility. For example, you can preload the exec in storage and then use IRXEXEC to run the exec. "Exec processing routines - IRXJCL and IRXEXEC" on page 245 describes the IRXJCL and IRXEXEC programming interfaces in more detail.

External Functions and Subroutines, and Function Packages: You can extend the capabilities of the REXX programming language by writing your own external functions and subroutines that you can then use in REXX execs. You can write an external function or subroutine in REXX. For performance reasons, you can write external functions and subroutines in either assembler or a high-level programming language and store them in a load library. You can also group frequently used external functions and subroutines into a *function package*, which provides quick access to the packaged functions and subroutines. When a REXX exec calls an external function or subroutine, the function packages are searched before load libraries or exec data sets, such as SYSEXEC and SYSPROC. See "Search order" on page 74 for a description of the complete search order.

If you write external functions and subroutines in any programming language other than REXX, the language must support the system-dependent interfaces that the language processor uses to invoke the function or subroutine. If you want to include an external function or subroutine in a function package, the function or subroutine must be link-edited into a load module. "External functions and subroutines, and function packages" on page 263 describes the system-dependent interfaces for writing external functions and subroutines and how to create function packages.

Variable Access: TSO/E provides the IRXEXCOM variable access routine that lets unauthorized commands and programs access and manipulate REXX variables. Using IRXEXCOM, you can inspect, set, or drop variables. IRXEXCOM can be called in both the TSO/E and non-TSO/E address spaces. "Variable access routine - IRXEXCOM" on page 274 describes IRXEXCOM in detail.

Note: TSO/E also provides the IKJCT441 routine that lets authorized and unauthorized commands and programs access REXX variables. IKJCT441 can be used only in the TSO/E address space and is described in *z/OS TSO/E Programming Services*.

Maintain Host Command Environments: When a REXX exec runs, there is at least one *host command environment* available for processing host commands. When an exec begins running, an initial environment is defined. You can change the host command environment using the ADDRESS instruction (see "ADDRESS" on page 41).

When the language processor processes an instruction that is a host command, it first evaluates the expression and then passes the command to the active host command environment for processing. A specific routine defined for the host command environment handles the command processing. TSO/E provides several host command environments for execs that run in non-TSO/E address spaces and in the TSO/E address space (for TSO/E and ISPF). "Commands to external environments" on page 22 describes how you issue commands to the host and the different environments TSO/E provides for MVS (non-TSO/E), TSO/E, and ISPF.

The valid host command environments, the routines that are invoked to handle command processing within each environment, and the initial environment that is available to a REXX exec when the exec begins running are defined in a *host command environment table*. You can customize REXX processing to define your own host command environment and provide a routine that handles command processing for that environment. Chapter 13, "TSO/E REXX customizing services," on page 305 describes how to customize REXX processing in more detail.

TSO/E also provide the IRXSUBCM routine that lets you access the entries in the host command environment table. Using IRXSUBCM, you can add, change, and delete entries in the table and also query the values for a particular host command environment entry. "Maintain entries in the host command environment table - IRXSUBCM" on page 280 describes the IRXSUBCM routine in detail.

Trace and Execution Control: TSO/E provides the trace and execution control routine, IRXIC, that lets you use the HI, HT, RT, TS, and TE commands to control the processing of REXX execs. For example, you can invoke IRXIC from a program written in assembler or a high-level language to control the tracing and execution of execs. "Trace and execution control routine - IRXIC" on page 284 describes the IRXIC routine in detail.

Get Result Routine: TSO/E provides the *get result* routine, IRXRLT, that lets you obtain the result from a REXX exec that was invoked using the IRXEXEC routine. You can also use IRXRLT if you write external functions and subroutines in a programming language other than REXX. IRXRLT lets your function or subroutine code obtain a large enough area of storage to return the result to the calling exec. The IRXRLT routine also lets a compiler runtime processor obtain an evaluation block to handle the result from a compiled REXX exec. "Get result routine - IRXRLT" on page 286 describes the IRXRLT routine in detail.

SAY Instruction Routine: The SAY instruction routine, IRXSAY, lets you write a character string to the same output stream as the REXX SAY keyword instruction. "SAY instruction routine - IRXSAY" on page 292 describes the IRXSAY routine in detail.

Halt Condition Routine: The halt condition routine, IRXHLT, lets you query or reset the halt condition. "Halt condition routine - IRXHLT" on page 295 describes the IRXHLT routine in detail.

Text Retrieval Routine: The text retrieval routine, IRXTXT, lets you retrieve the same text that the TSO/E REXX interpreter uses for the ERRORTEXT built-in function and for certain options of the DATE built-in function. For example, using IRXTXT, a program can retrieve the name of a month or the text of a syntax error message. "Text retrieval routine - IRXTXT" on page 297 describes the IRXTXT routine in detail.

LINESIZE Function Routine: The LINESIZE function routine, IRXLIN, lets you retrieve the same value that the LINESIZE built-in function returns. "LINESIZE function routine - IRXLIN" on page 302 describes the IRXLIN routine in detail.

General considerations for calling TSO/E REXX routines

Each topic in this book that describes the different TSO/E REXX routines describes how to use the routine, including entry and return specifications and parameter lists. The following topics provide general information about calling TSO/E REXX routines.

All TSO/E REXX routines, except for the initialization routine, IRXINIT, cannot run without a language processor environment being available. A language processor environment is the environment in which

REXX operates, that is, in which the language processor processes a REXX exec. REXX execs and TSO/E REXX routines run in a language processor environment.

The system automatically initializes a language processor environment in the TSO/E and non-TSO/E address spaces by calling the initialization routine, IRXINIT. In TSO/E, an environment is initialized during logon processing for TSO/E READY mode. During your TSO/E session, you can invoke an exec or use a TSO/E REXX routine. The exec or routine runs in the environment that was created during logon processing.

If you invoke ISPF, the system initializes another language processor environment for the ISPF screen. If you split the ISPF screen, a third environment is initialized for that screen. In ISPF, when you invoke an exec or TSO/E REXX routine, the exec or routine runs in the language processor environment from which it was invoked.

The system automatically terminates the three language processor environments it initializes as follows:

- When you return to one screen in ISPF, the environment for the second screen is terminated
- When you end ISPF and return to TSO/E READY mode, the environment for the first ISPF screen is terminated
- When you log off TSO/E, the environment for TSO/E READY mode is terminated.

In non-TSO/E address spaces, the system does not automatically initialize a language processor environment at a specific point, such as when the address space is activated. When you invoke either the IRXJCL or IRXEXEC routine to run an exec, the system automatically initializes an environment if an environment does not already exist. The exec then runs in that environment. The exec can then invoke a TSO/E REXX routine, such as IRXIC, and the routine runs in the same environment in which the exec is running. Chapter 14, “Language processor environments,” on page 311 describes environments in more detail, when they are initialized, and the different characteristics that make up an environment.

You can explicitly call the initialization routine, IRXINIT, to initialize language processor environments. Calling IRXINIT lets you *customize* the environment and how execs and services are processed and used. Using IRXINIT, you can create several different environments in an address space. IRXINIT is primarily intended for use in non-TSO/E address spaces, but you can also use it in TSO/E. Customization information is described in more detail in Chapter 13, “TSO/E REXX customizing services,” on page 305.

If you explicitly call IRXINIT to initialize environments, whenever you call a TSO/E REXX routine, you can specify in which language processor environment you want the routine to run. During initialization, IRXINIT creates several control blocks that contain information about the environment. The main control block is the environment block, which represents the language processor environment. If you use IRXINIT and initialize several environments and then want to call a TSO/E REXX routine to run in a specific environment, you can pass the address of the environment block for the environment on the call. When you call the TSO/E REXX routine, you can pass the address of the environment block either in register 0 or in the environment block address parameter in the parameter list if the routine supports the parameter. By using the TSO/E REXX customizing services and the environment block, you can customize REXX processing and also control in which environment you want TSO/E REXX routines to run. For more information, see “Specifying the address of the environment block” on page 243.

The following information describes some general conventions about calling TSO/E REXX routines:

- The REXX vector of external entry points is a control block that contains the addresses of the TSO/E REXX routines and the system-supplied and user-supplied replaceable routines. The vector lets you easily access the address of a specific routine to invoke the routine. See “Control blocks created for a Language Processor Environment” on page 357 for more information about the vector.
- All calls are in 31 bit addressing mode.
- All data areas may be above 16 MB in virtual storage.
- On entry to an external function or subroutine, register 0 contains the address of the environment block. This address should be passed to any TSO/E REXX programming service invoked from the external function or subroutine. Passing the address of the environment block is particularly important if the environment is reentrant because TSO/E REXX programming services cannot automatically

locate a reentrant environment. For more information about reentrant environments, see “Using the environment block for reentrant environments” on page 244.

- For most of the TSO/E REXX routines, you pass a parameter list on the call. Register 1 contains the address of the parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last parameter address must be a binary 1. If you do not use a parameter, you must pass either binary zeros (for numeric data or addresses) or blanks (for character data). For more information, see “Parameter lists for TSO/E REXX routines” on page 242.
- On calls to the TSO/E REXX routines, you can pass the address of an environment block to specify in which particular language processor environment you want the routine to run. For more information, see “Specifying the address of the environment block” on page 243.
- Specific return codes are defined for each TSO/E REXX routine. Some common return codes include 0, 20, 28, and 32. For more information, see “Return codes for TSO/E REXX routines” on page 244.

Parameter lists for TSO/E REXX routines

Most of the TSO/E REXX routines have parameter lists. The parameters provide information to the routine about what type of processing you want to perform and also provide a way for the routine to return information to the program that called it. All the parameter lists are passed to the routines in the same manner. Figure 14 on page 242 shows the format of the parameter lists for the TSO/E REXX routines. A description of the parameter list follows the figure.

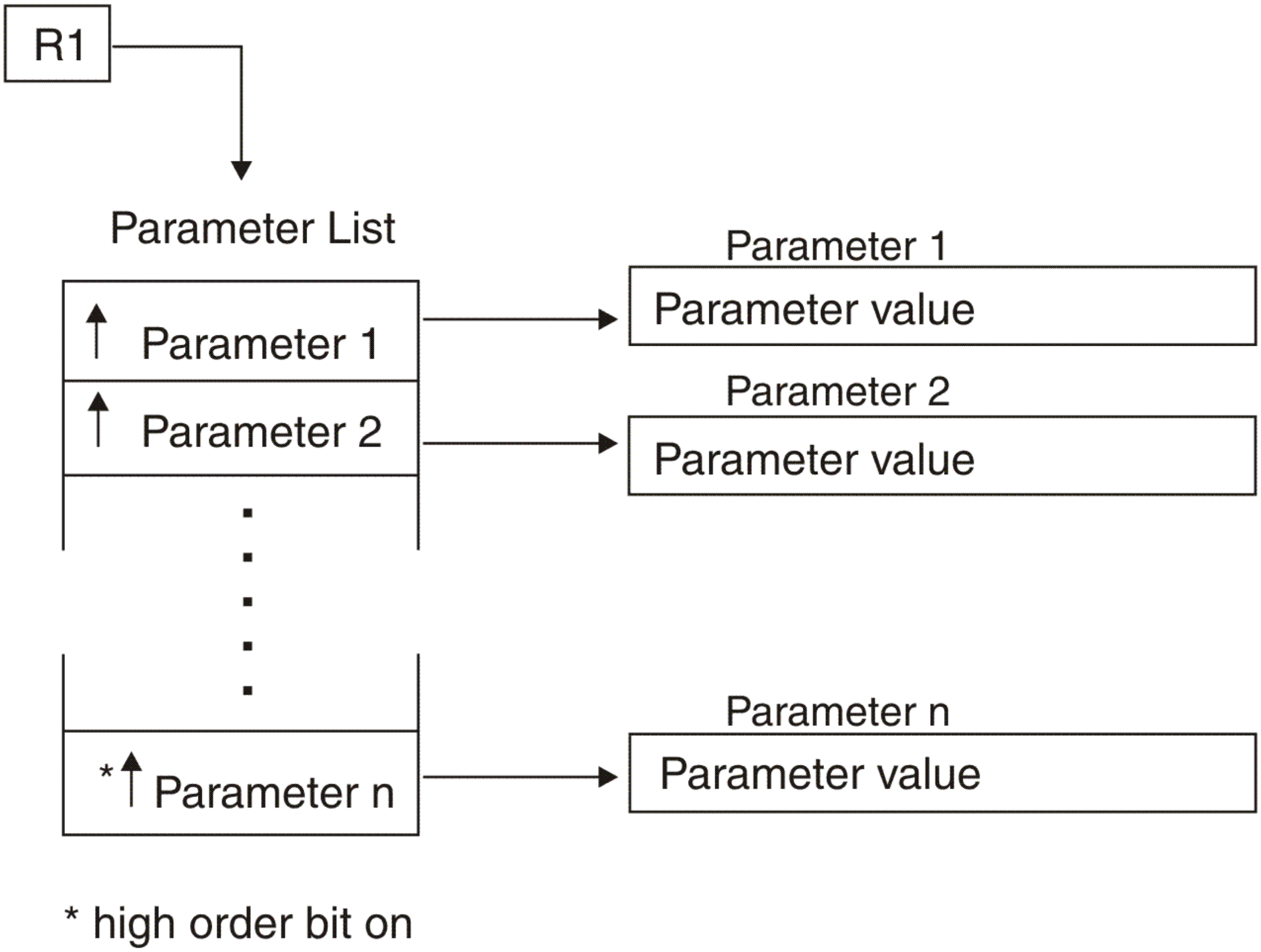


Figure 14. Overview of parameter lists for TSO/E REXX routines

Register 1 contains an address that points to a parameter list. The parameter list consists of a list of addresses. Each address in the parameter list points to a parameter. This is illustrated on the left side of the diagram in Figure 14 on page 242. The end of the parameter list (the list of addresses) is indicated by the high-order bit of the last address being set to a binary 1.

The parameters themselves are shown on the right side of the diagram in Figure 14 on page 242. The parameter value can be the data itself or it can be an address that points to the data.

All of the parameters for a specific routine might not be required. That is, some parameters may be optional. Because of this, the parameter lists are of *variable length* and the end of the parameter list must be indicated by the high-order bit being set on in the last address.

If there is an optional parameter you do not want to use and there are parameters after it you want to use, you can specify the address of the optional parameter in the parameter list, but set the optional parameter itself to either binary zeros (for numeric data or addresses) or to blanks (for character data). Otherwise, you can simply end the parameter list at the parameter before the optional parameter by setting the high-order bit on in the preceding parameter's address.

For example, suppose a routine has seven parameters and parameters 6 and 7 are optional. You do not want to use parameter 6, but you want to use parameter 7. In the parameter list, specify the address of parameter 6 and set the high-order bit on in the address of parameter 7. For parameter 6 itself, specify 0 or blanks, depending on whether the data is numeric or character data.

As another example, suppose the routine has seven parameters, parameters 6 and 7 are optional, and you do not want to use the optional parameters (parameters 6 and 7). You can end the parameter list at parameter 5 by setting the high-order bit of the address for parameter 5 on.

The individual descriptions of each routine in this book describe the parameters, the values you can specify for each parameter, and whether a parameter is optional.

Specifying the address of the environment block

You can explicitly call the initialization routine, IRXINIT, to initialize a language processor environment in an address space. If you explicitly call IRXINIT to initialize an environment, you can optionally specify this environment when you invoke any of the TSO/E REXX routines. The environment block represents the environment in which you want the routine to run. Generally, you can specify the address of the environment block:

- Using the environment block address parameter in the routine's parameter list
- In register 0.

If you specify the environment block address in the parameter list, TSO/E REXX uses the address you specify and ignores the contents of register 0. However, TSO/E does not validate the address you specify in the parameter list. Therefore, you must ensure that you pass a correct address or unpredictable results may occur. For more information, see “Using the environment block address parameter” on page 243.

If you do not specify an address in the environment block address parameter, the TSO/E REXX routine checks register 0 for the address of an environment block. If register 0 contains the address of a valid environment block, the routine runs in the environment represented by that environment block. If the address is not valid, the routine locates the current non-reentrant environment and runs in that environment. If register 0 contains a 0, the routine immediately searches for the last non-reentrant environment created, thereby eliminating the processing required to check whether register 0 contains a valid environment block address.

If you use IRXINIT to initialize reentrant environments, see “Using the environment block for reentrant environments” on page 244 for information about running in reentrant environments.

Using the environment block address parameter

The parameter lists of most of the TSO/E REXX routines contain the *environment block address parameter*. This parameter lets you specify the address of the environment block that represents the environment in which you want the routine to run. If you use the environment block address parameter, the routine uses the address you specify and ignores the contents of register 0. Additionally, the routine does not check the address you specify. Therefore, you must ensure that you pass a correct environment block address or unpredictable results may occur. For example, if you specify an invalid address, the routine may return with a return code of 28, which indicates a language processor environment could not be located. In other cases, processing could abend.

You could also specify an address for an environment that exists, but the address may be for a different environment than the one you want to use. In this case, the routine may run successfully, but the results will not be what you expected. For example, suppose you have four environments initialized in an address space; environments 1, 2, 3, and 4. You want to invoke the trace and execution control routine, IRXIC, to halt the interpretation of execs in environment 2. However, when you invoke IRXIC, you specify the address of the environment block for environment 4, instead of environment 2. IRXIC completes successfully, but the interpretation of execs is halted in environment 4, rather than in environment 2. This is a subtle problem that may be difficult to determine. Therefore, if you use the environment block address parameter, you must ensure the address you specify is correct.

If you do not want to pass an address in the environment block address parameter, specify a value of 0. Also, the parameter lists for the TSO/E REXX routines are of variable length. That is, register 1 points to a list of addresses and each address in the list points to a parameter. The end of the parameter list is indicated by the high-order bit being on in the last address in the parameter list. If you do not want to use the environment block address parameter and there are no other parameters after it that you want to use, you can simply end the parameter list at a preceding parameter. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

If you are using the environment block address parameter and you are having problems debugging an application, you may want to set the parameter to 0 for debugging purposes. This lets you determine whether any problems are a result of this parameter being specified incorrectly.

Using the environment block for reentrant environments

If you want to use a reentrant environment, you must explicitly call the initialization routine, IRXINIT, to initialize the environment. TSO/E REXX automatically initializes non-reentrant environments only. When you invoke IRXINIT to initialize a reentrant environment, you must set the RENTRANT flag on (see "Flags and corresponding masks" on page 322).

An application program would use a reentrant environment when it wants to isolate itself and its characteristics from other application programs. For example, an application program may provide a storage management routine, but does not want any other program to use the storage management routine. To ensure this, you would use IRXINIT to initialize the environment and set the RENTRANT flag on. When the RENTRANT flag is on, the environment is not added to the existing chain of environments. Instead, the environment is an independent entry isolated from all other environments.

The system routines do not locate reentrant environments. Additionally, if you use IRXINIT to find an environment, IRXINIT finds non-reentrant environments only, not reentrant environments. You can use a reentrant environment that you have initialized only by explicitly passing the address of the environment block for the reentrant environment when you call a TSO/E REXX programming routine. If you want to invoke a TSO/E REXX routine to run in a reentrant environment, you must pass the address of the environment block for the reentrant environment on the call to the routine. You can pass the address either in the parameter list (in the environment block address parameter) or in register 0.

If you do not explicitly pass an environment block address, the routine locates the current non-reentrant environment and runs in that environment.

Each task that is using REXX must have its own language processor environment. Two tasks cannot simultaneously use the same language processor environment for REXX processing.

Return codes for TSO/E REXX routines

The TSO/E REXX routines return a return code in register 15 that indicates whether processing was successful. The parameter lists for most of the routines also have a *return code parameter* that lets you specify a fullword field in which to receive the return code. The return code parameter lets high-level languages more easily obtain return code information. If you provide this parameter, the routine returns the return code in both the return code parameter and in register 15. If the parameter list you pass to the routine is invalid, the return code is returned in register 15 only.

Each TSO/E REXX routine has specific return codes. The individual topics in this book describe the return codes for each routine. Table 14 on page 245 shows the common return codes that most of the TSO/E REXX routines use.

Table 14. Common return codes for TSO/E REXX routines

Return code	Description
0	Successful processing.
20	Error occurred. Processing was unsuccessful. The requested service was either partially completed or was terminated. An error message may be written to the error message field in the environment block. If the NOPMSGS flag is off for the environment, the message is also written to the output DD that is defined for the environment or to the terminal. For some errors, an alternate message may also be issued. Alternate messages are printed only if the ALTMSGS flag is on for the environment. The NOPMSGS and ALTMSGS flags are described in the topic “Flags and corresponding masks” on page 322. If multiple errors occurred and multiple error messages were issued, all error messages are written to the output DD or to the terminal. Additionally, the first error message is stored in the environment block.
28	A service was requested, but a valid language processor environment could not be located. The requested service is not performed.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Exec processing routines - IRXJCL and IRXEXEC

This topic provides information about the IRXJCL and IRXEXEC routines, which you can use to run REXX execs. You can use IRXJCL to run a REXX exec in MVS batch from JCL. You can also call IRXJCL from a REXX exec or a program that is running in any address space to run an exec.

You can call the IRXEXEC routine from a REXX exec or program that is running in any address space to run an exec. IRXEXEC provides more flexibility than IRXJCL. With IRXJCL, you can pass the name of the exec and one argument on the call. Using IRXEXEC, you can, for example, pass multiple arguments or preload the exec in storage.

The following topics describe each routine. If you use either IRXJCL or IRXEXEC to run a REXX exec in TSO/E foreground or background, note that you cannot invoke the REXX exec as authorized.

Note: To permit FORTRAN programs to call IRXEXEC, TSO/E provides an alternate entry point for the IRXEXEC routine. The alternate entry point name is IRXEX.

The IRXJCL routine

You can use IRXJCL to run a REXX exec in MVS batch. You can also call IRXJCL from a REXX exec or a program in any address space to run an exec.

A program can access IRXJCL using either the CALL or LINK macro instructions, specifying IRXJCL as the entry point name. You can obtain the address of the IRXJCL routine from the REXX vector of external entry points. “Format of the REXX vector of external entry points” on page 362 describes the vector.

Using IRXJCL to run a REXX exec in MVS batch

To run an exec in MVS batch, specify IRXJCL as the program name (PGM=) on the JCL EXEC statement. Specify the member name of the exec and one argument you want to pass to the exec in the PARM field on the EXEC statement. You can specify only the name of a member of a PDS. You cannot specify the name of a sequential data set. The PDS must be allocated to the DD specified in the LOADDD field of the module name table. The default is SYSEXEC. Figure 15 on page 246 shows example JCL to invoke the exec MYEXEC.

```
//STEP1    EXEC PGM=IRXJCL,PARM='MYEXEC A1 b2 C3 d4'
//*
//STEPLIB
//* Next DD is the data set equivalent to terminal input
//SYSTSIN  DD   DSN=xxx.xxx.xxx,DISP=SHR,...
//*
//* Next DD is the data set equivalent to terminal output
//SYSTSPRT DD   DSN=xxx.xxx.xxx,DISP=OLD,...
//*
//* Next DD points to a library of execs
//* that include MYEXEC
//SYSEXEC  DD   DSN=xxx.xxx.xxx,DISP=SHR
```

Figure 15. Example of invoking an exec from a JCL EXEC statement using IRXJCL

Note: If you want output to be routed to a printer, specify the //SYSTSPRT DD statement as:

```
//SYSTSPRT DD   SYSOUT=A
```

As Figure 15 on page 246 shows, the exec MYEXEC is loaded from DD SYSEXEC. SYSEXEC is the default setting for the name of the DD from which an exec is to be loaded. In the example, one argument is passed to the exec. The argument can consist of more than one token. In this case, the argument is:

```
A1 b2 C3 d4
```

When the PARSE ARG keyword instruction is processed in the exec (for example, PARSE ARG EXVARS), the value of the variable EXVARS is set to the argument specified on the JCL EXEC statement. The variable EXVARS is set to:

```
A1 b2 C3 d4
```

The MYEXEC exec can perform any of the functions that an exec running in a non-TSO/E address space can perform. See “Writing execs that run in Non-TSO/E address spaces” on page 189 for more information about the services you can use in execs that run in non-TSO/E address spaces.

IRXJCL returns a return code as the step completion code. However, the step completion code is limited to a maximum of 4095, in decimal. If the return code is greater than 4095 (decimal), the system uses the rightmost three digits of the hexadecimal representation of the return code and converts it to decimal for use as the step completion code. See “Return codes” on page 249 for more information.

Using IRXJCL to execute an in-stream REXX exec

When IRXJCL is invoked, R1 points to a parameter string consisting of a halfword length followed by the data itself. The data usually contains the name of the exec to execute, followed optionally by any parameter string data to be passed to the exec.

The first non-blank name within the parm data is typically verified to be a 1-8 character name. That name is used as the member name of a REXX exec to be loaded from the SYSEXEC file. In this case, the SYSEXEC file must be a partitioned data set. Any non-blank data following the member name is then passed to the REXX exec as a single parameter string.

When the first non-blank name in the parm data passed to IRXJCL is either a single x'00' character or a 2 to 8 character name consisting of x'00' characters, that name is treated as a null-name.

If the exec name is a null-name, then there is no member name to be loaded. The SYSEXEC file is checked to verify that it represents a sequential data set. If it is a sequential file, then it is loaded and executed as a sequential REXX exec. Any non-blank data that might have followed the 1-8 character null-name is treated as a parameter string to be passed to the SEQ REXX exec.

Example: The user wants to invoke an in-stream REXX exec. The exec is defined within the JCL stream as in-stream data following the "//SYSEXEC DD *" JCL card. In this example, we are also passing the parameter string "MY_INPUT_DATA_TO_REXX" to the in-stream REXX exec.

```
//SEQLEXEC JOB /
//****************************************************************
//* JCL to demonstrate use of 1-byte hex-zero char as null-name
//* to be used to execute an in-stream SYSEXEC file as a SEQ
//* in-stream REXX exec.
//*
//* The '~' character within the PARM="~ ..." parameter represents
//* a null character, (x'00').
//*
//* If your in-stream REXX exec contains any comment starting with
//* a '/*', the '/*' should not start in column 1 anywhere within
//* the in-stream exec.
//****************************************************************
//STEP1 EXEC PGM=IRXJCL,PARM='~ MY_INPUT_DATA_TO_REXX'
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD DUMMY
//SYSEXEC DD *
  /* REXX : In-line sequential exec */
  PARSE ARG INPUT
  SAY "HELLO WORLD! DATE = "DATE()", TIME = "TIME()
  SAY "INPUT TO THIS EXEC IS <"INPUT">"
  PARSE SOURCE SRC_STRING
  SAY "SOURCE_STRING => <"SRC_STRING">"
  EXIT 0
/*
```

Output from this exec might be displayed as follows:

```
HELLO WORLD! DATE = 13 Jun 2021, TIME = 13:26:26
INPUT TO THIS EXEC IS <MY_INPUT_DATA_TO_REXX>
SOURCE_STRING => <TSO COMMAND ? SYSEXEC ?   MVS MVS ?>
```

Invoking IRXJCL from a REXX exec or a program

You can also call IRXJCL from an exec or a program to run a REXX exec. On the call to IRXJCL, you pass the address of a parameter list in register 1.

Environment Customization Considerations: If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXJCL to run. On the call to IRXJCL, you can optionally specify the address of the environment block for the environment in register 0.

If you do not pass an environment block address or if IRXJCL determines the address is not valid, IRXJCL locates the current environment and runs in that environment. "Chains of environments and how environments are located" on page 340 describes how environments are located. If a current environment does not exist or the current environment was initialized on a different task and the TSOFL flag is off in that environment, a new language processor environment is initialized. The exec runs in the new environment. Before IRXJCL returns, the language processor environment that was created is terminated. Otherwise, it runs in the located current environment.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the address of the environment block" on page 243.

Entry specifications

For the IRXJCL routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of one address. The high-order bit of the address in the parameter list must be set to 1 to indicate the end of the parameter list. Table 15 on page 248 describes the parameter for IRXJCL.

Table 15. Parameter for calling the IRXJCL routine

Parameter	Number of bytes	description
Parameter 1	variable	A buffer, which consists of a halfword length field followed by a data field. The first two bytes of the buffer is the length field that contains the length of the data that follows. The length does not include the two bytes that specify the length itself. The data field contains the name of the exec, followed by one or more blanks, followed by the argument (if any) to be passed to the exec. You can pass only one argument on the call.

Figure 16 on page 248 shows an example PL/I program that invokes IRXJCL to run a REXX exec. Note that the example is for PL/I Version 2.

```
JCLXMP1 : Procedure Options (Main);
/* Function: Call a REXX exec from a PL/I program using IRXJCL        */

  DCL IRXJCL EXTERNAL OPTIONS(RETCODE, ASSEMBLER);
  DCL 1 PARM_STRUCT,               /* Parm to be passed to IRXJCL     */
        5 PARM_LNG BIN FIXED (15), /* Length of the parameter         */
        5 PARM_STR CHAR (30);      /* String passed to IRXJCL         */
  DCL PLIRETV BUILTIN;             /* Defines the return code built-in*/
  PARM_LNG = LENGTH(PARM_STR);     /* Set the length of string        */
/*                                                                    */
  PARM_STR = 'JCLXMP2 This is an arg to exec'; /* Set string value
                                      In this case, call the exec named
                                      JCLXMP2 and pass argument:
                                      'This is an arg to exec'        */
  FETCH IRXJCL;                    /* Load the address of entry point */
  CALL IRXJCL (PARM_STRUCT);       /* Call IRXJCL to execute the REXX
                                      exec and pass the argument      */
  PUT SKIP EDIT ('Return code from IRXJCL was:', PLIRETV) (a, f(4));
                                   /* Print out the return code from
                                      exec JCLXMP2.                   */
  END ;                            /* End of program                  */
```

Figure 16. Example PL/I version 2 program using IRXJCL

Return specifications

For the IRXJCL routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

If IRXJCL encounters an error, it returns a return code. If you invoke IRXJCL from JCL to run an exec in MVS batch, IRXJCL returns the return code as the step condition code. If you call IRXJCL from an exec or program, IRXJCL returns the return code in register 15. Table 16 on page 249 describes the return codes.

Table 16. Return codes for IRXJCL routine

Return code	Description
0	Processing was successful. Exec processing completed.
20	Processing was not successful. The exec was not processed.
20021	An invalid parameter was specified on the JCL EXEC statement or the parameter list passed on the call to IRXJCL was incorrect. Some possible errors could be that a parameter was either blank or null or the name of the exec was not valid (more than eight characters long). If you run an exec in MVS batch and a return code of 20021 is returned, the value 3637, in decimal, is returned as the step completion code. For more information, see note “2” on page 249 below.
Other	Any other return code not equal to 0, 20, or 20021 is the return code from the REXX exec on the RETURN or EXIT keyword instruction. For more information, see the two notes below.

Note:

1. No distinction is made between the REXX exec returning a value of 0, 20, or 20021 on the RETURN or EXIT instruction and IRXJCL returning a return code of 0, 20, or 20021.
2. IRXJCL returns a return code as the step completion code. However, the step completion code is limited to a maximum of 4095, in decimal. If the return code is greater than 4095 (decimal), the system uses the rightmost three digits of the hexadecimal representation of the return code and converts it to decimal for use as the step completion code. For example, suppose the exec returns a return code of 8002, in decimal, on the RETURN or EXIT instruction. The value 8002 (decimal) is X'1F42' in hexadecimal. The system takes the rightmost three digits of the hexadecimal value (X'F42') and converts it to decimal (3906) to use as the step completion code. The step completion code that is returned is 3906, in decimal.

The IRXEXEC routine

Use the IRXEXEC routine to run an exec in any MVS address space.

Tip: To permit FORTRAN programs to call IRXEXEC, TSO/E provides an alternate entry point for the IRXEXEC routine. The alternate entry point name is IRXEX.

Most users do not need to use IRXEXEC. In TSO/E, you can invoke execs implicitly or explicitly using the TSO/E EXEC command. You can also run execs in TSO/E background. If you want to invoke an exec from a program that is written in a high-level programming language, you can use the TSO service facility to invoke the EXEC command. You can run an exec in MVS batch using JCL and the IRXJCL routine.

You can also call the IRXJCL routine from a REXX exec or a program that is running in any address space to invoke an exec. However, the IRXEXEC routine gives you more flexibility. For example, you can preload the REXX exec in storage and pass the address of the preloaded exec to IRXEXEC. This is useful if you want to run an exec multiple times to avoid the exec being loaded and freed whenever it is invoked. You may also want to use your own load routine to load and free the exec.

If you use the TSO/E EXEC command, you can pass only one argument to the exec. The argument can consist of several tokens. Similarly, if you call IRXJCL from an exec or program, you can only pass one argument. By using IRXEXEC, you can pass multiple arguments to the exec and each argument can consist of multiple tokens. If you pass multiple arguments, you must not set bit 0 (the command bit) in parameter 3.

Note: Use the EXEC command to invoke a REXX exec that uses ISPF services. The EXEC command allows ISPF variables to be resolved. There are cases where a REXX exec invoked using the programming routines IRXJCL and IRXEXEC does not have access to ISPF variables. For more information about ISPF variables, see *z/OS ISPF Services Guide*.

A program can access IRXEXEC using either the CALL or LINK macro instructions, specifying IRXEXEC as the entry point name. You can obtain the address of the IRXEXEC routine from the REXX vector of external entry points. "Format of the REXX vector of external entry points" on page 362 describes the vector.

If you use IRXEXEC, one parameter on the call is the user field. You can use this field for your own processing.

Environment Customization Considerations: If you use the IRXINIT initialization routine to initialize language processor environments, the following information provides several considerations about calling IRXEXEC.

When you call IRXEXEC, you can specify the environment in which you want IRXEXEC to run. On the call to IRXEXEC, you can optionally specify the address of the environment block for the environment in one of the following:

- the ECTENVBK field of the ECT. The ECT address is in the command processor parameter list (CPPL). Parameter 5 of the IRXEXEC routine specifies the address of the CPPL. For more information about the address of the CPPL, see the description of parameter 5 in Table 17 on page 251.
- the parameter list
- register 0.

If you do not pass an environment block address or IRXEXEC determines the address is not valid, IRXEXEC locates the current environment and runs in that environment. "Chains of environments and how environments are located" on page 340 describes how environments are located. If a current environment does not exist or the current environment was initialized on a different task and the TSOFL flag is off in that environment, a new language processor environment is initialized. The exec runs in the new environment. Before IRXEXEC returns, the language processor environment that was created is terminated. Otherwise, it runs in the located current environment.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the address of the environment block" on page 243.

Entry specifications

For the IRXEXEC routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see “Parameter lists for TSO/E REXX routines” on page 242.

Table 17 on page 251 describes the parameters for IRXEXEC.

Table 17. Parameters for IRXEXEC routine

Parameter	Number of bytes	Description
Parameter 1	4	Specifies the address of the exec block (EXECBLK). The exec block is a control block that describes the exec to be loaded. It contains information needed to process the exec, such as the DD from which the exec is to be loaded and the name of the initial host command environment when the exec starts running. “The exec block (EXECBLK)” on page 254 describes the format of the exec block. If the exec is preloaded and you pass the address of the preloaded exec in parameter 4, specify an address of 0 for this parameter. If you specify both parameter 1 and parameter 4, IRXEXEC uses the value in parameter 4 and ignores this parameter (parameter 1).
Parameter 2	4	Specifies the address of the arguments for the exec. The arguments are arranged as a vector of address/length pairs followed by X'FFFFFFFFFFFFFFFF'. “Format of argument list” on page 255 describes the format of the arguments.

Table 17. Parameters for IRXEXEC routine (continued)

Parameter	Number of bytes	Description
Parameter 3	4	A fullword of bits that IRXEXEC uses as flags. IRXEXEC uses bits 0, 1, 2, and 3 only. The remaining bits are reserved. Bits 0, 1, and 2 are mutually exclusive. PARSE SOURCE returns a token indicating how the exec was invoked. The bit you set on in bit positions 0, 1, or 2 indicates the token that PARSE SOURCE uses. For example, if you set bit 2 on, PARSE SOURCE returns the token *SUBROUTINE*. If you set bit 1 on, the exec must return a result. If you set either bit 0 or 2 on, the exec can optionally return a result. Use bit 3 to indicate how IRXEXEC should return information about a syntax error in the exec. The description of each bit is as follows: • Bit 0 - This bit must be set on if the exec is being invoked as a "command"; that is, the exec is not being invoked from another exec as an external function or subroutine. If you pass more than one argument to the exec, do not set bit 0 on. • Bit 1 - This bit must be set on if the exec is being invoked as an external function (a function call). • Bit 2 - This bit must be set on if the exec is being invoked as a subroutine for example, when the CALL keyword instruction is used. • Bit 3 - This bit must be set on if you want IRXEXEC to return *extended return codes* in the range 20001–20099. If a syntax error occurs, IRXEXEC returns a value in the range 20001–20099 in the evaluation block, regardless of the setting of bit 3. If bit 3 is on and a syntax error occurs, IRXEXEC returns with a return code in the range 20001–20099 that matches the value returned in the evaluation block. If bit 3 is off and a syntax error occurs, IRXEXEC returns with return code 0. For more information, see "How IRXEXEC returns information about syntax errors" on page 260.
Parameter 4	4	Specifies the address of the *in-storage control block* (INSTBLK), which defines the structure of a preloaded exec in storage. The INSTBLK contains pointers to each statement in the exec and the length of each statement. "The in-storage control block (INSTBLK)" on page 256 describes the control block. This parameter is required if the caller of IRXEXEC has preloaded the exec. Otherwise, this parameter must be 0. If you specify this parameter, IRXEXEC ignores parameter 1 (address of the exec block).
Parameter 5	4	Specifies the address of the command processor parameter list (CPPL) if you call IRXEXEC from the TSO/E address space. If you do not pass the address of the CPPL (you specify an address of 0), TSO/E builds the CPPL without a command buffer. If you call IRXEXEC from a non-TSO/E address space, specify an address of 0.

Table 17. Parameters for IRXEXEC routine (continued)

Parameter	Number of bytes	Description
Parameter 6	4	Specifies the address of an evaluation block (EVALBLOCK). IRXEXEC uses the evaluation block to return the result from the exec that was specified on either the RETURN or EXIT instruction. "The evaluation block (EVALBLOCK)" on page 259 describes the format of the evaluation block, how IRXEXEC uses the parameter, and whether you should provide an EVALBLOCK on the call. If you do not want to provide an evaluation block, specify an address of 0. If you do not provide an evaluation block, you must use the get result routine, IRXRLT, to obtain the result from the exec.
Parameter 7	4	Specifies the address of an 8-byte field that defines a work area for the IRXEXEC routine. In the 8-byte field, the: • First four bytes contain the address of the work area • Second four bytes contain the length of the work area. The work area is passed to the language processor to use for processing the exec. If the work area is too small, IRXEXEC returns with a return code of 20 and a message is issued that indicates an error. The minimum length required for the work area is X'1800' bytes. If you do not want to pass a work area, specify an address of 0. In this case, IRXEXEC obtains storage for its work area or calls the replaceable storage routine specified in the GETFREER field for the environment, if you provided a storage routine.
Parameter 8	4	Specifies the address of a user field. IRXEXEC does not use or check this pointer or the user field. You can use this field for your own processing. If you do not want to use a user field, specify an address of 0.
Parameter 9	4	The address of the environment block that represents the environment in which you want IRXEXEC to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, IRXEXEC uses the value you specify and ignores register 0. However, IRXEXEC does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the address of the environment block" on page 243.
Parameter 10	4	A 4-byte field that IRXEXEC uses to return the return code. The return code parameter is optional. If you use this parameter, IRXEXEC returns the return code in the parameter and also in register 15. Otherwise, IRXEXEC uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 261 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

The exec block (EXECBLK)

The exec block (EXECBLK) is a control block that describes the exec to be loaded. If the exec is not preloaded, you must build the exec block and pass the address in parameter 1 on the call to IRXEXEC. You need not pass an exec block if the exec is preloaded.

Note: If you want to preload the exec, you can use the system-supplied exec load routine IRXLOAD or your own exec load replaceable routine (see "Exec load routine" on page 392).

TSO/E provides a mapping macro IRXEXECB for the exec block. The mapping macro is in SYS1.MACLIB. Table 18 on page 254 describes the format of the exec block.

Table 18. Format of the exec block (EXECBLK)

Offset (decimal)	Number of bytes	Field name	Description
0	8	ACRYN	An eight-character field that identifies the exec block. It must contain the character string 'IRXEXECB'.
8	4	LENGTH	Specifies the length of the exec block in bytes.
12	4	—	Reserved.
16	8	MEMBER	Specifies the member name of the exec if the exec is in a partitioned data set. If the exec is in a sequential data set, this field must be blank.
24	8	DDNAME	Specifies the name of the DD from which the exec is loaded. An exec cannot be loaded from a DD that has not been allocated. The ddname you specify must be allocated to a data set containing REXX execs or to a sequential data set that contains an exec. If this field is blank, the exec is loaded from the DD specified in the LOADDD field of the module name table (see "Module name table" on page 326). The default is SYSEXEC.
32	8	SUBCOM	Specifies the name of the initial host command environment when the exec starts running. If this field is blank, the environment specified in the INITIAL field of the host command environment table is used. For TSO/E and ISPF, the default is TSO. For a non-TSO/E address space, the default is MVS. The table is described in "Host command environment table" on page 331.
40	4	DSNPTR	Specifies the address of a data set name that the PARSE SOURCE instruction returns. The name usually represents the name of the exec load data set. The name can be up to 54 characters long (44 characters for the fully qualified data set name, 8 characters for the member name, and 2 characters for the left and right parentheses). If you do not want to specify a data set name, specify an address of 0.
44	4	DSNLEN	Specifies the length of the data set name that is pointed to by the address at offset +40. The length can be 0-54. If no data set name is specified, the length is 0.

Table 18. Format of the exec block (EXECBLK) (continued)

Offset (decimal)	Number of bytes	Field name	Description
48	4	EXTNAME_PTR	Pointer to the extended execname. This field can be used to pass an execname if greater than eight characters. For example, this field may be used to pass *pathname/filename* of a UNIX file to a replaceable load routine that handles UNIX file system files (This name is not used by the TSO/E REXX load routine.) (This field is only valid if PTF for APAR OW28404 is applied.)
52	4	EXTNAME_LEN	Length of the extended name pointed to by EXTNAME_PTR, or 0 if no extended name is specified. The maximum length of an extended name is 4096 (x'1000'). Any length larger than this maximum value should be treated as 0 (that is, as no extended name specified).
56	8	—	Reserved

An exec cannot be loaded from a data set that has not been allocated. The ddname you specify (at offset +24 in the exec block) must be allocated to a data set containing REXX execs or to a sequential data set that contains an exec.

The fields at offset +40 and +44 in the exec block are used only for input to the PARSE SOURCE instruction and are for informational purposes only.

Loading of the exec is done as follows:

- If the exec is preloaded, loading is not performed.
- If you specify a ddname in the exec block, IRXEXEC loads the exec from that DD. You also specify the name of the member in the exec block.
- If you do not specify a ddname in the exec block, IRXEXEC loads the exec from the DD specified in the LOADDD field in the module name table for the language processor environment (see “Module name table” on page 326). The default is SYSEXEC. If you customize the environment values TSO/E provides or use the initialization routine IRXINIT, the DD may be different. See Chapter 14, “Language processor environments,” on page 311 for customizing information.

Format of argument list

Parameter 2 points to the arguments for the exec. The arguments are arranged as a vector of address/length pairs, one for each argument. The first four bytes are the address of the argument string. The second four bytes are the length of the argument string, in bytes. The vector must end in X'FFFFFFFFFFFFFFFF'. There is no limit on the number of arguments you can pass. Table 19 on page 255 shows the format of the argument list. TSO/E provides a mapping macro IRXARGTB for the vector. The mapping macro is in SYS1.MACLIB.

Table 19. Format of the argument list

Offset (dec)	Number of bytes	Field name	Description
0	4	ARGSTRING_PTR	Address of argument 1
4	4	ARGSTRING_LENGTH	Length of argument 1
8	4	ARGSTRING_PTR	Address of argument 2
12	4	ARGSTRING_LENGTH	Length of argument 2

Table 19. Format of the argument list (continued)

Offset (dec)	Number of bytes	Field name	Description
16	4	ARGSTRING_PTR	Address of argument 3
20	4	ARGSTRING_LENGTH	Length of argument 3

⋮ ⋮

Offset (dec)	Number of bytes	Field name	Description
x	4	ARGSTRING_PTR	Address of argument n
x+4	4	ARGSTRING_LENGTH	Length of argument n
x+8	8	—	X'FFFFFFFFFFFFFFFF'

The in-storage control block (INSTBLK)

Parameter 4 points to the in-storage control block (INSTBLK). The in-storage control block defines the structure of a preloaded exec in storage. The INSTBLK contains pointers to each record in the exec and the length of each record.

If you preload the exec in storage, you must pass the address of the in-storage control block (parameter 4). You must provide the storage, format the control block, and free the storage after IRXEXEC returns. IRXEXEC only reads information from the in-storage control block. IRXEXEC does not change any of the information.

To preload an exec into storage, you can use the exec load replaceable routine IRXLOAD. If you provide your own exec load replaceable routine, you can use your routine to preload the exec. "Exec load routine" on page 392 describes the replaceable routine.

If the exec is not preloaded, you must specify an address of 0 for the in-storage control block parameter (parameter 4).

The in-storage control block consists of a header and the records in the exec, which are arranged as a vector of address/length pairs. Table 20 on page 256 shows the format of the in-storage control block header. Table 21 on page 258 shows the format of the vector of records. TSO/E provides a mapping macro IRXINSTB for the in-storage control block. The mapping macro is in SYS1.MACLIB.

Table 20. Format of the header for the in-storage control block

Offset (decimal)	Number of bytes	Field name	Description
0	8	ACRONYM	An eight-character field that identifies the control block. The field must contain the characters 'IRXINSTB'.
8	4	HDRLEN	Specifies the length of the in-storage control block header only. The value must be 128 bytes.
12	4	—	Reserved.
16	4	ADDRESS	Specifies the address of the vector of records. See Table 21 on page 258 for the format of the address/length pairs. If this field is 0, the exec contains no records.

Table 20. Format of the header for the in-storage control block (continued)

Offset (decimal)	Number of bytes	Field name	Description
20	4	USEDLEN	Specifies the length of the address/length vector of records in bytes. This is not the number of records. The value is the number of records multiplied by 8. If this field is 0, the exec contains no records.
24	8	MEMBER	Specifies the name of the exec. This is the name of the member in the partitioned data set from which the exec was loaded. If the exec was loaded from a sequential data set, this field is blank. If the exec was loaded using an extended execname specification (as pointed to by EXTNAME_PTR) this field can be left blank. (See the EXTNAME_PTR field below.) The PARSE SOURCE instruction returns the folded member name in token3 of the PARSE SOURCE string. If this field is blank or if an extended execname is specified then the name that PARSE SOURCE returns in token3 is either: 1. A question mark (?), if no extended name is specified. 2. The extended execname pointed to by EXTNAME_PTR, if specified. An extended name is not folded to uppercase within the PARSE SOURCE string. Any blanks in the extended name are changed to null characters (x'00') when the extended name is placed in the PARSE SOURCE string. **Note:** If EXTNAME_PTR and MEMBER are both specified, EXTNAME_PTR is used to build the PARSE SOURCE string token3.
32	8	DDNAME	Specifies the name of the DD that represents the exec load data set from which the exec was loaded.
40	8	SUBCOM	Specifies the name of the initial host command environment when the exec starts running.
48	4	—	Reserved.
52	4	DSNLEN	Specifies the length of the data set name that is specified at offset +56. If a data set name is not specified, this field must be 0.
56	54	DSNAME	A 54-byte field that contains the name of the data set, if known, from which the exec was loaded. The name can be up to 54 characters long (44 characters for the fully qualified data set name, 8 characters for the member name, and 2 characters for the left and right parentheses).
110	2	—	Reserved.

Table 20. Format of the header for the in-storage control block (continued)

Offset (decimal)	Number of bytes	Field name	Description
112	4	EXTNAME_PTR	Pointer to the extended execname. The extended execname can be used instead of the MEMBER field to return the exec name of the loaded exec if the name is longer than eight characters or is case sensitive. For example, this field can be used to return the *pathname/filename* specification of an exec loaded from a UNIX file. If specified, the PARSE SOURCE instruction returns the name pointed to by this field, without folding to uppercase, instead of the MEMBER name. (Any blanks within an extended name are changed to null characters (x'00') when moved into the PARSE SOURCE string.) See the discussion of PARSE SOURCE under MEMBER field above. (**Note:** The extended execname is not currently used by default TSO/E REXX load routine). (This field is valid only if PTF for OW28404 is applied.)
116	4	EXTNAME_LEN	Length of extended execname pointed to by EXTNAME_PTR, or 0 if no extended name is specified. The maximum length of an extended name is 4096 (x'1000'). If a length larger than the maximum value is specified, the extended name is ignored.
120	8	—	Reserved.

At offset +16 in the in-storage control block header, the field points to the vector of records that are in the exec. The records are arranged as a vector of address/length pairs. Table 21 on page 258 shows the format of the address/length pairs.

The addresses point to the text of the record to be processed. This can be one or more REXX clauses, parts of a clause that are continued with the REXX continuation character (the continuation character is a comma), or a combination of these. The address is the actual address of the record. The length is the length of the record in bytes.

Table 21. Vector of records for the in-storage control block

Offset (decimal)	Number of bytes	Field name	Description
0	4	STMT@	Address of record 1
4	4	STMTLEN	Length of record 1
8	4	STMT@	Address of record 2
12	4	STMTLEN	Length of record 2
16	4	STMT@	Address of record 3
20	4	STMTLEN	Length of record 3
	⋮	⋮	

Offset (decimal)	Number of bytes	Field name	Description
x	4	STMT@	Address of record n
x+4	4	STMTLEN	Length of record n

The evaluation block (EVALBLOCK)

The evaluation block is a control block that IRXEXEC uses to return the result from the exec. The exec can return a result on either the RETURN or EXIT instruction. For example, the REXX instruction

```
RETURN var1
```

returns the value of the variable VAR1. IRXEXEC returns the value of VAR1 in the evaluation block.

If the exec you are running will return a result, specify the address of an evaluation block when you call IRXEXEC (parameter 6). You must obtain the storage for the control block yourself.

If the exec does not return a result or you want to ignore the result, you need not allocate an evaluation block. On the call to IRXEXEC, you must pass all of the parameters. Therefore, specify an address of 0 for the evaluation block.

If the result from the exec fits into the evaluation block, the data is placed into the block (EVDATA field) and the length of the block is updated (ENVLEN field). If the result does not fit into the area provided in the evaluation block, IRXEXEC:

- Places as much of the result that will fit into the evaluation block in the EVDATA field
- Sets the length of the result field (EVLEN) to the negative of the length that is required to store the complete result.

The result is not lost. The system has its own evaluation block that it uses to store the result. If the evaluation block you passed to IRXEXEC is too small to hold the complete result, you can then use the IRXRLT (get result) routine. Allocate another evaluation block that is large enough to hold the result and call IRXRLT. On the call to the IRXRLT routine, you pass the address of the new evaluation block. IRXRLT copies the result from the exec that was stored in the system's evaluation block into your evaluation block and returns. "Get result routine - IRXRLT" on page 286 describes the routine in more detail.

If you call IRXEXEC and do not pass the address of an evaluation block, and the exec returns a result, you can use the IRXRLT routine after IRXEXEC completes to obtain the result.

To summarize, if you call IRXEXEC to run an exec that returns a result and you pass the address of an evaluation block that is large enough to hold the result, IRXEXEC returns the result in the evaluation block. In this case, IRXEXEC does not store the result in its own evaluation block.

If IRXEXEC runs an exec that returns a result, the result is stored in the system's evaluation block if:

- The result did not fit into the evaluation block that you passed on the call to IRXEXEC
- You did not specify the address of an evaluation block on the call

You can then obtain the result by allocating a large enough evaluation block and calling the IRXRLT routine to get the result. The result is available until one of the following occurs:

- IRXRLT is called and successfully obtains the result
- Another REXX exec runs in the same language processor environment, or
- The language processor environment is terminated

The language processor environment is the environment in which the language processor processes the exec. See Chapter 14, "Language processor environments," on page 311 for more information about the initialization and termination of environments and customization services.

The evaluation block consists of a header and data, which contains the result. Table 22 on page 260 shows the format of the evaluation block. Additional information about each field is described after the table.

TSO/E provides a mapping macro IRXEVALB for the evaluation block. The mapping macro is in SYS1.MACLIB.

Table 22. Format of the evaluation block

Offset (decimal)	Number of bytes	Field name	Description
0	4	EVPAD1	A fullword that must contain X'00'. This field is reserved and is not used.
4	4	EVSIZE	Specifies the total size of the evaluation block in doublewords.
8	4	EVLEN	On entry, this field is not used and must be set to X'00'. On return, it specifies the length of the result, in bytes, that is returned. The result is returned in the EVDATA field at offset +16.
12	4	EVPAD2	A fullword that must contain X'00'. This field is reserved and is not used.
16	n	EVDATA	The field in which IRXEXEC returns the result from the exec. The length of the field depends on the total size specified for the control block in the EVSIZE field. The total size of the EVDATA field is: `EVSIZE * 8 - 16` It is suggested that you use 250 bytes for the EVDATA field. For information about the values IRXEXEC returns, if the language processor detects a syntax error in the exec, see "How IRXEXEC returns information about syntax errors" on page 260.

If the result does not fit into the EVDATA field, IRXEXEC stores as much of the result as it can into the field and sets the length field (EVLEN) to the negative of the required length for the result. You can then use the IRXRLT routine to obtain the result. See "Get result routine - IRXRLT" on page 286 for more information.

On return, if the result has a length of 0, the length field (EVLEN) is 0, which means the result is null. If no result is returned on the EXIT or RETURN instruction, the length field contains X'80000000'.

If you invoke the exec as a "command" (bit 0 is set on in parameter 3), the result the exec returns must be a numeric value. The result can be from -2,147,483,648 to +2,147,483,647. If the result is not numeric or is greater than or less than the valid values, this indicates a syntax error and the value 20026 is returned in the EVDATA field.

How IRXEXEC returns information about syntax errors

If the language processor detects a syntax error in the exec, IRXEXEC returns the following:

- A value of 20000 plus the REXX error number in the EVDATA field of the evaluation block
- A value of 5 for the length of the result in the EVLEN field of the evaluation block

The REXX error numbers are in the range 1-99. Therefore, the range of values that IRXEXEC can return for a syntax error are 20001–20099. The REXX error numbers correspond to the REXX message numbers. For example, error 26 corresponds to the REXX message IRX0026I. For error 26, IRXEXEC returns the value 20026 in the EVDATA field. The REXX error messages are described in *z/OS TSO/E Messages*.

The exec you run may also return a value on the RETURN or EXIT instruction in the range 20001–20099. IRXEXEC returns the value from the exec in the EVDATA field of the evaluation block. To determine whether the value in the EVDATA field is the value from the exec or the value related to a syntax error, use bit 3 in parameter 3 of the parameter list. Bit 3 lets you enable the extended return codes in the range 20001–20099.

If you set bit 3 off, and the exec processes successfully but the language processor detects a syntax error, the following occurs. IRXEXEC returns a return code of 0 in register 15. IRXEXEC also returns a value of 20000 plus the REXX error number in the EVDATA field of the evaluation block. In this case, you cannot determine whether the exec returned the 200xx value or whether the value represents a syntax error.

If you set bit 3 on and the exec processes successfully but the language processor detects a syntax error, the following occurs. IRXEXEC sets a return code in register 15 equal to 20000 plus the REXX error message. That is, the return code in register 15 is in the range 20001–20099. IRXEXEC also returns the 200xx value in the EVDATA field of the evaluation block. If you set bit 3 on and the exec processes without a syntax error, IRXEXEC returns with a return code of 0 in register 15. If IRXEXEC returns a value of 20001–20099 in the EVDATA field of the evaluation block, that value must be the value that the exec returned on the RETURN or EXIT instruction.

By setting bit 3 on in parameter 3 of the parameter list, you can check the return code from IRXEXEC to determine whether a syntax error occurred.

Return specifications

For the IRXEXEC routine, the contents of the registers on return are:

Register 0

Address of the environment block.

If IRXEXEC returns with return code 100 or 104, register 0 contains the abend and reason code. "Return codes" on page 261 describes the return codes and how IRXEXEC returns the abend and reason codes for return codes 100 and 104.

Registers 1-14

Same as on entry

Register 15

Return code

Return codes

Table 23 on page 262 shows the return codes for the IRXEXEC routine. IRXEXEC returns the return code in register 15. If you specify the return code parameter (parameter 10), IRXEXEC also returns the return code in the parameter.

Table 23. IRXEXEC return codes

Return code	Description
0	Processing was successful. The exec has completed processing. If the exec returns a result, the result may or may not fit into the evaluation block. You must check the length field (EVLEN). On the call to IRXEXEC, you can set bit 3 in parameter 3 of the parameter list to indicate how IRXEXEC should handle information about syntax errors. If IRXEXEC returns with return code 0 and bit 3 is on, the language processor did not detect a syntax error. In this case, the value IRXEXEC returns in the EVDATA field of the evaluation block is the value the exec returned. If IRXEXEC returns with return code 0 and bit 3 is off, the language processor may or may not have detected a syntax error. If IRXEXEC returns a value of 20001–20099 in the evaluation block, you cannot determine whether the value represents a syntax error or the value was returned by the exec. For more information, see "How IRXEXEC returns information about syntax errors" on page 260.
20	Processing was not successful. An error occurred. The exec has not been processed. The system issues an error message that describes the error.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.
36	Processing was not successful. Either: • A CPPL was passed (parm5) with an ECT address of 0, or • Both an environment block address and a CPPL were passed to IRXEXEC, but the ECT associated with the passed environment block does not match the ECT whose address was passed in the CPPL. For information about how an ECT is associated with a REXX environment, see the description of parameter 10 passed to IRXINIT during REXX environment initialization in Table 68 on page 373.
100	Processing was not successful. A system abend occurred during IRXEXEC processing. The system issues one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. The abend code is returned in the two low-order bytes of register 0. The abend reason code is returned in the high-order two bytes of register 0. If the abend reason code is greater than two bytes, only the two low-order bytes of the abend reason code are returned. See *z/OS MVS System Codes* for information about the abend codes and reason codes.
104	Processing was not successful. A user abend occurred during IRXEXEC processing. The system issues one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. The abend code is returned in the two low-order bytes of register 0. The abend reason code is returned in the two high-order bytes of register 0. If the abend reason code is greater than two bytes, only the two low-order bytes of the abend reason code are returned. See *z/OS MVS System Codes* for information about the abend codes and reason codes.

Table 23. IRXEXEC return codes (continued)

Return code	Description
20001–20099	Processing was successful. The exec completed processing, but the language processor detected a syntax error. The return code that IRXEXEC returns in register 15 is the value 20000 plus the REXX error number. The REXX error numbers are from 1 to 99 and correspond to the REXX message numbers. For example, error 26 corresponds to the REXX message IRX0026I. The REXX error messages are described in *z/OS TSO/E Messages*. IRXEXEC returns a return code of 20001–20099 only if bit 3 in parameter 3 is set on when you call IRXEXEC. IRXEXEC also returns the same 200xx value in the EVDATA field of the evaluation block. For more information about syntax errors, see "How IRXEXEC returns information about syntax errors" on page 260.

The language processor environment is the environment in which the exec runs. If IRXEXEC cannot locate an environment in which to process the exec, an environment is automatically initialized. If an environment was being initialized and an error occurred during the initialization process, IRXEXEC returns with return code 20, but an error message is not issued.

External functions and subroutines, and function packages

You can write your own external functions and subroutines, which allow you to extend the capabilities of the REXX language. You can write external functions or subroutines that supplement the built-in functions or TSO/E external functions that are provided. You can also write a function to replace one of the functions that is provided. For example, if you want a new substring function that performs differently from the SUBSTR built-in function, you can write your own substring function and name it STRING. Users at your installation can then use the STRING function in their execs.

You can write external functions or subroutines in REXX. You can store the exec containing the function or subroutine in:

- The same PDS from which the calling exec is loaded
- An alternative exec library as defined by ALTLIB (TSO/E address space only).
- A data set that is allocated to SYSEXEC (SYSEXEC is the default load ddname used for storing REXX execs)
- A data set that is allocated to SYSPROC (TSO/E address space only).

You can also write an external function or subroutine in assembler or a high-level programming language. You can then store the function or subroutine in a load library, which allows for faster access of the function or subroutine. By default, load libraries are searched before any exec libraries, such as SYSEXEC and SYSPROC. The language in which you write the exec must support the system-dependent interfaces that the language processor uses to invoke the function or subroutine.

For faster access of a function or subroutine, and therefore better performance, you can group frequently used external functions and subroutines in *function packages*. A function package is basically a number of external functions and subroutines that are grouped or *packaged* together. To include an external function or subroutine in a function package, the function or subroutine must be link-edited into a load module. If you write a function or subroutine as a REXX exec and the exec is interpreted (that is, the TSO/E REXX interpreter executes the exec), you cannot include the function or subroutine in a function package. However, if you write the function or subroutine in REXX and the REXX exec is compiled, you can include the exec in a function package because the compiled exec can be link-edited into a load module. For information about compiled execs, see the appropriate compiler publications.

When the language processor is processing an exec and encounters a function call or a call to a subroutine, the language processor searches the function packages before searching load libraries or exec libraries, such as SYSEXEC and SYSPROC. “Search order” on page 74 describes the complete search order.

The topics in this section describe:

- The system-dependent interfaces that the language processor uses to invoke external functions or subroutines. If you write a function or subroutine in a programming language other than REXX, the language must support the interface.
- How to define function packages.

Interface for writing external function and subroutine code

This topic describes the system interfaces for writing external functions and subroutines that are load modules. You can write the function or subroutine in assembler or any high-level programming language that can be invoked by using the MVS Link Macro and is capable of handling the REXX function interface as documented below (that documentation is in terms of an assembler programming interface). This means that the high-level language must be capable of properly handling the documented register interface, including the handling of parameters passed by using register 1 pointing to an EFPL parameter list, exactly as shown.

Note that for some high-level languages, it may be necessary to write an assembler stub routine which can take the defined REXX function interface (registers, parameters, AMODE, etc.) and manipulate these into a format which can be handled by the specific high-level language routine that is being invoked. For example, for C programs you may need to use an assembler-to-C interface stub to first create a C environment before the stub can pass control to your C-routine. Or, you may need to pass parameters in some different way which your high-level language understands. Refer to language specific documentation for a description of any language specific interface requirements, or for a description of how an assembler stub program can pass control to a routine in the high-level language.

The interface to the code is the same whether the code is called as a function or as a subroutine. The only difference is how the language processor handles the result after your code completes and returns control to the language processor. Before your code gets control, the language processor allocates a control block called the evaluation block (EVALBLOCK). The address of the evaluation block is passed to the function or subroutine code. The function or subroutine code places the result into the evaluation block, which is returned to the language processor. If the code was called as a subroutine, the result in the evaluation block is placed into the REXX special variable RESULT. If the code was called as a function, the result in the evaluation block is used in the interpretation of the REXX instruction that contained the function.

An external function or subroutine receives the address of an environment block in register 0. This environment block address should be passed on any TSO/E REXX programming services invoked from the external function or subroutine. This is particularly important if the environment is reentrant because TSO/E REXX programming services cannot automatically locate a reentrant environment. For more information about reentrant environments, see “Using the environment block for reentrant environments” on page 244.

The following topics describe the contents of the registers when the function or subroutine code gets control and the parameters the code receives.

Entry specifications

The code for the external function or subroutine receives control in an unauthorized state. The contents of the registers are:

Register 0
Address of the environment block of the exec that invoked the external function or subroutine.

Register 1
Address of the external function parameter list (EFPL)

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

When the external function or subroutine gets control, register 1 points to the external function parameter list (EFPL). Table 24 on page 265 describes the parameter list. TSO/E provides a mapping macro, IRXEFPL, for the external function parameter list. The mapping macro is in SYS1.MACLIB.

Table 24. External function parameter list

Offset (decimal)	Number of bytes	Description
0	4	Reserved.
4	4	Reserved.
8	4	Reserved.
12	4	Reserved.
16	4	An address that points to the parsed argument list. Each argument is represented by an address/length pair. The argument list is terminated by X'FFFFFFFFFFFFFFFF'. Table 25 on page 265 shows the format of the argument list. If there were no arguments included on the function or subroutine call, the address points to X'FFFFFFFFFFFFFFFF'.
20	4	An address that points to a fullword. The fullword contains the address of an evaluation block (EVALBLOCK). You use the evaluation block to return the result of the function or subroutine. Table 26 on page 266 describes the evaluation block.

Argument list

Table 25 on page 265 shows the format of the parsed argument list the function or subroutine code receives at offset +16 (decimal) in the external function parameter list. The figure is an example of three arguments. TSO/E provides a mapping macro IRXARGTB for the argument list. The mapping macro is in SYS1.MACLIB.

Table 25. Format of the argument list — three arguments

Offset (dec)	Number of bytes	Field name	Description
0	4	ARGSTRING_PTR	Address of argument 1
4	4	ARGSTRING_LENGTH	Length of argument 1
8	4	ARGSTRING_PTR	Address of argument 2
12	4	ARGSTRING_LENGTH	Length of argument 2
16	4	ARGSTRING_PTR	Address of argument 3
20	4	ARGSTRING_LENGTH	Length of argument 3
24	8	—	X'FFFFFFFFFFFFFFFF'

In the argument list, each argument consists of the address of the argument and its length. The argument list is terminated by X'FFFFFFFFFFFFFFFF'.

If an argument is omitted, the address of the argument is 0. If a null string is passed as an argument to a function, the address of the argument is nonzero, but the length of the argument is 0. For example, suppose you invoked an assembler function called "MYFUNC" using this invocation:

```
Z = MYFUNC(,'','ABC')
```

The argument list would contain the addresses and lengths of the three arguments followed by X'FFFFFFFFFFFFFFFF':

- The first argument is an omitted argument, also know as a dummy argument, so it is indicated by an address of 0.
- The second argument is a null string, so it is indicated by a nonzero address and a length of 0.
- The third argument is string 'ABC', so it is indicated by a pointer to the characters 'ABC' and a length of 3.

Evaluation block

Before the function or subroutine code is called, the language processor allocates a control block called the evaluation block (EVALBLOCK). The address of a fullword containing the address of the evaluation block is passed to your function or subroutine code at offset +20 in the external function parameter list. The function or subroutine code computes the result and returns the result in the evaluation block.

The evaluation block consists of a header and data, in which you place the result from your function or subroutine code. Table 26 on page 266 shows the format of the evaluation block.

TSO/E provides a mapping macro IRXEVALB for the evaluation block. The mapping macro is in SYS1.MACLIB.

The IRXEXEC routine also uses an evaluation block to return the result from an exec that is specified on either the RETURN or EXIT instruction. The format of the evaluation block that IRXEXEC uses is identical to the format of the evaluation block passed to your function or subroutine code. “The evaluation block (EVALBLOCK)” on page 259 describes the control block for IRXEXEC.

Table 26. Format of the evaluation block

Offset (decimal)	Number of bytes	Field name	Description
0	4	EVPAD1	A fullword that contains X'00'. This field is reserved and is not used.
4	4	EVSIZE	Specifies the total size of the evaluation block in doublewords. This field can be used by the function to determine the size of the EVDATA area available to return data (as shown below), but this field should NOT be changed by the function.
8	4	EVLEN	On entry, this field is set to X'80000000', which indicates no result is currently stored in the evaluation block. On return, specify the length of the result, in bytes, that your code is returning. The result is returned in the EVDATA field at offset +16.
12	4	EVPAD2	A fullword that contains X'00'. This field is reserved and is not used.

Table 26. Format of the evaluation block (continued)

Offset (decimal)	Number of bytes	Field name	Description
16	n	EVDATA	The field in which you place the result from the function or subroutine code. The length of the field depends on the total size specified for the control block in the EVSIZE field. The total size of the EVDATA field is: `EVSIZE * 8 - 16`

The function or subroutine code must compute the result, move the result into the EVDATA field (at offset +16), and update the EVLEN field (at offset +8) to the length of the result being returned in EVDATA. The function should NOT change the EVSIZE field of the evaluation block which is passed to the function. The EVDATA field of the evaluation block that TSO/E passes to your code is 256 bytes. Because the evaluation block is passed to the function or subroutine code, the EVDATA field in the evaluation block may be too small to hold the complete result. If the evaluation block is too small, you can call the IRXRLT (get result) routine to obtain a larger evaluation block. Call IRXRLT using the GETBLOCK function. IRXRLT creates the new evaluation block and returns the address of the new block. Your code can then place the result in the new evaluation block. You must also change the parameter at offset +20 in the external function parameter list to point to the new evaluation block. For information about using IRXRLT, see "Get result routine - IRXRLT" on page 286.

Functions must return a result. Subroutines may optionally return a result. If a subroutine does not return a result, it must return a data length of X'80000000' in the EVLEN field in the evaluation block.

Return specifications

When your function or subroutine code returns control, the contents of the registers must be:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

Your function or subroutine code must return a return code in register 15. Table 27 on page 267 shows the return codes.

Table 27. Return codes from function or subroutine code (in register 15)

Return code	Description
0	Function or subroutine code processing was successful. If the called routine is a function, the function must return a value in the EVDATA field of the evaluation block. The value replaces the function call. If the function does not return a result in the evaluation block, a syntax error occurs with error number 44. If the called routine is a subroutine, the subroutine can optionally return a value in the EVDATA field of the evaluation block. The REXX special variable RESULT is set to the returned value.
Non-zero	Function or subroutine code processing was not successful. The language processor stops processing the REXX exec that called your function or subroutine with an error code of 40, unless you trap the error with a SYNTAX trap.

Function packages

Function packages are basically several external functions and subroutines that are grouped or *packaged* together. When the language processor processes a function call or a call to a subroutine, the language processor searches the function packages before searching load libraries or exec libraries, such as SYSEXEC and SYSPROC. Grouping frequently used external functions and subroutines in a function package allows for faster access to the function and subroutine, and therefore, better performance. "Search order" on page 74 describes the complete search order the language processor uses to locate a function or subroutine.

TSO/E supports three types of function packages. Basically, there are no differences between the three types, although the intent of the design is as follows:

- User packages, which are function packages that an individual user may write to replace or supplement certain system-provided functions. When the function packages are searched, the user packages are searched before the local and system packages.
- Local packages, which are function packages that a system support group or application group may write. Local packages may contain functions and subroutines that are available to a specific group of users or to the entire installation. Local packages are searched after the user packages and before the system packages.
- System packages, which are function packages that an installation may write for system-wide use or for use in a particular language processor environment. System packages are searched after any user and local packages.

To provide function packages, there are several steps you must perform:

1. You must first write the individual external functions and subroutines you want to include in a function package. If you want to include an external function or subroutine in a function package, the function or subroutine must be link-edited into a load module. If you write the function or subroutine in REXX and the REXX exec is interpreted (that is, the TSO/E REXX interpreter executes the exec), you cannot include the function or subroutine in a function package. However, if you write the external function or subroutine in REXX and the REXX exec is compiled, you can include the function or subroutine in a function package because the compiled exec can be link-edited into a load module. For information about compiled execs, see the appropriate compiler publications.

   If you write the external function or subroutine in a programming language other than REXX, the language you use must support the system-dependent interfaces that the language processor uses to invoke the function or subroutine. "Interface for writing external function and subroutine code" on page 264 describes the interfaces.

2. After you write the individual functions and subroutines, you must write the directory for the function package. You need a directory for each individual function package.

   The function package directory is contained in a load module. The directory contains a header followed by individual entries that define the names and/or the addresses of the entry points of your function or subroutine code. "Directory for function packages" on page 269 describes the directory for function packages.

3. The name of the entry point at the beginning of the directory (the function package name) must be specified in the function package table for a language processor environment. "Function package table" on page 333 describes the format of the table. After you write the directory, you must define the directory name in the function package table. There are several ways you can do this depending on the type of function package you are defining (user, local, or system) and whether you are providing only one or several user and local function packages.

   If you are providing a local or user function package, you can name the function package directory IRXFLOC (local package) or IRXFUSER (user package). TSO/E provides these two "dummy" directory names in the three default parameters modules IRXPARMS, IRXTSPRM, and IRXISPRM. By naming your local function package directory IRXFLOC and your user function package directory IRXFUSER, the external functions and subroutines in the packages are automatically available to REXX execs that run in non-TSO/E and the TSO/E address space.

If you write your own system function package or more than one local or user function package, you must provide a function package table containing the name of your directory. You must also provide your own parameters module that points to your function package table. Your parameters module then replaces the default parameters module that the system uses to initialize a default language processor environment. "Specifying directory names in the function package table" on page 273 describes how to define directory names in the function package table.

Note: If you explicitly call the IRXINIT routine, you can pass the address of a function package table containing your directory names on the call.

TSO/E provides the IRXEFMVS and IRXEFPCK system function packages. The two function packages provide the TSO/E external functions, which are described in "TSO/E external functions" on page 109. The IRXEFMVS and IRXEFPCK system function packages are defined in the default parameters modules TSO/E provides (see "Values provided in the three default parameters modules" on page 336).

Other IBM products may also provide system function packages that you can use for REXX processing in TSO/E and MVS. If you install a product that provides a system function package for TSO/E REXX, you must change the function package table and provide your own parameters modules. The product itself supplies the individual functions in the function package and the directory for their function package. To use the functions, you must do the following:

1. Change the function package table. The function package table contains information about the user, local, and system function packages for a particular language processor environment. Table 61 on page 334 shows the format of the table. Add the name of the function package directory to the entries in the table. You must also change the SYSTEM_TOTAL and SYSTEM_USED fields in the table header (offsets +28 and +32). Increment the value in each field by 1 to indicate the additional function package supplied by the IBM product.
2. Provide your own IRXTSPRM, IRXISPRM, or IRXPARMS parameters module. The function package table is part of the parameters module that the system uses to initialize language processor environments. You need to code one or more parameters modules depending on whether you want the function package available to REXX execs that run in ISPF only, TSO/E only, TSO/E and ISPF, non-TSO/E only, or any address space.

   Chapter 14, "Language processor environments," on page 311 describes environments, their characteristics, and the format of the parameters modules. In the same chapter, "Changing the default values for initializing an environment" on page 344 describes how to provide your own parameters modules.

Directory for function packages

After you write the code for the functions and subroutines you want to group in a function package, you must write a directory for the function package. You need a directory for each individual function package you want defined.

The function package directory is contained in a load module. The name of the entry point at the beginning of the directory is the function package directory name. The name of the directory is specified only on the CSECT. In addition to the name of the entry point, the function package directory defines each entry point for the individual functions and subroutines that are part of the function package. The directory consists of two parts; a header followed by individual entries for each function and subroutine included in the function package. Table 28 on page 269 shows the format of the directory header. Table 29 on page 270 illustrates the rows of entries in the function package directory. TSO/E provides a mapping macro, IRXFPDIR, for the function package directory header and entries. The mapping macro is in SYS1.MACLIB.

Table 28. Format of the function package directory header

Offset (decimal)	Number of bytes	Description
0	8	An 8-byte character field that must contain the character string 'IRXFPACK'.

Table 28. Format of the function package directory header (continued)

Offset (decimal)	Number of bytes	Description
8	4	Specifies the length, in bytes, of the header. This is the offset from the beginning of the header to the first entry in the directory. This must be a fullword binary number equivalent to decimal 24.
12	4	The number of functions and subroutines defined in the function package (the number of rows in the directory). The format is a fullword binary number.
16	4	A fullword of X'00'.
20	4	Specifies the length, in bytes, of an entry in the directory (length of a row). This must be a fullword binary number equivalent to decimal 32.

In the function package table for the three default parameters modules (IRXPARMS, IRXTSPRM, and IRXISPRM), TSO/E provides two "dummy" function package directory names:

- IRXFLOC for a local function package
- IRXFUSER for a user function package

If you create a local or user function package, you can name the directory IRXFLOC and IRXFUSER, respectively. By using IRXFLOC and IRXFUSER, you need not create a new function package table containing your directory names.

If you are creating a system function package or several local or user packages, you must define the directory names in a function package table. “Specifying directory names in the function package table” on page 273 describes how to do this in more detail.

You must link-edit the external function or subroutine code and the directory for the function package into a load module. You can link-edit the code and directory into separate load modules or into the same load module. Place the data set with the load modules in the search sequence for an MVS LOAD. For example, the data set can be in the data set concatenation for either a STEPLIB or JOBLIB, or you can install the data set in the LINKLST or LPALIB. Refer to subsection “Programming considerations” on page 273 for information about how to link-edit these load modules.

Recommendation: For best performance, link-edit the code for individual functions or subroutines in the same load module as the function package directory. Because the function package directory is always loaded during REXX environment initialization and remains in storage, the functions and subroutines are loaded once and are in storage when you need them. If the code for your external function or subroutine is link-edited into a load module separate from the function package directory, that load module will be loaded before each invocation of the function or subroutine and then deleted after that function or subroutine has completed.

In the TSO/E address space, you can use the EXECUTIL command with the RENAME operand to dynamically change entries in a function package (see “EXECUTIL” on page 216 for information about EXECUTIL). If you plan to use the EXECUTIL command to change entries in the function package you provide, you should not install the function package in the LPALIB.

Format of Entries in the Directory: Table 29 on page 270 shows two rows (two entries) in a function package directory. The first entry starts immediately after the directory header. Each entry defines a function or subroutine in the function package. The individual fields are described following the table.

Table 29. Format of entries in function package directory

Offset (decimal)	Number of bytes	Field name	Description
0	8	FUNC-NAME	The name of the first function or subroutine (entry) in the directory.

Table 29. Format of entries in function package directory (continued)

Offset (decimal)	Number of bytes	Field name	Description
8	4	ADDRESS	The address of the entry point of the function or subroutine code (for the first entry).
12	4	—	Reserved.
16	8	SYS-NAME	The name of the entry point in a load module that corresponds to the function or subroutine code (for the first entry).
24	8	SYS-DD	The ddname from which the function or subroutine code is loaded (for the first entry).
32	8	FUNC-NAME	The name of the second function or subroutine (entry) in the directory.
40	4	ADDRESS	The address of the entry point of the function or subroutine code (for the second entry).
44	4	—	Reserved.
48	8	SYS-NAME	The name of the entry point in a load module that corresponds to the function or subroutine code (for the second entry).
56	8	SYS-DD	The ddname from which the function or subroutine code is loaded (for the second entry).

The following describes each entry (row) in the directory.

FUNC-NAME
The eight character name of the external function or subroutine. This is the name that is used in the REXX exec. The name must be in uppercase, left justified, and padded to the right with blanks.

If this field is blank, the entry is ignored.

ADDRESS
A 4-byte field that contains the address, in storage, of the entry point of the function or subroutine code. This address is used only if the code has already been loaded.

If the address is 0, the sys-name and, optionally, the sys-dd fields are used. An MVS LOAD will be issued for *sys-name* from the DD *sys-dd*.

If the address is specified, the sys-name and sys-dd fields for the entry are ignored.

Reserved
A 4-byte field that is reserved.

SYS-NAME
An 8-byte character name of the entry point in a load module that corresponds to the function or subroutine code to be called for the *func-name*. The name must be in uppercase, left justified, and padded to the right with blanks.

If the address is specified, this field can be blank. If an address of 0 is specified and this field is blank, the entry is ignored.

SYS-DD
An 8-byte character name of the DD from which the function or subroutine code is loaded. The name must be in uppercase, left justified, and padded to the right with blanks.

If the address is 0 and this field is blank, the module is loaded from the link list.

Example of a Function Package Directory: Figure 17 on page 272 shows an example of a function package directory. The example is explained following the figure.

```
IRXFUSER CSECT
         DC    CL8'IRXFPACK'       String identifying directory
         DC    FL4'24'             Length of header
         DC    FL4'4'              Number of rows in directory
         DC    FL4'0'              Word of zeros
         DC    FL4'32'             Length of directory entry
*                                  Start of definition of first entry
         DC    CL8'MYF1    '       Name used in exec
         DC    FL4'0'              Address of preloaded code
         DC    FL4'0'              Reserved field
         DC    CL8'ABCFUN1 '       Name of entry point
         DC    CL8'FUNCTDD1'       DD from which to load entry point
*                                  Start of definition of second entry
         DC    CL8'MYF2    '       Name used in exec
         DC    FL4'0'              Address of preloaded code
         DC    FL4'0'              Reserved field
         DC    CL8'ABCFUN2 '       Name of entry point
         DC    CL8'        '       DD from which to load entry point
*                                  Start of definition of third entry
         DC    CL8'MYS3    '       Name used in exec
         DC    AL4(ABCSUB3)        Address of preloaded code
         DC    FL4'0'              Reserved field
         DC    CL8'        '       Name of entry point
         DC    CL8'        '       DD from which to load entry point
*                                  Start of definition of fourth entry
         DC    CL8'MYF4    '       Name used in exec
         DC    VL4(ABCFUNC4)       Address of preloaded code
         DC    FL4'0'              Reserved field
         DC    CL8'        '       Name of entry point
         DC    CL8'        '       DD from which to load entry point
          SPACE 2
ABCSUB3   EQU  *
*  Subroutine code for subroutine MYS3
*
*  End of subroutine code
         END  IRXFUSER



            - - - - - New Object Module - - - - -


ABCFUNC4 CSECT
*  Function code for function MYF4
*
*  End of function code
         END  ABCFUNC4
```

Figure 17. Example of a function package directory

In Figure 17 on page 272, the name of the function package directory is IRXFUSER, which is one of the "dummy" function package directory names TSO/E provides in the default parameters modules. Four entries are defined in this function package:

- MYF1, which is an external function
- MYF2, which is an external function
- MYS3, which is an external subroutine
- MYF4, which is an external function

If the external function MYF1 is called in an exec, the load module with entry point ABCFUN1 is loaded from DD FUNCTDD1. If MYF2 is called in an exec, the load module with entry point ABCFUN2 is loaded from the linklist because the sys-dd field is blank.

The load modules for MYS3 and MYF4 do not have to be loaded. The MYS3 subroutine has been assembled as part of the same object module as the function package directory. The MYF4 function has been assembled in a different object module, but has been link-edited as part of the same load module as the directory. The assembler, linkage editor, and loader have resolved the addresses.

If the name of the directory is not IRXFLOC or IRXFUSER, you must specify the directory name in the function package table for an environment. "Specifying directory names in the function package table" on page 273 describes how you can do this.

When a language processor environment is initialized, either by default or when IRXINIT is explicitly called, the load modules containing the function package directories for the environment are automatically loaded. External functions or subroutines that are link-edited as separate, stand-alone load modules and are not defined in any function package, are loaded before each invocation of the functions or subroutines and then deleted after that function or subroutine has completed.

For best performance, link-edit the code for individual functions or subroutines in the same load module as the function package directory. Because the function package directory is always loaded during REXX environment initialization, the functions and subroutines are loaded once and are in storage when you need them.

Specifying directory names in the function package table

After you write the function and subroutine code and the directory, you must define the directory name in the function package table. The function package table contains information about the user, local, and system function packages that are available to REXX execs running in a specific language processor environment. Each environment that is initialized has its own function package table. "Function package table" on page 333 describes the format of the table.

The parameters module (and the PARMBLOCK that is created) defines the characteristics for a language processor environment and contains the address of the function package table (in the PACKTB field). In the three default modules that TSO/E provides (IRXPARMS, IRXTSPRM, and IRXISPRM), the function package table contains two "dummy" function package directory names:

- IRXFLOC for a local function package
- IRXFUSER for a user function package

If you name your local function package directory IRXFLOC and your user function package directory IRXFUSER, the external functions and subroutines in your package are then available to execs that run in non-TSO/E, TSO/E, and ISPF. There is no need for you to provide a new function package table.

If you provide a system function package or several local or user packages, you must then define the directory name in a function package table. To do this, you must provide your own function package table. You must also provide your own IRXPARMS, IRXTSPRM, and/or IRXISPRM load module depending on whether you want the function package available to execs running in non-TSO/E, TSO/E, or ISPF.

You first write the code for the function package table. You must include the default entries provided by TSO/E. The IRXPARMS, IRXTSPRM, and IRXISPRM modules contain the default directory names IRXEFMVS, IRXFLOC, and IRXFUSER. In addition, the IRXTSPRM and IRXISPRM modules also contain the default IRXEFPCK directory name. "Function package table" on page 333 describes the format of the function package table.

You must then write the code for one or more parameters modules. The module you provide depends on whether the function package should be made available to execs that run in ISPF only, TSO/E only, TSO/E and ISPF, non-TSO/E only, or any address space. "Changing the default values for initializing an environment" on page 344 describes how to create the code for your own parameters module and which modules you should provide.

Programming considerations

Link-edit function packages and function package directories with a reusability attribute (RENT for reenterable or REUS for serially reusable). If you are going to use the RENAME operand of EXECUTIL, the function package directory must be linked as serially reusable only (not reenterable).

Variable access routine - IRXEXCOM

The language processor provides an interface whereby called commands and programs can easily access and manipulate the current generation of REXX variables. Any variable can be inspected, set, or dropped; if required, all active variables can be inspected in turn. Names are checked for validity by the interface code, and optionally substitution into compound symbols is carried out according to normal REXX rules. Certain other information about the program that is running is also made available through the interface.

TSO/E REXX provides two variable access routines you can call to access and manipulate REXX exec variables:

- IRXEXCOM
- IKJCT441

The IRXEXCOM variable access routine lets unauthorized commands and programs access and manipulate REXX variables. IRXEXCOM can be used in both the TSO/E and non-TSO/E address spaces. IRXEXCOM can be used only if a REXX exec has been *enabled for variable access* in the language processor environment. That is, an exec must have been invoked, but is not currently being processed. For example, you can invoke an exec that calls a routine and the routine can then invoke IRXEXCOM. When the routine calls IRXEXCOM, the REXX exec is *enabled for variable access*, but it is not being processed. If a routine calls IRXEXCOM and an exec has not been enabled, IRXEXCOM returns with an error.

Tip: To permit FORTRAN programs to call IRXEXCOM, TSO/E provides an alternate entry point for the IRXEXCOM routine. The alternate entry point name is IRXEXC.

A program can access IRXEXCOM using either the CALL or LINK macro instructions, specifying IRXEXCOM as the entry point name. You can obtain the address of the IRXEXCOM routine from the REXX vector of external entry points. “Format of the REXX vector of external entry points” on page 362 describes the vector.

If a program uses IRXEXCOM, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations: If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXEXCOM to run. On the call to IRXEXCOM, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see “Specifying the address of the environment block” on page 243.

The IKJCT441 routine lets authorized and unauthorized commands and programs access REXX variables. IKJCT441 can be used in the TSO/E address space only. You can use IKJCT441 to access REXX or CLIST variables depending on whether the program that calls IKJCT441 was called by a REXX exec or a CLIST. *z/OS TSO/E Programming Services* describes IKJCT441.

Entry specifications

For the IRXEXCOM routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 30 on page 275 describes the parameters for IRXEXCOM.

Table 30. Parameters for IRXEXCOM

Parameter	Number of bytes	Description
Parameter 1	8	An 8-byte character field that must contain the character string 'IRXEXCOM'.
Parameter 2	4	Parameter 2 and parameter 3 must be identical, that is, they must be at the same location in storage. This means that in the parameter list pointed to by register 1, the address at offset +4 and the address at offset +8 must be the same. Both addresses in the parameter list may be set to 0.
Parameter 3	4	Parameter 2 and parameter 3 must be identical, that is, they must be at the same location in storage. This means that in the parameter list pointed to by register 1, the address at offset +4 and the address at offset +8 must be the same. Both addresses in the parameter list may be set to 0.
Parameter 4	32	The first shared variable (request) block (SHVBLOCK) in a chain of one or more request blocks. The format of the SHVBLOCK is described in "The shared variable (request) block - SHVBLOCK" on page 276.
Parameter 5	4	The address of the environment block that represents the environment in which you want IRXEXCOM to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, IRXEXCOM uses the value you specify and ignores register 0. However, IRXEXCOM does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the address of the environment block" on page 243.

Table 30. Parameters for IRXEXCOM (continued)

Parameter	Number of bytes	Description
Parameter 6	4	A 4-byte field that IRXEXCOM uses to return the return code. The return code parameter is optional. If you use this parameter, IRXEXCOM returns the return code in the parameter and also in register 15. Otherwise, IRXEXCOM uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 279 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

The shared variable (request) block - SHVBLOCK

Parameter 4 is the first shared variable (request) block in a chain of one or more blocks. Each SHVBLOCK in the chain must have the structure shown in Figure 18 on page 276.

```
********************************************************
* SHVBLOCK:  Layout of shared-variable PLIST element
********************************************************
SHVBLOCK DSECT
SHVNEXT  DS    A     Chain pointer (0 if last block)
SHVUSER  DS    F     Available for private use, except during
*                      "Fetch Next" when it identifies the
*                      length of the buffer pointed to by SHVNAMA.
SHVCODE  DS    CL1   Individual function code indicating
*                      the type of variable access request
*                      (S,F,D,s,f,d,N, or P)
SHVRET   DS    XL1   Individual return code flags
         DS    H'0'  Reserved, should be zero
SHVBUFL  DS    F     Length of 'fetch' value buffer
SHVNAMA  DS    A     Address of variable name
SHVNAML  DS    F     Length of variable name
SHVVALA  DS    A     Address of value buffer
SHVVALL  DS    F     Length of value
SHVBLEN  EQU   *-SHVBLOCK  (length of this block = 32)
         SPACE
*
*     Function Codes (Placed in SHVCODE):
*
*     (Note that the symbolic name codes are lowercase)
SHVFETCH EQU   C'F'  Copy value of variable to buffer
SHVSTORE EQU   C'S'  Set variable from given value
SHVDROPV EQU   C'D'  Drop variable
SHVSYSET EQU   C's'  Symbolic name Set variable
SHVSYFET EQU   C'f'  Symbolic name Fetch variable
SHVSYDRO EQU   C'd'  Symbolic name Drop variable
SHVNEXTV EQU   C'N'  Fetch "next" variable
SHVPRIV  EQU   C'P'  Fetch private information
         SPACE
*
*     Return Code Flags (Stored in SHVRET):
*
SHVCLEAN EQU   X'00' Execution was OK
SHVNEWV  EQU   X'01' Variable did not exist
SHVLVAR  EQU   X'02' Last variable transferred (for "N")
SHVTRUNC EQU   X'04' Truncation occurred during "Fetch"
SHVBADN  EQU   X'08' Invalid variable name
SHVBADV  EQU   X'10' Value too long
SHVBADF  EQU   X'80' Invalid function code (SHVCODE)
```

Figure 18. Request block (SHVBLOCK)

Table 31 on page 277 describes the SHVBLOCK. TSO/E provides a mapping macro, IRXSHVB, for the SHVBLOCK. The mapping macro is in SYS1.MACLIB. The services you can perform using IRXEXCOM are specified in the SHVCODE field of each SHVBLOCK. "Function codes (SHVCODE)" on page 277 describes the values you can use.

"Return codes" on page 279 describes the return codes from the IRXEXCOM routine.

Table 31. Format of the SHVBLOCK

Offset (decimal)	Number of bytes	Field name	Description
0	4	SHVNEXT	Specifies the address of the next SHVBLOCK in the chain. If this is the only SHVBLOCK in the chain or the last one in a chain, this field is 0.
4	4	SHVUSER	Specifies the length of a buffer pointed to by the SHVNAMA field. This field is available for the user's own use, except for a "FETCH NEXT" request. A FETCH NEXT request uses this field.
8	1	SHVCODE	A 1-byte character field that specifies the function code, which indicates the type of variable access request. "Function codes (SHVCODE)" on page 277 describes the valid codes.
9	1	SHVRET	Specifies the return code flag, whose values are shown in Figure 18 on page 276.
10	2	---	Reserved.
12	4	SHVBUFL	Specifies the length of the "Fetch" value buffer.
16	4	SHVNAMA	Specifies the address of the variable name.
20	4	SHVNAML	Specifies the length of the variable name. The maximum length of a variable name is 250 characters.
24	4	SHVVALA	Specifies the address of the value buffer.
28	4	SHVVALL	Specifies the length of the value. This is set for a "Fetch".

Function codes (SHVCODE)

The function code is specified in the SHVCODE field in the SHVBLOCK.

Three function codes (S, F, and D) may be given either in lowercase or in uppercase:

Lowercase
(The **Symbolic** interface). The names must be valid REXX symbols (in mixed case if desired), and normal REXX substitution will occur in compound variables.

Uppercase
(The **Direct** interface). No substitution or case translation takes place. Simple symbols must be valid REXX variable names (that is, in uppercase and not starting with a digit or a period), but in compound symbols **any** characters (including lowercase, blanks, and so on) are permitted following a valid REXX stem.

Recommendation: The **Direct** interface should be used in preference to the **Symbolic** interface whenever generality is desired.

The other function codes, N and P, must always be given in uppercase. The specific actions for each function code are as follows:

S and s

Set variable. The SHVNAMA/SHVNAML address/length pair describes the name of the variable to be set, and SHVVALA/SHVVALL describes the value which is to be assigned to it. The name is validated to ensure that it does not contain invalid characters, and the variable is then set from the value given. If the name is a stem, all variables with that stem are set, just as though this were a REXX assignment. SHVNEWV is set if the variable did not exist before the operation.

F and f

Fetch variable. The SHVNAMA/SHVNAML address/length pair describes the name of the variable to be fetched. SHVVALA specifies the address of a buffer into which the data is copied, and SHVBUFL contains the length of the buffer. The name is validated to ensure that it does not contain invalid characters, and the variable is then located and copied to the buffer. The total length of the variable is put into SHVVALL, and if the value was truncated (because the buffer was not big enough), the SHVTRUNC bit is set. If the variable is shorter than the length of the buffer, no padding takes place. If the name is a stem, the initial value of that stem (if any) is returned.

SHVNEWV is set if the variable did not exist before the operation. In this case, the value copied to the buffer is the derived name of the variable (after substitution, and so on) — see "Compound symbols" on page 19.

D and d

Drop variable. The SHVNAMA/SHVNAML address/length pair describes the name of the variable to be dropped. SHVVALA/SHVVALL are not used. The name is validated to ensure that it does not contain invalid characters, and the variable is then dropped, if it exists. If the name given is a stem, all variables starting with that stem are dropped.

N

Fetch Next variable. This function may be used to search through all the variables known to the language processor (that is, all those of the current generation, excluding those "hidden" by PROCEDURE instructions). The order in which the variables are revealed is not specified.

The language processor maintains a pointer to its list of variables, which is reset to point to the first variable in the list whenever:

- A host command is issued, or
- Any function other than "N" is processed using the IRXEXCOM interface.

Whenever an N (Next) function is processed, the name and value of the next variable available are copied to two buffers supplied by the caller.

SHVNAMA specifies the address of a buffer into which the name is to be copied, and SHVUSER contains the length of that buffer. The total length of the name is put into SHVNAML, and if the name was truncated (because the buffer was not big enough) the SHVTRUNC bit is set. If the name is shorter than the length of the buffer, no padding takes place. The value of the variable is copied to the user's buffer area using exactly the same protocol as for the Fetch operation.

If SHVRET has SHVLVAR set, the end of the list of known variables has been found, the internal pointers have been reset, and no valid data has been copied to the user buffers. If SHVTRUNC is set, either the name or the value has been truncated.

By repeatedly executing the N function (until the SHVLVAR flag is set), a user program may locate all the REXX variables of the current generation.

P

Fetch private information. This interface is identical to the F fetch interface, except that the name refers to certain fixed information items that are available to the current REXX program. Only the first letter of each name is checked (though callers should supply the whole name), and the following names are recognized:

ARG

Fetch primary argument string. The first argument string that would be parsed by the ARG instruction is copied to the user's buffer.

SOURCE
Fetch source string. The source string, as described for PARSE SOURCE on page "PARSE" on page 57, is copied to the user's buffer.

VERSION
Fetch version string. The version string, as described for PARSE VERSION on page "PARSE" on page 57, is copied to the user's buffer.

Return specifications

For the IRXEXCOM routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

The output from IRXEXCOM is stored in each SHVBLOCK.

Return codes

Table 32 on page 279 shows the return codes for the IRXEXCOM routine. IRXEXCOM returns the return code in register 15. If you specify the return code parameter (parameter 6), IRXEXCOM also returns the return code in the parameter.

Figure 18 on page 276 shows the return code flags that are stored in the SHVRET field in the SHVBLOCK.

Table 32. Return codes from IRXEXCOM (in register 15)

Return code	Description
-2	Processing was not successful. Insufficient storage was available for a requested SET. Processing was terminated. Some of the request blocks (SHVBLOCKs) may not have been processed and their SHVRET bytes will be unchanged.
-1	Processing was not successful. Entry conditions were not valid for one of the following reasons: • The values in the parameter list may have been incorrect, for example, parameter 2 and parameter 3 may not have been identical • A REXX exec was not currently running • Another task is accessing the variable pool • A REXX exec is currently running, but is not enabled for variable access.
0	Processing was successful.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.
n	Any other return code not equal to -2, -1, 0, 28, or 32 is a composite formed by the logical OR of SHVRETs, excluding SHVNEWV and SHVLVAR.

Maintain entries in the host command environment table - IRXSUBCM

Use the IRXSUBCM routine to maintain entries in the host command environment table. The table contains the names of the valid host command environments that REXX execs can use to process host commands. In an exec, you can use the ADDRESS instruction to direct a host command to a specific environment for processing. The host command environment table also contains the name of the routine that is invoked to handle the processing of commands for each specific environment. "Host command environment table" on page 331 describes the table in more detail.

Tip: To permit FORTRAN programs to call IRXSUBCM, TSO/E provides an alternate entry point for the IRXSUBCM routine. The alternate entry point name is IRXSUB.

Using IRXSUBCM, you can add, delete, update, or query entries in the table. You can also use IRXSUBCM to dynamically update the host command environment table while a REXX exec is running.

A program can access IRXSUBCM using either the CALL or LINK macro instructions, specifying IRXSUBCM as the entry point name. You can obtain the address of the IRXSUBCM routine from the REXX vector of external entry points. "Format of the REXX vector of external entry points" on page 362 describes the vector.

If a program uses IRXSUBCM, it must create a parameter list and pass the address of the parameter list in register 1.

IRXSUBCM changes or queries the host command environment table for the current language processor environment, that is, for the environment in which it runs (see "General considerations for calling TSO/E REXX routines" on page 240 for information). IRXSUBCM affects only the environment in which it runs. Changes to the table take effect immediately and remain in effect until the language processor environment is terminated.

Environment Customization Considerations: If you use the IRXINIT initialization routine to initialize language processors, you can specify the environment in which you want IRXSUBCM to run. On the call to IRXSUBCM, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the address of the environment block" on page 243.

If the environment in which IRXSUBCM runs is part of a chain of environments and you use IRXSUBCM to change the host command environment table, the following applies:

- The changes do not affect the environments that are higher in the chain or existing environments that are lower in the chain.
- The changes are propagated to any language processor environment that is created on the chain after IRXSUBCM updates the table.

Entry specifications

For the IRXSUBCM routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 33 on page 281 describes the parameters for IRXSUBCM.

Table 33. Parameters for IRXSUBCM

Parameter	Number of bytes	Description
Parameter 1	8	The function to be performed. The name of the function must be left justified and padded to the right with blanks. The valid functions are: • ADD • DELETE • UPDATE • QUERY Each function is described after the table in "Functions" on page 282.
Parameter 2	4	The address of a string. On both input and output, the string has the same format as an entry in the host command environment table. "Format of a host command environment table entry" on page 282 describes the entry in more detail.
Parameter 3	4	The length of the string (entry) that is pointed to by parameter 2.
Parameter 4	8	The name of the host command environment. The name must be left justified and padded to the right with blanks. The host command environment name can contain alphabetic (a-z, A-Z), national (@, $, #), or numeric (0-9) characters and is translated to uppercase before it is stored in the host command table.
Parameter 5	4	The address of the environment block that represents the environment in which you want IRXSUBCM to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, IRXSUBCM uses the value you specify and ignores register 0. However, IRXSUBCM does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the address of the environment block" on page 243.

Table 33. Parameters for IRXSUBCM (continued)

Parameter	Number of bytes	Description
Parameter 6	4	A 4-byte field that IRXSUBCM uses to return the return code. The return code parameter is optional. If you use this parameter, IRXSUBCM returns the return code in the parameter and also in register 15. Otherwise, IRXSUBCM uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 283 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

Functions

Parameter 1 contains the name of the function IRXSUBCM is to perform. The functions are:

ADD
Adds an entry to the table using the values specified on the call. IRXSUBCM does not check for duplicate entries. If a duplicate entry is added and then IRXSUBCM is called to delete the entry, IRXSUBCM deletes the duplicate entry and leaves the original one.

DELETE
Deletes the last occurrence of the specified entry from the table.

UPDATE
Updates the specified entry with the new values specified on the call. The entry name itself (the name of the host command environment) is not changed.

QUERY
Returns the values associated with the last occurrence of the entry specified on the call.

Format of a host command environment table entry

Parameter 2 points to a string that has the same format as an entry (row) in the host command environment table. Table 34 on page 282 shows the format of an entry. TSO/E provides a mapping macro IRXSUBCT for the table entries. The mapping macro is in SYS1.MACLIB. "Host command environment table" on page 331 describes the table in more detail.

Table 34. Format of an entry in the host command environment table

Offset (decimal)	Number of bytes	Field name	Description
0	8	NAME	The name of the host command environment. The name must contain alphabetic (a-z, A-Z), national (@, $, #), or numeric (0-9) characters and is translated to uppercase before it is stored in the host command table.

Table 34. Format of an entry in the host command environment table (continued)

Offset (decimal)	Number of bytes	Field name	Description
8	8	ROUTINE	The name of the *host command environment routine* that is invoked to handle the processing of host commands in the specified environment. The host command environment routine is one of the *replaceable routines*. See "Host command environment routine" on page 412 for information about writing the routine. The routine must contain alphabetic (a-z, A-Z), national (@, $, #), or numeric (0-9) characters, must begin with an alphabetic or national character, and is translated to uppercase before it is stored in the host command table.
16	16	TOKEN	A user token that is passed to the routine when it is invoked.

For the ADD, UPDATE, and QUERY functions, the length of the string (parameter 3) must be the length of the entry.

For the DELETE function, the address of the string (parameter 2) and the length of the string (parameter 3) must be 0.

Return specifications

For the IRXSUBCM routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

Table 35 on page 283 shows the return codes for the IRXSUBCM routine. IRXSUBCM returns the return code in register 15. If you specify the return code parameter (parameter 6), IRXSUBCM also returns the return code in the parameter.

Table 35. Return codes for IRXSUBCM

Return code	Description
0	Processing was successful.
8	Processing was not successful. The specified entry was not found in the table. A return code of 8 is used only for the DELETE, UPDATE, and QUERY functions.
20	Processing was not successful. An error occurred. A message that explains the error is also issued.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Note: IRXSUBCM does not support the use of DBCS characters in host command environment names.

Trace and execution control routine - IRXIC

Use the IRXIC routine to control the tracing and execution of REXX execs. A program can call IRXIC to use the following REXX immediate commands:

- HI (Halt Interpretation) — to halt the interpretation of REXX execs
- HT (Halt Typing) — to suppress terminal output that REXX execs generate
- RT (Resume Typing) — to restore terminal output you previously suppressed
- TS (Trace Start) — to start tracing of REXX execs
- TE (Trace End) — to end tracing of REXX execs

The immediate commands are described in Chapter 10, “TSO/E REXX commands,” on page 201.

A program can access IRXIC using either the CALL or LINK macro instructions, specifying IRXIC as the entry point name. You can obtain the address of the IRXIC routine from the REXX vector of external entry points. “Format of the REXX vector of external entry points” on page 362 describes the vector.

If a program uses IRXIC, the program must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations: If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXIC to run. On the call to IRXIC, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see “Specifying the address of the environment block” on page 243.

IRXIC affects only the language processor environment in which it runs.

Entry specifications

For the IRXIC routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see “Parameter lists for TSO/E REXX routines” on page 242.

Table 36 on page 285 describes the parameters for IRXIC.

Table 36. Parameters for IRXIC

Parameter	Number of bytes	Description
Parameter 1	4	The address of the name of the command you want IRXIC to process. The valid command names are HI, HT, RT, TS, and TE. The command names are described below.
Parameter 2	4	The length of the command name that parameter 1 points to.
Parameter 3	4	The address of the environment block that represents the environment in which you want IRXIC to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, IRXIC uses the value you specify and ignores register 0. However, IRXIC does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the address of the environment block" on page 243.
Parameter 4	4	A 4-byte field that IRXIC uses to return the return code. The return code parameter is optional. If you use this parameter, IRXIC returns the return code in the parameter and also in register 15. Otherwise, IRXIC uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 286 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

The valid command names that you can specify are:

HI (Halt Interpretation)
The halt condition is set. Between instructions, the language processor checks whether it should halt the interpretation of REXX execs. If HI has been issued, the language processor stops interpreting REXX execs. HI is reset if a halt condition is enabled or when no execs are running in the environment.

HT (Halt Typing)
When the halt typing condition is set, output that REXX execs generate is suppressed (for example, the SAY instruction does not display its output). HT does not affect output from any other part of the system and does not affect error messages. HT is reset when the last exec running in the environment ends.

RT (Resume Typing)
Resets the halt typing condition. Output from REXX execs is restored.

TS (Trace Start)
Starts tracing of REXX execs.

TE (Trace End)
Ends tracing of REXX execs.

Return specifications

For the IRXIC routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

Table 37 on page 286 shows the return codes for the IRXIC routine. IRXIC returns the return code in register 15. If you specify the return code parameter (parameter 4), IRXIC also returns the return code in the parameter.

Table 37. Return codes for IRXIC

Return code	Description
0	Processing was successful.
20	Processing was not successful. An error occurred. The system issues a message that explains the error.
28	Processing was not successful. IRXIC could not locate a language processor environment.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Get result routine - IRXRLT

Use the IRXRLT (get result) routine to obtain:

- The result from an exec that was processed by calling the IRXEXEC routine.

  You can call the IRXEXEC routine to run a REXX exec. The exec can return a result using the RETURN or EXIT instruction. When you call IRXEXEC, you can optionally pass the address of an evaluation block that you have allocated. If the exec returns a result, IRXEXEC places the result in the evaluation block. "The IRXEXEC routine" on page 249 describes IRXEXEC in detail.

  The evaluation block that you pass to IRXEXEC may be too small to hold the complete result. If so, IRXEXEC places as much of the result that will fit into the evaluation block and sets the length field in the block to the negative of the length required for the complete result. If you call IRXEXEC and the complete result cannot be returned, you can allocate a larger evaluation block, and call the IRXRLT routine and pass the address of the new evaluation block to obtain the complete result. You can also call IRXEXEC and not pass the address of an evaluation block. If the exec returns a result, you can then use the IRXRLT routine to obtain the result.

- A larger evaluation block to return the result from an external function or subroutine that you have written in a programming language other than REXX.

  You can write your own external functions and subroutines. You can write external functions and subroutines in REXX or in any programming language that supports the system-dependent interfaces. If you write your function or subroutine in a programming language other than REXX, when your code is called, it receives the address of an evaluation block that the language processor has allocated. Your code returns the result it calculates in the evaluation block. "Interface for writing external function and subroutine code" on page 264 describes the system interfaces for writing external functions and subroutines and how you use the evaluation block.

  If the evaluation block that your function or subroutine code receives is too small to store the result, you can call the IRXRLT routine to obtain a larger evaluation block. You can then use the new evaluation block to store the result from your function or subroutine.

- An evaluation block that a compiler runtime processor can use to handle the result from a compiled REXX exec.

  A compiler runtime processor can also use IRXRLT to obtain an evaluation block to handle the result from a compiled REXX exec that is currently running. The evaluation block that IRXRLT returns has the same format as the evaluation block for IRXEXEC or for external functions or subroutines. For information about when a compiler runtime processor might require an evaluation block, see *z/OS TSO/E Customization*.

For information about the format of the evaluation block, see "The IRXEXEC routine" on page 249 and "Interface for writing external function and subroutine code" on page 264.

Environment Customization Considerations: If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXRLT to run. On the call to IRXRLT, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the address of the environment block" on page 243.

Entry specifications

For the IRXRLT routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 38 on page 288 describes the parameters for IRXRLT.

Table 38. Parameters for IRXRLT

Parameter	Number of bytes	Description
Parameter 1	8	The function to be performed. The name of the function must be left justified, in uppercase, and padded to the right with blanks. The valid functions are summarized below and are described in "Functions" on page 289. **GETBLOCK** Obtain a larger evaluation block for the external function or subroutine that is running. The GETBLOCK function is valid only when an exec is currently running. **GETRLT** Obtain the result from the last REXX exec that was processed in the current language processor environment. The GETRLT function is valid only if an exec is not currently running. **GETRLTE** Obtain the result from the last REXX exec that was processed in the current language processor environment. The GETRLTE function is the same as GETRLT, except that GETRLTE provides support when REXX execs are nested. **GETEVAL** Obtain an evaluation block to handle the result from a compiled REXX exec. The GETEVAL function is intended for use only by a compiler runtime processor and is valid only when a compiled exec is currently running.
Parameter 2	4	The address of the evaluation block. On input, this parameter is used only for the GETRLT and GETRLTE functions. The parameter is not used for the GETBLOCK and GETEVAL functions. On input, specify the address of an evaluation block that is large enough to hold the result from the exec. On output, this parameter is used only for the GETBLOCK and GETEVAL functions. The parameter is not used for the GETRLT and GETRLTE functions. • On output for the GETBLOCK function, the parameter returns the address of a larger evaluation block that the function or subroutine code can use to return a result. • On output for the GETEVAL function, the parameter returns the address of an evaluation block that the compiler runtime processor can use for the compiled exec that is currently running.
Parameter 3	4	The length, in bytes, of the data area in the evaluation block. This parameter is used on input for the GETBLOCK and GETEVAL functions only. Specify the size needed to store the result from the exec that is currently running. This parameter is not used for the GETRLT and GETRLTE functions.

Table 38. Parameters for IRXRLT (continued)

Parameter	Number of bytes	Description
Parameter 4	4	The address of the environment block that represents the environment in which you want IRXRLT to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, IRXRLT uses the value you specify and ignores register 0. However, IRXRLT does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the address of the environment block" on page 243.
Parameter 5	4	A 4-byte field that IRXRLT uses to return the return code. The return code parameter is optional. If you use this parameter, IRXRLT returns the return code in the parameter and also in register 15. Otherwise, IRXRLT uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 291 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

Functions

Parameter 1 contains the name of the function IRXRLT is to perform. The functions are described below.

GETBLOCK

Use the GETBLOCK function to obtain a larger evaluation block for the external function or subroutine that is running.

You can write external functions and subroutines in REXX or in any programming language that supports the system-dependent interfaces. If you write an external function or subroutine in a programming language other than REXX, when your code is called, it receives the address of an evaluation block. Your code can use the evaluation block to return the result.

For your external function or subroutine code, if the value of the result does not fit into the evaluation block your code receives, you can call IRXRLT to obtain a larger evaluation block. Call IRXRLT with the GETBLOCK function. When you call IRXRLT, specify the length of the data area that you require in parameter 3. IRXRLT allocates a new evaluation block with the specified data area size and returns the address of the new evaluation block in parameter 2. IRXRLT also frees the original evaluation block that was not large enough for the complete result. Your code can then use the new evaluation block to store the result. See "Interface for writing external function and subroutine code" on page 264 for more information about writing external functions and subroutines and the format of the evaluation block.

Note that you can use the GETBLOCK function only when an exec is currently running in the language processor environment.

GETRLT and GETRLTE

You can use either the GETRLT or GETRLTE function to obtain the result from the last REXX exec that was processed in the language processor environment. If you use the IRXEXEC routine to run an exec and then need to invoke IRXRLT to obtain the result from the exec, invoke IRXRLT with the GETRLT or GETRLTE function. You can use the GETRLT function only if an exec is not currently running in the language processor environment. You can use the GETRLTE function regardless of whether an exec is currently running in the environment, which provides support for nested REXX execs.

When you call IRXEXEC, you can allocate an evaluation block and pass the address of the evaluation block to IRXEXEC. IRXEXEC returns the result from the exec in the evaluation block. If the evaluation block is too small, IRXEXEC returns the negative length of the area required for the result. You can allocate another evaluation block that has a data area large enough to store the result and call IRXRLT and pass the address of the new evaluation block in parameter 2. IRXRLT returns the result from the exec in the evaluation block.

You can call IRXEXEC to process an exec that returns a result and not pass the address of an evaluation block on the call. To obtain the result, you can use IRXRLT after IRXEXEC returns. You must allocate an evaluation block and pass the address on the call to IRXRLT.

If you call IRXRLT to obtain the result (GETRLT or GETRLTE function) and the evaluation block you pass to IRXRLT is not large enough to store the result, IRXRLT:

- Places as much of the result that will fit into the evaluation block
- Sets the length of the result field in the evaluation block to the negative of the length required for the complete result.

If IRXRLT cannot return the complete result, the result is not lost. The result is still stored in a system evaluation block. You can then allocate a larger evaluation block and call IRXRLT again specifying the address of the new evaluation block. This is more likely to occur if you had called IRXEXEC without an evaluation block and then use IRXRLT to obtain the result from the exec that executed. It can also occur if you miscalculate the area required to store the complete result.

The result from the exec is available until one of the following occurs:

- You successfully obtain the result using the IRXRLT routine
- Another REXX exec is invoked in the same language processor environment
- The language processor environment is terminated.

Note: The language processor environment is the environment in which REXX execs and routines run. See “General considerations for calling TSO/E REXX routines” on page 240 for information. Chapter 14, “Language processor environments,” on page 311 provides more details about environments and customization services.

You can use the GETRLT function to obtain the result from a REXX exec only if an exec is not currently running in the language processor environment. For example, suppose you use the IRXEXEC routine to run an exec and the result from the exec does not fit into the evaluation block. After IRXEXEC returns control, you can invoke the IRXRLT routine with the GETRLT function to get the result from the exec. At this point, the REXX exec is no longer running in the environment.

You can use the GETRLTE function regardless of whether a REXX exec is currently running in the language processor environment. For example, GETRLTE is useful in the following situation. Suppose you have an exec that calls an external function that is written in assembler. The external function (assembler program) uses the IRXEXEC routine to invoke a REXX exec. However, the result from the invoked exec is too large to be returned to the external function in the evaluation block. The external function can allocate a larger evaluation block and then use IRXRLT with the GETRLTE function to obtain the result from the exec. At this point, the original exec that called the external function is still running in the language processor environment. GETRLTE obtains the result from the last exec that completed in the environment, which, in this case, is the exec the external function invoked.

For more information about running an exec using the IRXEXEC routine and the evaluation block, see “The IRXEXEC routine” on page 249.

GETEVAL

The GETEVAL function is intended for use by a compiler runtime processor. GETEVAL lets a compiler runtime processor obtain an evaluation block whenever it has to handle the result from a compiled REXX exec that is currently running. The GETEVAL function is supported only when a compiled exec is currently running in the language processor environment.

Note that if you write an external function or subroutine in a programming language other than REXX and your function or subroutine code requires a larger evaluation block, you should use the GETBLOCK function, not the GETEVAL function.

Return specifications

For the IRXRLT get result routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

IRXRLT returns a return code in register 15. If you specify the return code parameter (parameter 5), IRXRLT also returns the return code in the parameter.

Table 39 on page 291 shows the return codes if you call IRXRLT with the GETBLOCK function. Additional information about certain return codes is provided after the tables.

Table 39. IRXRLT return codes for the GETBLOCK function

Return code	Description
0	Processing was successful. IRXRLT allocated a new evaluation block and returned the address of the evaluation block.
20	Processing was not successful. A new evaluation block was not allocated.
28	Processing was not successful. A valid language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Table 40 on page 291 shows the return codes if you call IRXRLT with the GETRLT or GETRLTE function.

Table 40. IRXRLT return codes for the GETRLT and GETRLTE functions

Return code	Description
0	Processing was successful. A return code of 0 indicates that IRXRLT completed successfully. However, the complete result may not have been returned.
20	Processing was not successful. IRXRLT could not perform the requested function. The result is not returned.
28	Processing was not successful. A valid language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Table 41 on page 292 shows the return codes if you call IRXRLT with the GETEVAL function.

Table 41. IRXRLT return codes for the GETEVAL function

Return code	Description
0	Processing was successful. IRXRLT allocated an evaluation block and returned the address of the evaluation block.
20	Processing was not successful. An evaluation block was not allocated.
28	Processing was not successful. A valid language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Return Code 0 for the GETRLT and GETRLTE Functions: If you receive a return code of 0 for the GETRLT or GETRLTE function, IRXRLT completed successfully but the complete result may not have been returned. IRXRLT returns a return code of 0 if:

- The entire result was stored in the evaluation block.
- The data field (EVDATA) in the evaluation block was too small. IRXRLT stores as much of the result as it can and sets the length field (EVLEN) in the evaluation block to the negative value of the length that is required.
- No result was available.

Return Code 20: If you receive a return code of 20 for the GETBLOCK, GETRLT, GETRLTE, or GETEVAL function, you may have incorrectly specified the function name in parameter 1.

If you receive a return code of 20 for the GETBLOCK function, some possible errors could be:

- The length you requested (parameter 3) was not valid. Either the length was a negative value or exceeded the maximum value. The maximum is 16 MB minus the length of the evaluation block header.
- The system could not obtain storage.
- You called IRXRLT with the GETBLOCK function and an exec was not running in the language processor environment.

If you receive a return code of 20 for the GETRLT function, some possible errors could be:

- The address of the evaluation block (parameter 2) was 0.
- The evaluation block you allocated was not valid. For example, the EVLEN field was less than 0.

If you receive a return code of 20 for the GETEVAL function, some possible errors could be:

- The length you requested (parameter 3) was not valid. Either the length was a negative value or exceeded the maximum value. The maximum is 16 MB minus the length of the evaluation block header.
- The system could not obtain storage.
- You called IRXRLT with the GETEVAL function and a compiled exec was not currently running in the language processor environment. The GETEVAL function is intended for a compiler runtime processor and can be used only when a compiled REXX exec is currently running.

SAY instruction routine - IRXSAY

The SAY instruction routine, IRXSAY, lets you write a character string to the same output stream as the REXX keyword instruction SAY. For example, in TSO/E foreground, you can write a string to the terminal. "SAY" on page 64 describes the SAY keyword instruction.

A program can access IRXSAY using either the CALL or LINK macro instructions, specifying IRXSAY as the entry point name. You can obtain the address of the IRXSAY routine from the REXX vector of external entry points. "Format of the REXX vector of external entry points" on page 362 describes the vector.

If a program uses IRXSAY, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations: If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXSAY to run. On the call to IRXSAY, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the address of the environment block" on page 243.

Entry specifications

For the IRXSAY routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 42 on page 293 describes the parameters for IRXSAY.

Table 42. Parameters for IRXSAY

Parameter	Number of bytes	Description
Parameter 1	8	The function to be performed. The name of the function must be in uppercase, left justified, and padded to the right with blanks. The valid functions are: • WRITE • WRITEERR "Functions" on page 294 describes the functions in more detail.
Parameter 2	4	The address of an input buffer containing a string. The caller supplies the string, which is a string of bytes that you want IRXSAY to write to the output stream. There are no restrictions on the contents of the string. However, the target device for displaying the data may limit the characters you can specify.

Table 42. Parameters for IRXSAY (continued)

Parameter	Number of bytes	Description
Parameter 3	4	The length, in bytes, of the string that is pointed to by parameter 2.
Parameter 4	4	The address of the environment block that represents the environment in which you want IRXSAY to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, IRXSAY uses the value you specify and ignores register 0. However, IRXSAY does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the address of the environment block" on page 243.
Parameter 5	4	A 4-byte field that IRXSAY uses to return the return code. The return code parameter is optional. If you use this parameter, IRXSAY returns the return code in the parameter and also in register 15. Otherwise, IRXSAY uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 295 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

Functions

Parameter 1 contains the name of the function IRXSAY is to perform. The functions are:

WRITE

Specifies that you want IRXSAY to write the input string you provide to the output stream. In environments that are not integrated into TSO/E, the output is directed to the file specified in the OUTDD field in the module name table. The default OUTDD file is SYSTSPRT.

In environments that are integrated into TSO/E, the output is directed to a terminal (TSO/E foreground) or to SYSTSPRT (TSO/E background).

WRITEERR

Specifies that you want IRXSAY to write the input string you provide to the output stream to which error messages are written.

The settings for the NOMSGWTO and NOMSGIO flags control message processing in a language processor environment. The flags are described in "Flags and corresponding masks" on page 322.

Return specifications

For the IRXSAY routine, the contents of the registers on return are:

Registers 0-14

Same as on entry

Register 15

Return code

Return codes

Table 43 on page 295 shows the return codes for the IRXSAY routine. IRXSAY returns the return code in register 15. If you specify the return code parameter (parameter 5), IRXSAY also returns the return code in the parameter.

Table 43. Return codes for IRXSAY

Return code	Description
0	Processing was successful. The input string was written to the output stream.
8	Processing was successful. However, the input string was not written to the output stream because Halt Typing (HT) is in effect.
20	Processing was not successful. An error occurred and the requested function is not performed. The system may issue a message that describes the error.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Halt condition routine - IRXHLT

The halt condition routine, IRXHLT, lets you query or reset the halt condition. Using IRXHLT, you can determine whether a halt condition has been set, for example, with the HI immediate command. You can also reset the halt condition.

A program can access IRXHLT using either the CALL or LINK macro instructions, specifying IRXHLT as the entry point name. You can obtain the address of the IRXHLT routine from the REXX vector of external entry points. “Format of the REXX vector of external entry points” on page 362 describes the vector.

If a program uses IRXHLT, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations: If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXHLT to run. On the call to IRXHLT, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see “Specifying the address of the environment block” on page 243.

Entry specifications

For the IRXHLT routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 44 on page 296 describes the parameters for IRXHLT.

Table 44. Parameters for IRXHLT

Parameter	Number of bytes	Description
Parameter 1	8	The function to be performed. The name of the function must be left justified, in uppercase, and padded to the right with blanks. The valid functions are: • TESTHLT • CLEARHLT "Functions" on page 296 describes the functions in more detail.
Parameter 2	4	The address of the environment block that represents the environment in which you want IRXHLT to run. This parameter is optional. If you specify an environment block address, IRXHLT uses the value you specify and ignores register 0. However, IRXHLT does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. You can also use register 0 to specify the address of an environment block. If you use register 0, IRXHLT checks whether the address is valid. For more information, see "Specifying the address of the environment block" on page 243.
Parameter 3	4	A 4-byte field that IRXHLT uses to return the return code. The return code parameter is optional. If you use this parameter, IRXHLT returns the return code in the parameter and also in register 15. Otherwise, IRXHLT uses register 15 only. "Return codes" on page 297 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

Functions

Parameter 1 contains the name of the function IRXHLT is to perform. The functions are:

TESTHLT

Determines whether the halt condition has been set. For example, the halt condition may be set by the HI immediate command, the EXECUTIL HI command, or the trace and execution control routine, IRXIC.

Return codes 0 and 4 from IRXHLT indicate whether the halt condition has been set. See "Return codes" on page 297 for more information.

CLEARHLT

Resets the halt condition.

Return specifications

For the IRXHLT routine, the contents of the registers on return are:

Registers 0-14

Same as on entry

Register 15

Return code

Return codes

Table 45 on page 297 shows the return codes for the IRXHLT routine. IRXHLT returns the return code in register 15. If you specify the return code parameter (parameter 3), IRXHLT also returns the return code in the parameter.

Table 45. Return codes for IRXHLT

Return code	Description
0	Processing was successful. For the TESTHLT function, a return code of 0 indicates the halt condition was tested and the condition has not been set. This means that REXX exec processing will continue. For the CLEARHLT function, a return code of 0 indicates the halt condition was successfully reset.
4	Processing was successful. A return code of 4 is used only for the TESTHLT function. The return code indicates the halt condition was tested and the condition has been set. This means that REXX processing will be halted, for example, just as if EXECUTIL HI were processed.
20	Processing was not successful. An error occurred and the requested function is not performed. IRXHLT returns a return code of 20 if the function you specify in parameter 1 is invalid.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Text retrieval routine - IRXTXT

The text retrieval routine, IRXTXT, lets you retrieve the same text the TSO/E REXX interpreter uses for several options of the DATE built-in function and for the ERRORTEXT built-in function. Using IRXTXT, you can retrieve the:

- English names for the days of the week, in mixed case (for example, Thursday)
- English names for the months of the year, in mixed case (for example, August)

- Abbreviated English names for the months of the year, in mixed case (for example, Aug)
- Text of a REXX syntax error message. For example, for error number 26 (message IRX0026I), the message text is:

```
Invalid whole number
```

A program can access IRXTXT using either the CALL or LINK macro instructions, specifying IRXTXT as the entry point name. You can obtain the address of the IRXTXT routine from the REXX vector of external entry points. "Format of the REXX vector of external entry points" on page 362 describes the vector.

If a program uses IRXTXT, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations: If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXTXT to run. On the call to IRXTXT, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see "Specifying the address of the environment block" on page 243.

Entry specifications

For the IRXTXT routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 46 on page 299 describes the parameters for IRXTXT.

Table 46. Parameters for IRXTXT

Parameter	Number of bytes	Description
Parameter 1	8	The function to be performed. The name of the function must be left justified, in uppercase, and padded to the right with blanks. The valid functions are: • DAY • MTHLONG • MTHSHORT • SYNTXMSG "Functions and text units" on page 300 describes the functions in more detail.
Parameter 2	4	A fullword binary field that contains the text unit corresponding to the function in parameter 1. The text unit you specify depends on the function you use in parameter 1 and the corresponding value you want IRXTXT to return. "Functions and text units" on page 300 describes the text units in more detail.
Parameter 3	4	The address of an area in storage to hold the text that IRXTXT retrieves.
Parameter 4	4	The length of the area in storage that is pointed to by parameter 3. It is suggested that you provide a large buffer area to hold the result, for example, 250 bytes. If the buffer is too small to hold the returned text, IRXTXT returns with return code 20. On output, IRXTXT updates parameter 4 to contain the length of the actual text it returns.
Parameter 5	4	The address of the environment block that represents the environment in which you want IRXTXT to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, IRXTXT uses the value you specify and ignores register 0. However, IRXTXT does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Specifying the address of the environment block" on page 243.
Parameter 6	4	A 4-byte field that IRXTXT uses to return the return code. The return code parameter is optional. If you use this parameter, IRXTXT returns the return code in the parameter and also in register 15. Otherwise, IRXTXT uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 302 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

Functions and text units

Parameter 1 contains the name of the function IRXTXT is to perform. Parameter 2 specifies the text unit you want IRXTXT to retrieve for the particular function. The functions and their corresponding text units you can request are described below:

DAY

The DAY function returns the English name of a day of the week, in mixed case. The names that IRXTXT retrieves are the same values the TSO/E REXX interpreter uses for the DATE(Weekday) function.

The name of the day that IRXTXT retrieves depends on the text unit you specify in parameter 2. Table 47 on page 300 shows the text units for parameter 2 and the corresponding day IRXTXT retrieves for each text unit. For example, if you want IRXTXT to return the value *Saturday*, you would specify text unit 3.

Table 47. Text unit and day returned - DAY function

Text unit	Name of day returned
1	Thursday
2	Friday
3	Saturday
4	Sunday
5	Monday
6	Tuesday
7	Wednesday

MTHLONG

The MTHLONG function returns the English name of a month, in mixed case. The names that IRXTXT retrieves are the same values the TSO/E REXX interpreter uses for the DATE(Month) function.

The name of the month that IRXTXT retrieves depends on the text unit you specify in parameter 2. Table 48 on page 300 shows the text units for parameter 2 and the corresponding name of the month IRXTXT retrieves for each text unit. For example, if you wanted IRXTXT to return the value *April*, you would specify text unit 4.

Table 48. Text unit and month returned - MTHLONG function

Text unit	Name of month returned
1	January
2	February
3	March
4	April
5	May
6	June
7	July
8	August
9	September
10	October

Table 48. Text unit and month returned - MTHLONG function (continued)

Text unit	Name of month returned
11	November
12	December

MTHSHORT

The MTHSHORT function returns the first three characters of the English name of a month, in mixed case. The names that IRXTXT retrieves are the same values the TSO/E REXX interpreter uses for the month in the DATE(Normal) function.

The abbreviated name of the month that IRXTXT retrieves depends on the text unit you specify in parameter 2. Table 49 on page 301 shows the text units for parameter 2 and the corresponding abbreviated names of the month that IRXTXT retrieves for each text unit. For example, if you wanted IRXTXT to return the value *Sep*, you would specify text unit 9.

Table 49. Text unit and abbreviated month returned - MTHSHORT function

Text unit	Abbreviated name of month returned
1	Jan
2	Feb
3	Mar
4	Apr
5	May
6	Jun
7	Jul
8	Aug
9	Sep
10	Oct
11	Nov
12	Dec

SYNTXMSG

The SYNTXMSG function returns the message text for a specific REXX syntax error message. The text that IRXTXT retrieves is the same text the ERRORTEXT function returns.

The message text that IRXTXT retrieves depends on the text unit you specify in parameter 2. For the text unit, specify the error number corresponding to the error message. For example, error number 26 corresponds to message IRX0026I. The message text for IRX0026I is:

```
Invalid whole number
```

This is the value the SYNTXMSG function returns if you specify text unit 26.

The values 1-99 are reserved for error numbers. However, not all of the values are used for REXX syntax error messages. If you specify a text unit in the range 1-99 and the value is not supported, IRXTXT returns a string of length 0.

Return specifications

For the IRXTXT routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

Table 50 on page 302 shows the return codes for the IRXTXT routine. IRXTXT returns the return code in register 15. If you specify the return code parameter (parameter 6), IRXTXT also returns the return code in the parameter.

Table 50. Return codes for IRXTXT

Return code	Description
0	Processing was successful. IRXTXT retrieved the text you requested and placed the text into the buffer area.
20	Processing was not successful. An error occurred and the requested function is not performed. IRXTXT does not retrieve the text. You may receive a return code of 20 if the: • Buffer is too small to hold the complete text • Function you specified for parameter 1 is invalid • Text unit you specified for parameter 2 is invalid for the particular function you requested in parameter 1.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

LINESIZE function routine - IRXLIN

The LINESIZE function routine, IRXLIN, lets you obtain the same value that the LINESIZE built-in function returns. “LINESIZE” on page 94 describes the built-in function.

A program can access IRXLIN using either the CALL or LINK macro instructions, specifying IRXLIN as the entry point name. You can obtain the address of the IRXLIN routine from the REXX vector of external entry points. “Format of the REXX vector of external entry points” on page 362 describes the vector.

If a program uses IRXLIN, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations: If you use the IRXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want IRXLIN to run. On the call to IRXLIN, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

For more information about specifying environments and how routines determine the environment in which to run, see “Specifying the address of the environment block” on page 243.

Entry specifications

For the IRXLIN routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 51 on page 303 describes the parameters for IRXLIN.

Table 51. Parameters for IRXLIN

Parameter	Number of bytes	Description
Parameter 1	8	The function to be performed. The function name must be left justified, in uppercase, and padded to the right with blanks. The only valid function is LINESIZE, which returns the same value that the LINESIZE built-in function returns.
Parameter 2	4	IRXLIN returns the LINESIZE value in this parameter. IRXLIN returns the same value that the LINESIZE built-in function returns. "LINESIZE" on page 94 describes the built-in function. The value IRXLIN returns in this parameter is valid only if the return code is 0.
Parameter 3	4	The address of the environment block that represents the environment in which you want IRXLIN to run. This parameter is optional. If you specify an environment block address, IRXLIN uses the value you specify and ignores register 0. However, IRXLIN does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. You can also use register 0 to specify the address of an environment block. If you use register 0, IRXLIN checks whether the address is valid. For more information, see "Specifying the address of the environment block" on page 243.

Table 51. Parameters for IRXLIN (continued)

Parameter	Number of bytes	Description
Parameter 4	4	A 4-byte field that IRXLIN uses to return the return code. The return code parameter is optional. If you use this parameter, IRXLIN returns the return code in the parameter and also in register 15. Otherwise, IRXLIN uses register 15 only. "Return codes" on page 304 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

Return specifications

For the IRXLIN routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

Table 52 on page 304 shows the return codes for the IRXLIN routine. IRXLIN returns the return code in register 15. If you specify the return code parameter (parameter 4), IRXLIN also returns the return code in the parameter.

Table 52. Return codes for IRXLIN

Return code	Description
0	Processing was successful. IRXLIN returned the LINESIZE value in parameter 2.
20	Processing was not successful. You may have specified a not valid function (parameter 1). The only valid function is LINESIZE.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Chapter 13. TSO/E REXX customizing services

In addition to the instructions, functions, and commands for writing a REXX exec and the programming services that interface with REXX and the language processor, TSO/E also provides customizing services for REXX processing. The customizing services let you change how REXX execs are processed and how system services are accessed and used.

The REXX language itself, which consists of instructions and built-in functions, is address space independent. The language processor environment, which interprets a REXX exec, processes the REXX language instructions and functions in the same manner in any address space. However, when a REXX exec executes, the language processor environments must interface with different host services, such as I/O and storage. MVS address spaces differ in how they access and use system services, for example, how they use and manage I/O and storage. Although these differences exist, the language processor environments must run in an environment that is not dependent on the address space in which it is executing an exec. The environment must allow REXX execs to execute independently of the way in which an address space handles system services. The TSO/E REXX customizing routines and services provide an interface between the language processor and underlying host services and allow you to customize the environment in which the language processor processes REXX execs.

TSO/E REXX customizing services include the following:

Environment Characteristics
TSO/E provides various routines and services that allow you to customize the environment in which the language processor executes a REXX exec. This environment is known as the *language processor environment* and defines various characteristics relating to how execs are processed and how system services are accessed and used. TSO/E provides default environment characteristics that you can change and also provides a routine you can use to define your own environment.

Replaceable Routines
When a REXX exec executes, various system services are used, such as services for loading and freeing an exec, performing I/O, obtaining and freeing storage, and handling data stack requests. TSO/E provides routines that handle these types of system services. The routines are known as *replaceable routines* because you can provide your own routine that replaces the system routine.

Exit Routines
You can provide exit routines to customize various aspects of REXX processing.

The topics in this chapter introduce the major interfaces and customizing services. The following chapters describe the customizing services in more detail:

- Chapter 14, “Language processor environments,” on page 311 describes how you can customize the environment in which the language processor executes a REXX exec and accesses and uses system services.
- Chapter 15, “Initialization and termination routines,” on page 371 describes the IRXINIT and IRXTERM routines that TSO/E provides to initialize and terminate language processor environments.
- Chapter 16, “Replaceable routines and exits,” on page 389 describes the routines you can provide that access system services, such as I/O and storage, and the exits you can use to customize REXX processing.

Flow of REXX exec processing

Figure 19 on page 306 shows the processing of a REXX exec in any MVS address space.

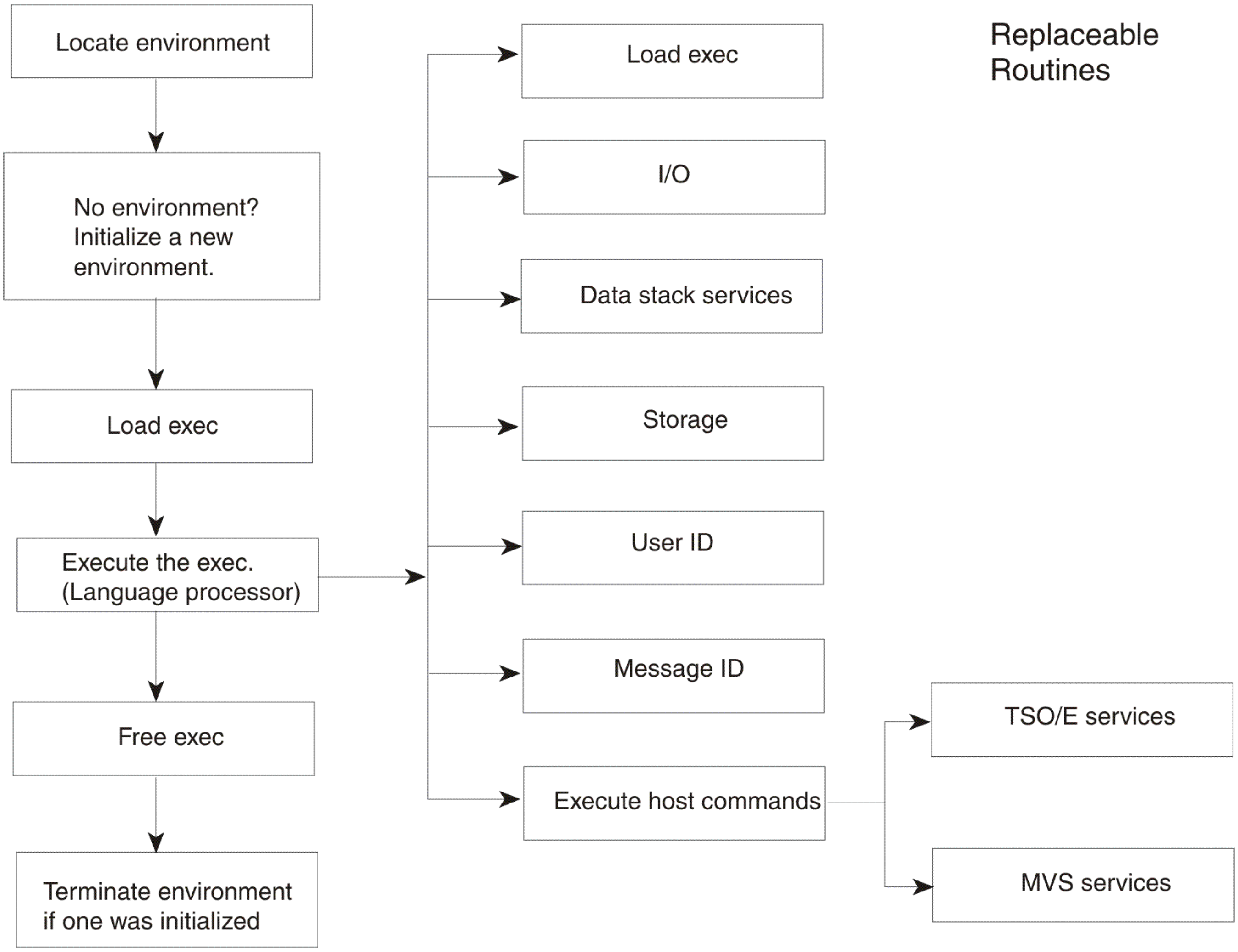


Figure 19. Overview of REXX Exec processing in any address space

As shown in the figure, before the language processor executes a REXX exec, a language processor environment must exist. After an environment is located or initialized, the exec is loaded into storage and is then executed. While an exec is executing, the language processor may need to access different system services, for example, to handle data stack requests or for I/O processing. The system services are handled by routines that are known as replaceable routines. The following topics describe the initialization and termination of language processor environments, the loading and freeing of an exec, and the replaceable routines. In addition, there are several exits you can provide to customize REXX processing. See "REXX exit routines" on page 426 where the exits are summarized.

Initialization and termination of a language processor environment

Before the language processor environment can process a REXX exec, a *language processor environment* must exist. A language processor environment is the environment in which the language processor "interprets" or processes the exec. This environment defines characteristics relating to how the exec is processed and how the language processor accesses system services.

A language processor environment defines various characteristics, such as:

- The search order used to locate commands and external functions and subroutines
- The ddnames for reading and writing data and from which REXX execs are loaded
- The host command environments you can use in an exec to execute host commands (that is, the environments you can specify using the ADDRESS instruction)

- The function packages (user, local, and system) that are available to execs that execute in the environment and the entries in each package
- Whether execs that execute in the environment can use the data stack or can perform I/O operations
- The names of routines that handle system services, such as I/O operations, loading of an exec, obtaining and freeing storage, and data stack requests. These routines are known as replaceable routines.

The concept of a language processor environment is different from that of a host command environment. The language processor environment is the environment in which a REXX exec executes. This includes how an exec is loaded, how commands, functions, and subroutines are located, and how requests for system services are handled. A host command environment is the environment to which the language processor passes commands for execution. The host command environment handles the execution of host commands. The host command environments that are available to a REXX exec are one characteristic of a language processor environment. For more information about executing host commands from a REXX exec, see "Commands to external environments" on page 22.

TSO/E automatically initializes a language processor environment in both the TSO/E and non-TSO/E address spaces by calling the *initialization routine* IRXINIT. TSO/E terminates a language processor environment by calling the *termination routine* IRXTERM.

In the TSO/E address space, IRXINIT is called to initialize a default language processor environment when a user logs on and starts a TSO/E session. When a user invokes ISPF, another language processor environment is initialized. The ISPF environment is a separate environment from the one that is initialized when the TSO/E session is started. Similarly, if you enter split screen mode in ISPF, another language processor environment is initialized for the second ISPF screen. Therefore, three separate language processor environments exist. If the user invokes a REXX exec from the second ISPF screen, the exec executes within the language processor environment that was initialized for that second screen. If the user invokes the exec from TSO/E READY mode, it executes within the environment that was initialized when the user first logged on.

When the user returns to a single ISPF screen, the IRXTERM routine is called to automatically terminate the language processor environment that is associated with the second ISPF screen. Similarly, when the user exits from ISPF and returns to TSO/E READY mode, the system calls IRXTERM to terminate the environment associated with the ISPF screen. When the user logs off from TSO/E, that language processor environment is then terminated.

In non-TSO/E address spaces, a language processor environment is not automatically initialized at a specific point, such as when the address space is activated. An environment is initialized when either the IRXEXEC or IRXJCL routines are called to execute a REXX exec, if an environment does not already exist.

As described above, many language processor environments can exist in an address space. A language processor environment is associated with an MVS task and environments can be chained together. This is discussed in more detail in Chapter 14, "Language processor environments," on page 311.

Whenever a REXX exec is invoked in any address space, the system first determines whether a language processor environment exists. If an environment does exist, the REXX exec executes in that environment. If an environment does not exist, the system automatically initializes one by calling the IRXINIT routine. For example, if you are logged on to TSO/E and issue the TSO/E EXEC command to execute a REXX exec, the system checks whether a language processor environment exists. An environment was initialized when you logged on to TSO/E, therefore, the exec executes in that environment. If you execute a REXX exec in MVS batch by specifying IRXJCL as the program name (PGM=) on the JCL EXEC statement, a language processor environment is initialized for the execution of the exec. When the exec completes processing, the environment is terminated.

If either IRXJCL or IRXEXEC is called from a program, the system first determines whether a language processor environment already exists. If an environment exists, the exec executes in that environment. If an environment does not exist, an environment is initialized. When the exec completes, the environment is terminated. "Chains of environments and how environments are located" on page 340 describes how the system locates a previous environment in the TSO/E and non-TSO/E address spaces.

TSO/E provides default values that are used to define a language processor environment. The defaults are provided in three *parameters modules* that are load modules. The load modules contain the default characteristics for initializing language processor environments for TSO/E (READY mode), ISPF, and non-TSO/E address spaces. The parameters modules are:

- IRXTSPRM (for TSO/E)
- IRXISPRM (for ISPF)
- IRXPARMS (for non-TSO/E)

You can provide your own parameters modules to change the default values that are used to initialize a language processor environment. Your load modules are then used instead of the default modules provided by TSO/E. The parameters modules are described in detail in Chapter 14, "Language processor environments," on page 311.

You can also explicitly invoke IRXINIT to initialize a language processor environment and define the environment characteristics on the call. Although IRXINIT is primarily intended for use in non-TSO/E address spaces, you can call it in any address space. When you call IRXINIT, you specify any or all of the characteristics you want defined for the language processor environment. Using IRXINIT gives you the flexibility to define your own environment, and therefore, *customize* how REXX execs execute within the environment and how system services are handled. If you explicitly call IRXINIT, you must use the IRXTERM routine to terminate that environment. The system does not automatically terminate an environment that you initialized by explicitly calling IRXINIT. Chapter 15, "Initialization and termination routines," on page 371 describes the IRXINIT and IRXTERM routines.

Types of language processor environments

There are two types of language processor environments; environments that are integrated into TSO/E and environments that are not integrated into TSO/E. If an environment is integrated into TSO/E, REXX execs that run in the environment can use TSO/E commands and services. If an environment is not integrated into TSO/E, execs that run in the environment cannot use TSO/E commands and services.

When a language processor environment is automatically initialized in the TSO/E address space, the environment is integrated into TSO/E. When an environment is automatically initialized in a non-TSO/E address space, the environment is not integrated into TSO/E. Environments that are initialized in non-TSO/E address spaces cannot be integrated into TSO/E. Environments that are initialized in the TSO/E address space may or may not be integrated into TSO/E.

Many TSO/E customizing routines and services are only available to language processor environments that are *not* integrated into TSO/E. "Types of environments - integrated and not integrated into TSO/E" on page 316 describes the types of language processor environments in more detail.

Loading and freeing a REXX exec

After a language processor environment has been located or one has been initialized, the exec must be loaded into storage in order for the language processor to process it. After the exec executes, it must be freed. The exec load routine loads and frees REXX execs. The default exec load routine is IRXLOAD.

The exec load routine is one of the replaceable routines that you can provide to customize REXX processing. You can provide your own exec load routine that either replaces the system default or that performs pre-processing and then calls the default routine IRXLOAD. The name of the load routine is defined for each language processor environment. You can only provide your own load routine in language processor environments that are not integrated into TSO/E.

Note: If you use the IRXEXEC routine to execute a REXX exec, you can preload the exec in storage and pass the address of the preloaded exec on the call to IRXEXEC. In this case, the exec load routine is not called to load the exec. "Exec processing routines - IRXJCL and IRXEXEC" on page 245 describes the IRXEXEC routine and how you can preload an exec.

Processing of the REXX exec

After the REXX exec is loaded into storage, the language processor is called to process (interpret) the exec. During processing, the exec can issue commands, call external functions and subroutines, and request various system services. When the language processor processes a command, it first evaluates the expression and then passes the command to the host for execution. The specific host command environment handles command execution. When the exec calls an external function or subroutine, the language processor searches for the function or subroutine. This includes searching any function packages that are defined for the language processor environment in which the exec is executing.

When system services are requested, specific routines are called to perform the requested service (for example, obtaining and freeing storage, I/O, and data stack requests). TSO/E provides routines for these services that are known as replaceable routines because you can provide your own routine that replaces the system routine. "Overview of replaceable routines" on page 309 summarizes the routines.

Overview of replaceable routines

When a REXX exec executes, various system services are used, such as services for loading and freeing the exec, I/O, obtaining and freeing storage, and handling data stack requests. TSO/E provides routines that handle these types of system services. These routines are known as *replaceable routines* because you can provide your own routine that replaces the system routine. You can only provide your own replaceable routines in language processor environments that are not integrated into TSO/E (see "Types of environments - integrated and not integrated into TSO/E" on page 316).

Your routine can check the request for a system service, change the request if needed, and then call the system-supplied routine to actually perform the service. Your routine can also terminate the request for a system service or perform the request itself instead of calling the system-supplied routine.

Replaceable routines are defined on a language processor environment basis and are specified in the parameters module for an environment (see page "Characteristics of a Language Processor Environment" on page 317).

Table 53 on page 309 provides a brief description of the functions your replaceable routine must perform. Chapter 16, "Replaceable routines and exits," on page 389 describes each replaceable routine in detail, its input and output parameters, and return codes.

Table 53. Overview of replaceable routines

Replaceable routine	**Description**
Exec load	The exec load routine is called to load a REXX exec into storage and to free the exec when it is no longer needed.
Read input and write output (I/O)	The I/O routine is called to read a record from or write a record to a specified ddname. For example, this routine is called for the SAY instruction, for the PULL instruction (when the data stack is empty), and for the EXECIO command. The routine is also called to open and close a data set.
Data stack	This routine is called to handle any requests for data stack services. For example, it is called for the PULL, PUSH, and QUEUE instructions and for the MAKEBUF and DROPBUF commands.
Storage management	This routine is called to obtain and free storage.
User ID	This routine is called to obtain the user ID. The result that it obtains is returned by the USERID built-in function.
Message identifier	This routine determines if the message identifier (message ID) is displayed with a REXX error message.
Host command environment	This routine is called to handle the execution of a host command for a particular host command environment.

To provide your own replaceable routine, you must do the following:

- Write the code for the routine. Chapter 16, “Replaceable routines and exits,” on page 389 describes each routine in detail.
- Define the routine name to a language processor environment.

  If you use IRXINIT to initialize a new environment, you can pass the names of your routines on the call.

Chapter 14, “Language processor environments,” on page 311 describes the concepts of replaceable routines and their relationship to language processor environments in more detail.

The replaceable routines that TSO/E provides are external interfaces that you can call from a program in any address space. For example, a program can call the system-supplied data stack routine to perform data stack operations. If you provide your own replaceable data stack routine, a program can call your routine to perform data stack operations. You can call a system-supplied or user-supplied replaceable routine only if a language processor environment exists in which the routine can execute.

Exit routines

TSO/E also provides several exit routines you can use to customize REXX processing. Several exits have fixed names. Other exits do not have a fixed name. You supply the name of these exits on the call to IRXINIT or by changing the appropriate default parameters modules that TSO/E provides. Chapter 16, “Replaceable routines and exits,” on page 389 describes the exits in more detail. A summary of each exit follows.

- IRXINITX -- Pre-environment initialization exit routine. The exit receives control whenever IRXINIT is called to initialize a new language processor environment. It gets control before IRXINIT evaluates any parameters.
- IRXITTS or IRXITMV -- Post-environment initialization exit routines. IRXITTS is for environments that are integrated into TSO/E and IRXITMV is for environments that are not integrated into TSO/E. The IRXITTS or IRXITMV exit receives control whenever IRXINIT is called to initialize a new language processor environment. It receives control after IRXINIT initializes a new environment but before IRXINIT completes.
- IRXTERMX -- Environment termination exit routine. The exit receives control whenever IRXTERM is called to terminate a language processor environment. It gets control before IRXTERM starts termination processing.
- Attention handling exit routine -- The exit receives control whenever a REXX exec is executing in the TSO/E address space (in a language processor environment that is integrated into TSO/E) and an attention interruption occurs.
- Exec initialization -- The exit receives control after the variable pool for a REXX exec has been initialized but before the language processor processes the first clause in the exec.
- Exec termination -- The exit receives control after a REXX exec has completed processing but before the variable pool has been terminated.
- Exit for the IRXEXEC routine (exec processing exit) -- The exit receives control whenever the IRXEXEC routine is called to execute a REXX exec. The IRXEXEC routine can be explicitly called by a user or called by the system to execute an exec. IRXEXEC is always called by the system to handle exec execution. For example, if you use IRXJCL to execute an exec in MVS batch, IRXEXEC is called to execute the exec. If you provide an exit for IRXEXEC, the exit is invoked.

The exit routines for REXX processing are different from the replaceable routines that are described in the previous topic. You can provide replaceable routines only in language processor environments that are not integrated into TSO/E. Except for the attention handling exit, you can provide exits in any type of language processor environment (integrated and not integrated into TSO/E). Note that for post-environment initialization, you use IRXITTS for environments that are integrated into TSO/E and IRXITMV for environments that are not integrated into TSO/E.

You can use the attention handling exit only in an environment that is integrated into TSO/E.

Chapter 14. Language processor environments

As described in Chapter 13, “TSO/E REXX customizing services,” on page 305, a language processor environment is the environment in which the language processor "interprets" or processes a REXX exec. Such an environment must exist before an exec can run.

The topics in this chapter explain language processor environments and the default parameters modules in more detail. They explain the various tasks that you can perform to customize the environment in which REXX execs run. This chapter describes:

- Different aspects of a language processor environment and the characteristics that make up such an environment. This chapter explains when the system invokes the initialization routine, IRXINIT, to initialize an environment and the values IRXINIT uses to define the environment. This chapter describes the values TSO/E provides in the default parameters modules and how to change the values. It also describes what values you can and cannot specify when the environment is integrated into TSO/E (TSOFL flag is on) and when the environment is not integrated into TSO/E (TSOFL flag is off).
- The various control blocks that are defined when a language processor environment is initialized and how you can use the control blocks for REXX processing.
- How language processor environments are chained together.
- How the data stack is used in different language processor environments.

Rule: The control blocks created for a language processor environment provide information about the environment. You can obtain information from the control blocks. However, you *must not change* any of the control blocks. If you do, unpredictable results may occur.

Overview of Language Processor Environments

The language processor environment defines various characteristics that relate to how execs are processed and how system services are accessed and used. Some of the environment characteristics include the following:

- The language in which the system displays REXX messages
- The ddnames from which input is read, to which output is written, and from which REXX execs are fetched
- The names of several *replaceable routines* that you can provide for system services, such as I/O processing, loading REXX execs, and processing data stack requests
- The names of exit routines that the system invokes at different points in REXX processing, such as when the IRXEXEC routine is invoked or when a user enters attention mode in TSO/E
- The names of host command environments and the corresponding routines that process commands for each host command environment
- The function packages that are available to execs that run in the environment
- The subpool the system uses for storage allocation
- The name of the address space
- Bit settings (flags) that define many characteristics, such as:
  - Whether the environment is integrated into TSO/E (that is, whether execs running in the environment can use TSO/E commands and services)
  - The search order for commands and for functions and subroutines
  - Whether the system displays primary and alternate messages

“Characteristics of a Language Processor Environment” on page 317 describes the environment characteristics.

The REXX language itself is address space independent. For example, if an exec includes a DO loop, the language processor processes the DO loop in the same manner regardless of whether the exec runs in TSO/E or in a non-TSO/E address space. However, when the language processor processes a REXX exec, various host services are used, such as I/O and storage. MVS address spaces differ in how they access and use system services, such as I/O and storage management. Although these differences exist, the REXX exec must run in an environment that is not dependent on the particular address space in which the exec was invoked. Therefore, a REXX exec runs in a language processor environments which is an environment that can be *customized* to support how each address space accesses and uses host services.

When a language processor environment is initialized, different routines can be defined that the system invokes for system services, such as obtaining and freeing storage and handling I/O requests. The language processor environment provides for consistency across MVS address spaces by ensuring that REXX execs run independently of the way in which the system accesses system services. At the same time, the language processor environment provides flexibility to handle the differences between the address spaces and also lets you customize how REXX execs are processed and how the system accesses and uses system services.

Initialization of an Environment: The initialization routine, IRXINIT, initializes language processor environments. The system calls IRXINIT in both TSO/E and non-TSO/E address spaces to automatically initialize an environment. Because the system automatically initializes language processor environments, users need not be concerned with setting up such an environment, changing any values, or even that the environment exists. The language processor environment allows application programmers and system programmers to customize the system interfaces between the language processor and host services. "When environments are automatically initialized in TSO/E" on page 314 describes when the system automatically initializes an environment in the TSO/E address space. "When environments are automatically initialized in MVS" on page 315 describes when the system initializes environments in non-TSO/E address spaces.

When the system calls IRXINIT to automatically initialize an environment, the system uses default values. TSO/E provides three default parameters modules (load modules) that contain the parameter values IRXINIT uses to initialize three different types of language processor environments. The three default parameters modules are:

- IRXTSPRM (for a TSO/E session)
- IRXISPRM (for ISPF)
- IRXPARMS (for non-TSO/E address spaces)

"Characteristics of a Language Processor Environment" on page 317 describes the parameters module that contains all of the characteristics for defining a language processor environment. "Values provided in the three default parameters modules" on page 336 describes the defaults TSO/E provides in the three parameters modules. You can change the default parameters that TSO/E provides by providing your own load modules. "Changing the default values for initializing an environment" on page 344 describes how to change the parameters.

You can also explicitly invoke IRXINIT and pass the parameter values for IRXINIT to use to initialize the environment. Using IRXINIT gives you the flexibility to customize the environment in which REXX execs run and how the system accesses and uses system services.

Chains of Environments: Many language processor environments can exist in a particular address space. A language processor environment is associated with an MVS task. There can be multiple environments associated with one task. language processor environments are chained together in a hierarchical structure and form a *chain of environments* where each environment on a chain is related to the other environments on that chain. Although many environments can be associated with one MVS task, each individual language processor environment is associated with one and only one MVS task. Environments on a particular chain may share various resources, such as data sets and the data stack. "Chains of environments and how environments are located" on page 340 describes the relationship between language processor environments and MVS tasks and how environments are chained together.

Maximum Number of Environments: Although there can be many language processor environments initialized in a single address space, there is a default maximum. The load module IRXANCHR contains an environment table that defines the maximum number of environments for one address space. The default maximum is not a specific number of environments. The maximum number of environments depends on the number of chains of environments and the number of environments defined on each chain. The default maximum should be sufficient for any address space. However, if a new environment is being initialized and the maximum has already been used, IRXINIT completes unsuccessfully and returns with a return code of 20 and a reason code of 24. If this error occurs, you can change the maximum value by providing a new IRXANCHR load module. "Changing the maximum number of environments in an address space" on page 365 describes the IRXANCHR load module and how to provide a new module.

Control Blocks: When IRXINIT initializes a new language processor environments IRXINIT creates a number of control blocks that contain information about the environment. The main control block that IRXINIT creates is called the *environment block* (ENVBLOCK). Each language processor environment is represented by its environment block. The environment block contains pointers to other control blocks that contain information about the parameters that define the environment, the resources within the environment, and the exec currently running in the environment. "Control blocks created for a Language Processor Environment" on page 357 describes all of the control blocks that IRXINIT creates. IRXINIT creates an environment block for each language processor environment that it creates. Except for the initialization routine, IRXINIT, all REXX execs and services cannot operate without an environment being available.

Rule About Changing Any Control Blocks: You can obtain information from the control blocks. However, you *must not change* any of the control blocks. If you do, unpredictable results may occur.

Using the environment block

The main control block that IRXINIT creates for a language processor environment is the environment block. The environment block represents the language processor environment and points to other control blocks that contain information about the environment.

The environment block is known as the *anchor* that all callable interfaces to REXX use. All REXX routines, except for the IRXINIT initialization routine, cannot run unless an environment block exists, that is, a language processor environment must exist. When IRXINIT initializes a new language processor environment, IRXINIT always returns the address of the environment block in register 0. (If you explicitly invoke the IRXINIT routine, IRXINIT also returns the address of the environment block in the parameter list.) You can also use IRXINIT to obtain the address of the environment block for the current non-reentrant environment (see "Initialization routine - IRXINIT" on page 371). IRXINIT returns the address in register 0 and also in a parameter in the parameter list.

The address of the environment block is useful for calling a REXX routine or for obtaining information from the control blocks that IRXINIT created for the environment. If you invoke any of the TSO/E REXX routines (for example, IRXEXEC to process an exec or the variable access routine IRXEXCOM), you can optionally pass the address of an environment block to the routine in register 0. By passing the address of an environment block, you can specify in which specific environment you want either the exec or the service to run. This is particularly useful if you use the IRXINIT routine to initialize several environments on a chain and then want to process a TSO/E REXX routine in a specific environment. When you invoke the routine, you can pass the address of the environment block in register 0.

An external function or subroutine receives the address of an environment block in register 0. This environment block address should be passed on all calls to any TSO/E REXX programming service. Passing the environment block address is particularly important when the environment is a reentrant environment because TSO/E REXX programming services cannot automatically locate a reentrant environment. For more information about reentrant environments, see "Using the environment block for reentrant environments" on page 244.

If you invoke a TSO/E REXX routine and do not pass the address of an environment block in register 0 or the environment block parameter, the routine runs:

- In the current non-reentrant environment on the chain under the current task (environment not integrated into TSO/E)
- In the last environment on the chain under the current task or a parent task (environment integrated into TSO/E)

For more information, see "Types of environments - integrated and not integrated into TSO/E" on page 316.

If you invoke the IRXEXEC or IRXJCL routine and a language processor environment does not exist, the system calls IRXINIT to initialize an environment in which the exec will run. When the exec completes processing, the system terminates the newly created environment.

If you are running separate tasks simultaneously and two or more tasks are running REXX, each task must have its own environment block. That is, you must initialize a language processor environment for each of the tasks.

The environment block points to several other control blocks that contain the parameters IRXINIT used in defining the environment and the addresses of TSO/E REXX routines, such as IRXINIT, IRXEXEC, and IRXTERM, and replaceable routines. You can access these control blocks to obtain this information. The control blocks are described in "Control blocks created for a Language Processor Environment" on page 357.

Rule About Changing Any Control Blocks: You can obtain information from the control blocks. However, you *must not change* any of the control blocks. If you do, unpredictable results may occur.

When environments are automatically initialized in TSO/E

The initialization routine, IRXINIT, initializes a language processor environment. The system calls IRXINIT to automatically initialize a default environment when a user logs on to TSO/E and when a user invokes ISPF.

When a user logs on TSO/E, the system calls IRXINIT as part of the logon process to automatically initialize a language processor environment for the TSO/E session. The initialization of a language processor environment is transparent to the user. After users log on to TSO/E, they can simply invoke a REXX exec without performing any other tasks.

Note: If your installation uses a user-written terminal monitor program (TMP) instead of the TMP provided by TSO/E, the system does not automatically initialize a language processor environment. See "Initializing environments for user-written TMPs" on page 315 for information about the tasks you must perform to initialize a language processor environment to run REXX execs.

Similarly, when a user invokes ISPF from TSO/E, the system calls the IRXINIT routine to automatically initialize a language processor environment for ISPF, that is, for the ISPF screen. The second language processor environment is separate from the environment that IRXINIT initialized for the TSO/E session. If the user enters split screen in ISPF, IRXINIT initializes a third language processor environment for the second ISPF screen. At this point, three separate language processor environments exist. If the user invokes a REXX exec from the second ISPF screen, the exec runs under the third language processor environment, that is, the environment IRXINIT initialized for the second ISPF screen. If the user invokes the exec from the first ISPF screen, the exec runs under the second language processor environment.

The termination routine, IRXTERM, terminates a language processor environment. Continuing the above example, when the user returns to one screen in ISPF, the system calls the IRXTERM routine. IRXTERM terminates the third language processor environment that the system initialized for the second ISPF screen. Similarly, when the user exits from ISPF and returns to TSO/E READY mode, IRXTERM terminates the language processor environment for the first ISPF screen. In TSO/E READY mode, the first language processor environment still exists. At this point, if the user invokes a REXX exec from READY mode, the exec runs under the environment that IRXINIT initialized during TSO/E logon. When the user logs off, IRXTERM terminates the language processor environment for the TSO/E session.

To summarize, the IRXINIT routine automatically initializes a language processor environment when a user logs on to TSO/E and whenever an ISPF screen is initialized. Each environment that IRXINIT initializes is separate from another environment. The IRXTERM routine automatically terminates the language processor environment for an ISPF screen when the screen session ends and terminates the environment created at TSO/E logon when the user logs off.

You can also invoke the IRXINIT routine to initialize a language processor environment. On the call to IRXINIT, you specify values you want defined for the new environment. Using IRXINIT gives you the ability to define a language processor environment and *customize* how REXX execs run and how the system accesses and uses system services. Using IRXINIT to initialize environments is particularly important in non-TSO/E address spaces where you may want to provide replaceable routines to handle system services. However, you may want to use IRXINIT in TSO/E to create an environment that is similar to a non-TSO/E address space to test any replaceable routines or REXX execs you have developed for non-TSO/E.

If you explicitly invoke IRXINIT to initialize a language processor environment, you must invoke the IRXTERM routine to terminate the environment. The system does not terminate language processor environments that you initialized by calling IRXINIT. Information about IRXINIT and IRXTERM is described later in this chapter. Chapter 15, "Initialization and termination routines," on page 371 provides reference information about the parameters and return codes for IRXINIT and IRXTERM.

Initializing environments for user-written TMPs

If your installation uses a user-written terminal monitor program (TMP) instead of the TMP provided by TSO/E, the system does not automatically initialize a language processor environment in the TSO/E address space when a user logs on to TSO/E. That is, the system does not initialize a language processor environment for TSO/E READY mode. A language processor environment is required for processing REXX execs. To allow users to invoke REXX execs from TSO/E READY mode, your user-written TMP must invoke the initialization routine, IRXINIT, to initialize a language processor environment. To initialize the environment, the TMP must do the following:

- Invoke the initialization routine, IRXINIT, to initialize a language processor environment. The environment must be integrated into TSO/E, that is, the TSOFL flag must be on. On the call to IRXINIT, you can provide parameters that are equivalent to the default values that TSO/E provides in the IRXTSPRM default parameters module.
- If the TMP is not using the STACK ENVIRON=CREATE service to obtain a new ECT (that is, the user-written TMP is obtaining its own storage for the ECT), the TMP must ensure that the ECTEXTPR field is set to zeros. If the TMP is using the STACK ENVIRON=CREATE service to obtain the ECT, you should not set the ECTEXTPR field.
- When all user-written TMP processing is completed, you must invoke the termination routine, IRXTERM, to terminate the language processor environment that IRXINIT initialized. The system does not automatically terminate the environment.

The following topics in this chapter describe the characteristics of a language processor environment, the different types of environments, and the default parameters modules that TSO/E provides. Chapter 15, "Initialization and termination routines," on page 371 describes the initialization and termination routines IRXINIT and IRXTERM.

When environments are automatically initialized in MVS

As described in the previous topic, the system automatically initializes a language processor environment in the TSO/E address space whenever a user logs on to TSO/E and when a user invokes ISPF. After a TSO/E session has been started, users can simply invoke a REXX exec and the exec runs in the language processor environment in which it was invoked.

In non-TSO/E address spaces, the system does not automatically initialize language processor environments at a specific point, such as when the address space is activated. The system initializes an environment whenever you invoke the IRXJCL or IRXEXEC routine to invoke a REXX exec if an environment does not already exist on the current task.

TSO/E provides the TSO/E environment service, IKJTSOEV, that lets you create a TSO/E environment in a non-TSO/E address space. If you invoke IKJTSOEV to create a TSO/E environment, IKJTSOEV also initializes a REXX language processor environment within that TSO/E environment. IKJTSOEV initializes the language processor environment only if another language processor environment does not already exist in that address space. See *z/OS TSO/E Programming Services* for more information about the TSO/E environment service, IKJTSOEV.

You can run a REXX exec in MVS batch by specifying IRXJCL as the program on the JCL EXEC statement. You can invoke either the IRXJCL or IRXEXEC routines from a program in any address space to invoke an exec. "Exec processing routines - IRXJCL and IRXEXEC" on page 245 describes the two routines in detail.

When the IRXJCL or IRXEXEC routine is called, the routine determines whether a language processor environment already exists. (As discussed previously, more than one environment can be initialized in a single address space. The environments are chained together in a hierarchical structure). IRXJCL or IRXEXEC do not invoke IRXINIT to initialize an environment if an environment already exists. The routines use the current environment to run the exec. "Chains of environments and how environments are located" on page 340 describes how language processor environments are chained together and how environments are located.

If either IRXEXEC or IRXJCL invokes the IRXINIT routine to initialize an environment, after the REXX exec completes processing, the system calls the IRXTERM routine to terminate the environment that IRXINIT initialized.

Tip: If several language processor environments already exist when you invoke IRXJCL or IRXEXEC, you can pass the address of an environment block in register 0 on the call to indicate the environment in which the exec should run. For more information, see "Using the environment block" on page 313.

Types of environments - integrated and not integrated into TSO/E

There are two types of language processor environments:

- Environments that are integrated into TSO/E
- Environments that are not integrated into TSO/E.

The type of language processor environment that IRXINIT initializes depends on the address space in which IRXINIT creates the environment. Whether a language processor environment is integrated into TSO/E is determined by the setting of the TSOFL flag (see "Flags and corresponding masks" on page 322). The TSOFL flag is one characteristic (parameter) that IRXINIT uses to initialize a new environment. If the TSOFL flag is off, the new environment is not integrated into TSO/E. If the flag is on, the environment is integrated into TSO/E.

In non-TSO/E address spaces, language processor environments cannot be integrated into TSO/E. Therefore, when the system automatically initializes an environment in a non-TSO/E address space, the TSOFL flag is off. Similarly, if you explicitly invoke the initialization routine (IRXINIT) to initialize an environment in a non-TSO/E address space, the TSOFL flag must be off.

In the TSO/E address space, a language processor environment may or may not be integrated into TSO/E; that is, the TSOFL flag can be on or off. When the system automatically initializes an environment in the TSO/E address space, the environment is integrated into TSO/E (the TSOFL flag is on). If you explicitly invoke the initialization routine, IRXINIT, to initialize an environment in the TSO/E address space, the environment may or may not be integrated into TSO/E. That is, the TSOFL flag can be on or off. You may want to initialize an environment in the TSO/E address space that is not integrated into TSO/E. This lets you initialize an environment that is the same as an environment for a non-TSO/E address space. By doing this, for example, you can test REXX execs you have written for a non-TSO/E address space.

The type of language processor environment affects two different aspects of REXX processing:

- The functions, commands, and services you can use in a REXX exec itself
- The different characteristics (parameters) that define the language processor environment that IRXINIT initializes.

The following topics describe the two aspects of REXX processing.

Functions, Commands, and Services in an Exec: The type of language processor environment in which a REXX exec runs affects the kinds of functions, commands, and services you can use in the exec itself. If the exec runs in an environment that is integrated into TSO/E, you can use TSO/E commands, such as ALLOCATE, TEST, and PRINTDS in the exec. You can also use TSO/E programming services, such as the parse service routine (IKJPARS) and the dynamic allocation interface routine (DAIR). The TSO/E programming service routines are described in *z/OS TSO/E Programming Services*. In addition, the exec can use all the TSO/E external functions, ISPF services, and can invoke and be invoked by CLISTs.

If an exec runs in an environment that is not integrated into TSO/E, the exec cannot contain TSO/E commands or the TSO/E service routines, such as IKJPARS and DAIR, or use ISPF services or CLISTs. The exec can use the TSO/E external functions SETLANG and STORAGE only. The exec cannot use the other TSO/E external functions, such as MSG and OUTTRAP. Chapter 8, "Using REXX in different address spaces," on page 187 describes the instructions, functions, commands, and services you can use in REXX execs that you write for TSO/E and for non-TSO/E address spaces.

Different Characteristics for the Environment: When IRXINIT initializes a language processor environment, IRXINIT defines different characteristics for the environment. The three parameters modules TSO/E provides (IRXTSPRM, IRXISPRM, and IRXPARMS) define the default values for initializing environments. If you provide your own parameters module or explicitly invoke the initialization routine (IRXINIT), the characteristics you can define for the environment depend on the type of environment.

Some characteristics can be used for any type of language processor environment. In some cases, the values you specify may differ depending on the environment. Other characteristics can be specified only for environments that are integrated into TSO/E or for environments that are not integrated into TSO/E. For example, you can provide your own replaceable routines only for environments that are not integrated into TSO/E. TSO/E also provides exit routines for REXX processing. In general, you can provide exits for any type of language processor environment (integrated and not integrated into TSO/E). One exception is the attention handling exit, which is only for environments that are integrated into TSO/E. Chapter 16, "Replaceable routines and exits," on page 389 describes the replaceable routines and exits in more detail.

"Specifying values for different environments" on page 348 describes the environment characteristics you can specify for language processor environments that either are or are not integrated into TSO/E.

Characteristics of a Language Processor Environment

When IRXINIT initializes a language processor environment, IRXINIT creates several control blocks that contain information about the environment. One of the control blocks is the parameter block (PARMBLOCK). The parameter block contains the parameter values that IRXINIT used to define the environment, that is, the parameter block contains the characteristics that define the environment. The block also contains the addresses of the module name table, the host command environment table, and the function package table, which contain additional characteristics for the environment.

TSO/E provides three default *parameters modules*, which are load modules that contain the values for initializing language processor environments. The three default modules are IRXPARMS (MVS), IRXTSPRM (TSO/E), and IRXISPRM (ISPF). "Values provided in the three default parameters modules" on page 336 shows the default values that TSO/E provides in each of these modules. A parameters module consists of the parameter block (PARMBLOCK), the module name table, the host command environment table, and the function package table. Figure 20 on page 318 shows the format of the parameters module.

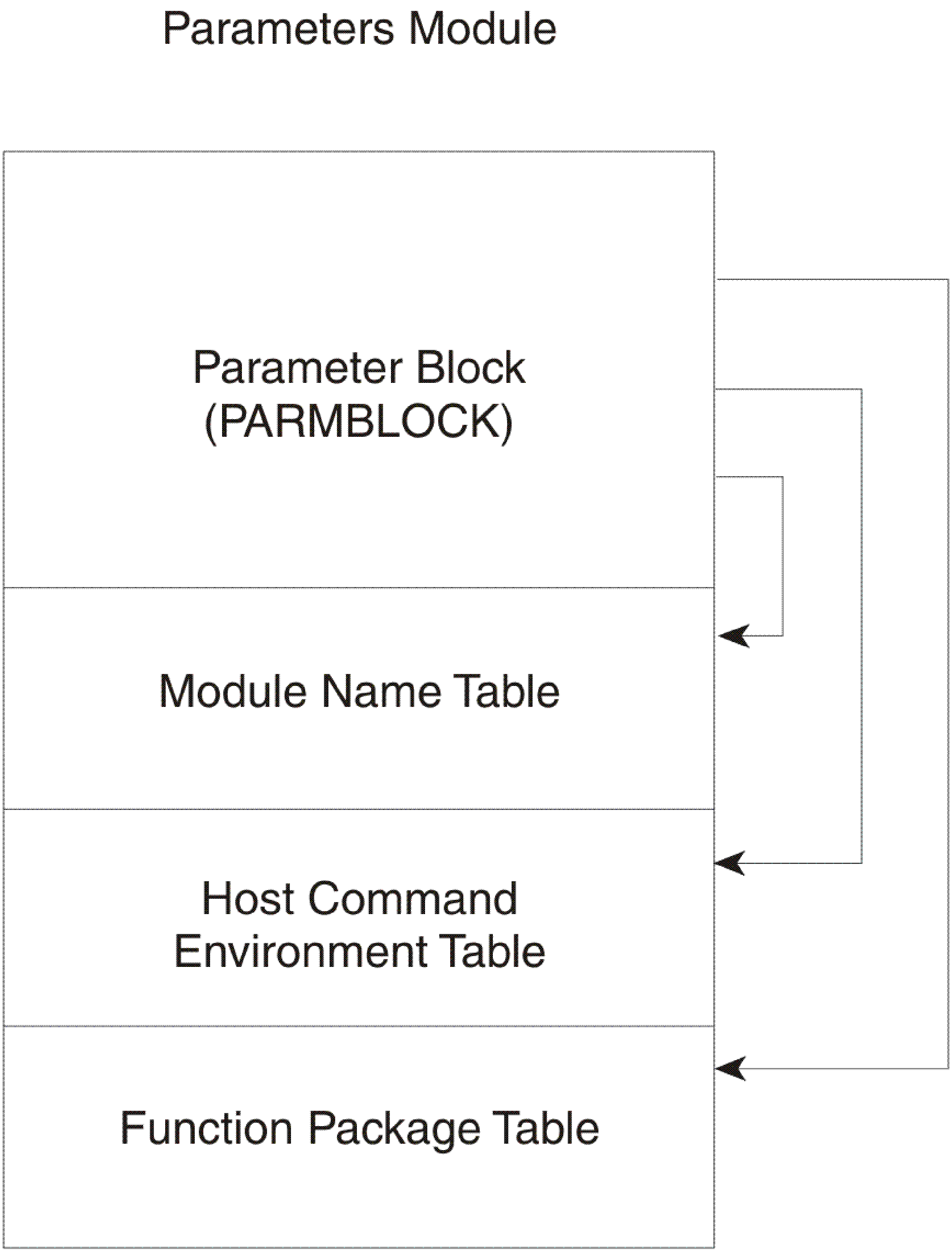


Figure 20. Overview of parameters module

Table 54 on page 319 shows the format of PARMBLOCK. Each field is described in more detail following the table. The end of the PARMBLOCK must be indicated by X'FFFFFFFFFFFFFFFF'. The format of the module name table, host command environment table, and function package table are described in subsequent topics.

Table 54. Format of the parameter block (PARMBLOCK)

Offset (decimal)	Number of bytes	Field name	Description
0	8	ID	Identifies the parameter block (PARMBLOCK).
8	4	VERSION	Identifies the version of the parameter block.
12	3	LANGUAGE	Language code for REXX messages.
15	1	RESERVED	Reserved.
16	4	MODNAMET	Address of module name table.
20	4	SUBCOMTB	Address of host command environment table.
24	4	PACKTB	Address of function package table.
28	8	PARSETOK	Token for PARSE SOURCE instruction.
36	4	FLAGS	A fullword of bits that IRXINIT uses as flags to define characteristics for the environment.
40	4	MASKS	A fullword of bits that IRXINIT uses as a mask for the setting of the flag bits.
44	4	SUBPOOL	Number of the subpool for storage allocation.
48	8	ADDRSPN	Name of the address space.
56	8	—	The end of the PARMBLOCK must be indicated by X'FFFFFFFFFFFFFFFF'.

The following information describes each field in the PARMBLOCK. If you change any of the default parameters modules that TSO/E provides or you use IRXINIT to initialize a language processor environment, read "Changing the default values for initializing an environment" on page 344, which provides information about changing the different values that define an environment.

ID

An 8-byte character field that is used only to identify the parameter block that IRXINIT creates. The field name is ID.

The value that TSO/E provides in the three default parameters modules is IRXPARMS. You must not change the value in the ID field in any of the parameters modules.

Version

A 4-byte character field that identifies the version of the parameter block for a particular release and level of TSO/E. The field name is VERSION.

The value that TSO/E provides in the three default parameters modules is 0200. You must not change the Version field in any of the parameters modules.

Language Code

A 3-byte field that contains a language code. The field name is LANGUAGE.

The language code identifies the language in which REXX messages are displayed. The default that TSO/E provides in all three parameters modules is ENU, which is the language code for US English in mixed case (upper and lowercase). The possible values are:

- CHS – Simplified Chinese
- CHT – Traditional Chinese
- DAN – Danish
- DEU – German
- ENP – US English in uppercase

- ENU – US English in mixed case (upper and lowercase)
- ESP – Spanish
- FRA – French
- JPN – Japanese (Kanji)
- KOR – Korean
- PTB – Brazilian Portuguese

Reserved
A 1-byte field that is reserved.

Module Name Table
A 4-byte field that contains the address of the module name table. The field name is MODNAMET.

The table contains the ddnames for reading and writing data and for loading REXX execs, the names of several replaceable routines, and the names of several exit routines. "Module name table" on page 326 describes the table in detail.

Host Command Environment Table
A 4-byte field that contains the address of the host command environment table. The field name is SUBCOMTB.

The table contains the names of the host command environments for processing host commands. These are the environments that REXX execs can specify using the ADDRESS instruction. "Commands to external environments" on page 22 describes how to issue host commands from a REXX exec and the different environments TSO/E provides for command processing.

The table also contains the names of the routines that are invoked to handle the processing of commands that are issued in each host command environment. "Host command environment table" on page 331 describes the table in detail.

Function Package Table
A 4-byte field that contains the address of the function package table for function packages. The field name is PACKTB. "Function package table" on page 333 describes the table in detail.

Token for PARSE SOURCE
An 8-byte character string that contains the value of a token that the PARSE SOURCE instruction uses. The field name is PARSETOK. The default that TSO/E provides in all three parameters modules is a blank.

This token is the last token of the string that PARSE SOURCE returns. Every PARSE SOURCE instruction processed in the environment returns the token.

Flags
A fullword of bits that IRXINIT uses as flags. The field name is FLAGS.

The flags define certain characteristics for the new language processor environment and how the environment and execs running in the environment operate.

In addition to the flags field, the parameter following the flags is a *mask* field that works together with the flags. The mask field is a string that has the same length as the flags field. Each bit position in the mask field corresponds to a bit position in the flags field. IRXINIT uses the mask field to determine whether it should use or ignore the corresponding flag bit.

See **Mask** for a description of the bit settings for the mask field and how the value for each flag is determined.

Table 55 on page 321 summarizes each flag. "Flags and corresponding masks" on page 322 describes each of the flags in more detail and the bit settings for each flag. The mapping of the parameter block (PARMBLOCK) includes the mapping of the flags. TSO/E provides a mapping macro IRXPARMB for the parameter block. The mapping macro is in SYS1.MACLIB.

Table 55. Summary of each flag bit in the parameters module

Bit position number	Flag name	Description
0	TSOFL	Indicates whether the new environment is to be integrated into TSO/E.
1	Reserved	This bit is reserved.
2	CMDSOFL	Specifies the search order the system uses to locate a command.
3	FUNCSOFL	Specifies the search order the system uses to locate functions and subroutines.
4	NOSTKFL	Prevents REXX execs running in the environment from using any data stack operations.
5	NOREADFL	Prevents REXX execs running in the environment from reading any input file.
6	NOWRTFL	Prevents REXX execs running in the environment from writing to any output file.
7	NEWSTKFL	Indicates whether a new data stack is initialized for the new environment.
8	USERPKFL	Indicates whether the user function packages that are defined for the previous language processor environment are also available in the new environment.
9	LOCPKFL	Indicates whether the local function packages that are defined for the previous language processor environment are also available in the new environment.
10	SYSPKFL	Indicates whether the system function packages that are defined for the previous language processor environment are also available in the new environment.
11	NEWSCFL	Indicates whether the host command environments (as specified in the host command environment table) that are defined for the previous language processor environment are also available in the new environment.
12	CLOSEXFL	Indicates whether the data set from which REXX execs are obtained is closed after an exec is loaded or remains open.
13	NOESTAE	Indicates whether a recovery ESTAE is permitted under the environment.
14	RENTRANT	Indicates whether the environment is initialized as either reentrant or non-reentrant.
15	NOPMSGS	Indicates whether primary messages are printed.
16	ALTMSGS	Indicates whether alternate messages are printed.
17	SPSHARE	Indicates whether the subpool specified in the SUBPOOL field is shared across MVS tasks.
18	STORFL	Indicates whether REXX execs running in the environment can use the STORAGE function.
19	NOLOADDD	Indicates whether the DD specified in the LOADDD field in the module name table is searched for execs.
20	NOMSGWTO	Indicates whether REXX messages are processed normally in the environment or if they should be routed to a file.
21	NOMSGIO	Indicates whether REXX messages are processed normally in the environment or if they should be routed to a JCL listing.

Table 55. Summary of each flag bit in the parameters module (continued)

Bit position number	Flag name	Description
22	ROSTORFL	Indicates whether the STORAGE function should be limited to read-only mode. In read-only mode the STORAGE function can read but not change storage. (This flag is meaningful only if STORFL is OFF so that the STORAGE function itself is allowed).
23	Reserved	The remaining bits are reserved.

Mask

A fullword of bits that IRXINIT uses as a mask for the setting of the flag bits. See**Flags** for a description of the flags field.

The field name is MASKS. The mask field is a string that has the same length as the flags field. Each bit position in the mask field corresponds to a bit in the same position in the flags field. IRXINIT uses the mask field to determine whether it should use or ignore the corresponding flag bit. For a given bit position, if the value in the mask field is:

- 0 — the corresponding bit in the flags field is ignored (that is, the bit is considered null)
- 1 — the corresponding bit in the flags field is used.

Subpool Number

A fullword that specifies the number of the subpool (in binary) in which storage is allocated for the entire language processor environment. The field name is SUBPOOL. The default value in the IRXPARMS module is 0. The value can be from 0 to 127 in decimal.

In the IRXTSPRM and IRXISPRM modules, the default is 78 (in decimal). For environments that are integrated into TSO/E (see "Types of environments - integrated and not integrated into TSO/E" on page 316), the subpool number must be 78.

Address Space Name

An 8-byte character field that specifies the name of the address space. The field name is ADDRSPN. TSO/E provides the following defaults:

- IRXPARMS module – MVS
- IRXTSPRM module – TSO/E
- IRXISPRM module – ISPF

X'FFFFFFFFFFFFFFFF'

The end of the parameter block is indicated by X'FFFFFFFFFFFFFFFF'.

Flags and corresponding masks

This topic describes the flags field.

TSOFL

The TSOFL flag indicates whether IRXINIT should integrate the new language processor environment into TSO/E. That is, the flag indicates whether REXX execs that run in the environment can use TSO/E services and commands.

0 — The environment is *not* integrated into TSO/E.
1 — The environment is integrated into the TSO/E.

You can initialize an environment in the TSO/E address space and set the TSOFL flag off. In this case, any REXX execs that run in the environment must not use any TSO/E commands or services. If they do, unpredictable results can occur.

Setting the TSOFL off for an environment that is initialized in the TSO/E address space lets you provide your own replaceable routines for different system services, such as I/O and data stack requests. It also lets you test REXX execs in an environment that is similar to a language processor environment that is initialized in a non-TSO/E address space.

If the TSOFL flag is on, there are many values that you cannot specify in the parameter block. "Specifying values for different environments" on page 348 describes the parameters you can use for environments that are integrated into TSO/E and for environments that are not integrated into TSO/E.

Restriction: The TSOFL flag cannot be set to 1 if a previous environment contains a TSOFL flag set to 0. Essentially, if the previous environment is not integrated into TSO/E, a newly created environment cannot be integrated into TSO/E.

Reserved

This bit is reserved.

CMDSOFL

The CMDSOFL flag is the command search order flag. The flag specifies the search order the system uses to locate a command that is issued from an exec.

0 — Search for modules first, followed by REXX execs, followed by CLISTs. The ddname the system uses to search for REXX execs is specified in the LOADDD field in the module name table.

1 — Search for REXX execs first, followed by modules, followed by CLISTs. The ddname the system uses to search for REXX execs is specified in the LOADDD field in the module name table.

FUNCSOFL

The FUNCSOFL flag is the external function or subroutine search order flag. The flag specifies the search order the system uses to locate external functions and subroutines that an exec calls.

- 0 — Search load libraries first. If the function or subroutine is not found, search for a REXX exec.
- 1 — Search for a REXX exec. If the exec is not found, search the load libraries.

"Search order" on page 74 describes the complete search order TSO/E uses to locate an exec.

NOSTKFL

The NOSTKFL flag is the no data stack flag. Use the flag to prevent REXX execs running in the environment from using the data stack.

0 — A REXX exec can use the data stack.

1 — Attempts to use the data stack are processed as though the data stack were empty. Any data that is pushed (PUSH) or queued (QUEUE) is lost. A PULL operates as though the data stack were empty. The QSTACK command returns a 0. The NEWSTACK command seems to work, but a new data stack is not created and any subsequent data stack operations operate as if the data stack is permanently empty.

NOREADFL

The NOREADFL flag is the no read flag. Use the flag to prevent REXX execs from reading any input file using either the EXECIO command or the system-supplied I/O replaceable routine IRXINOUT.

0 — Reads from any input file are permitted.

1 — Reads from any input file are not permitted.

NOWRTFL

The NOWRTFL flag is the no write flag. Use the flag to prevent REXX execs from writing to any output file using either the EXECIO command or the system-supplied I/O replaceable routine IRXINOUT.

0 — Writes to any output file are permitted.

1 — Writes to any output file are not permitted.

NEWSTKFL

The NEWSTKFL flag is the new data stack flag. Use the flag to specify whether IRXINIT should initialize a new data stack for the language processor environment. If IRXINIT creates a new data stack, any REXX exec or other program that runs in the new environment cannot access any data stacks for previous environments. Any subsequent environments that are initialized under this environment will access the data stack that was most recently created by the NEWSTKFL flag. The first environment that is initialized on any chain of environments is always initialized as though the NEWSTKFL flag is on, that is, IRXINIT automatically creates a new data stack.

When you terminate the environment that is initialized, the data stack that was created at the time of initialization is deleted regardless of whether the data stack contains any elements. All data on that data stack is lost.

0 — IRXINIT does not create a new data stack. However, if this is the first environment being initialized on a chain, IRXINIT automatically initializes a data stack.
1 — IRXINIT creates a new data stack during the initialization of the new language processor environment. The data stack will be deleted when the environment is terminated.

"Using the data stack in different environments" on page 366 describes the data stack in different environments.

Note: The NOSTKFL overrides the setting of the NEWSTKFL.

USERPKFL

The USERPKFL flag is the user package function flag. The flag determines whether the user function packages that are defined for the previous language processor environment are also available to the new environment.

0 — The user function packages from the previous environment are added to the user function packages for the new environment.
1 — The user function packages from the previous environment are not added to the user function packages for the new environment.

LOCPKFL

The LOCPKFL flag is the local function package flag. The flag determines whether the local function packages that are defined for the previous language processor environment are also available to the new environment.

0 — The local function packages from the previous environment are added to the local function packages for the new environment.
1 — The local function packages from the previous environment are not added to the local function packages for the new environment.

SYSPKFL

The SYSPKFL flag is the system function package flag. The flag determines whether the system function packages that are defined for the previous language processor environment are also available to the new environment.

0 — The system function packages from the previous environment are added to the system function packages for the new environment.
1 — The system function packages from the previous environment are not added to the system function packages for the new environment.

NEWSCFL

The NEWSCFL flag is the new host command environment table flag. The flag determines whether the environments for issuing host commands that are defined for the previous language processor environment are also available to execs running in the new environment.

0 — The host command environments from the previous environment are added to the host command environment table for the new environment.
1 — The host command environments from the previous environment are not added to the host command environment table for the new environment.

CLOSEXFL

The CLOSEXFL flag is the close data set flag. The flag determines whether the data set (specified in the LOADDD field in the module name table) from which execs are fetched is closed after the exec is loaded or remains open.

The CLOSEXFL flag is needed if you are editing REXX execs and then running the changed execs under the same language processor environment. If the data set is not closed, results may be unpredictable.

0 — The data set is opened once and remains open.
1 — The data set is opened for each load and then closed.

NOESTAE

The NOESTAE flag is the no ESTAE flag. The flag determines whether a recovery ESTAE is established under the environment.

0 — IRXINIT establishes a recovery ESTAE.
1 — IRXINIT does not establish a recovery ESTAE.

When IRXINIT initializes the environment, IRXINIT first temporarily establishes a recovery ESTAE regardless of the setting of the NOESTAE flag. However, if the NOESTAE flag is on, IRXINIT removes the recovery ESTAE for the environment before IRXINIT finishes processing.

RENTRANT

The RENTRANT flag is the initialize reentrant language processor environment flag. The flag determines whether IRXINIT initializes the new environment as a reentrant or a non-reentrant environment.

0 — IRXINIT initializes a non-reentrant language processor environment.
1 — IRXINIT initializes a reentrant language processor environment.

For information about reentrant environments, see "Using the environment block for reentrant environments" on page 244.

NOPMSGS

The NOPMSGS flag is the primary messages flag. The flag determines whether REXX primary messages are printed in the environment.

0 — Primary messages are printed.
1 — Primary messages are not printed.

ALTMSGS

The ALTMSGS flag is the alternate messages flag. The flag determines whether REXX alternate messages are printed in the environment.

0 — Alternate messages are not printed.
1 — Alternate messages are printed.

Note: Alternate messages are also known as secondary messages.

SPSHARE

The SPSHARE flag is the sharing subpools flag. The flag determines whether the subpool specified in the SUBPOOL field in the module name table should be shared across MVS tasks.

0 — The subpool is not shared.
1 — The subpool is shared.

If the subpool is shared, REXX uses the same subpool for all of these tasks.

STORFL

The STORFL flag is the STORAGE function flag. The flag controls the STORAGE external function and indicates whether REXX execs running in the environment can use the STORAGE function.

- 0 — Execs can use the STORAGE external function.
- 1 — Execs cannot use the STORAGE external function.

NOLOADDD

The NOLOADDD flag is the exec search order flag. The flag controls the search order for REXX execs. The flag indicates whether the system should search the data set specified in the LOADDD field in the module name table.

0 — The system searches the DD specified in the LOADDD field.
1 — The system does not search the DD specified in the LOADDD field.

With the defaults that TSO/E provides, the NOLOADDD flag is off (0), which means the system searches the DD specified in the LOADDD field. The default ddname is SYSEXEC. If the language processor environment is integrated into TSO/E, the system searches SYSEXEC followed by SYSPROC. For more information, see "Using SYSPROC and SYSEXEC for REXX execs" on page 353.

"Search order" on page 74 describes the complete search order TSO/E uses to locate an exec.

NOMSGWTO and NOMSGIO

Together, these two flags control where REXX error messages are routed when running in a language processor environment that is not integrated into TSO/E.

Table 56. Flag settings for NOMSGWTO and NOMSGIO

NOMSGWTO	NOMSGIO	
0	0	Error messages are issued using the WTO service (ROUTCDE 11), and typically go to the JCL listing. In addition, if REXX tracing is active at the time an error message is being written, the message is also written to the OUTDD file defined by the OUTDD field in the module name table. SYSTSPRT is the default OUTDD file. When an exec is initially invoked, TRACE 'Normal' is active by default. Unless the exec turns off tracing (TRACE 'Off'), error messages are written to both the JCL listing and the OUTDD file when both the NOMSGWTO and NOMSGIO flags are off.
1	0	REXX error messages cannot be written by using WTO. Instead, error messages are written to the OUTDD file. This happens regardless of whether REXX tracing is active.
0	1	REXX error messages cannot be written to the OUTDD file. Instead, error messages are written by using WTO. This happens regardless of whether REXX tracing is active.
1	1	REXX error messages are suppressed, regardless of whether REXX tracing is active.

The default flag settings provided by TSO/E are off (0) for both NOMSGWTO and NOMSGIO.

REXX error messages include all of the REXX messages numbered IRXnnnnE or IRXnnnnI, where nnnn is the message number. Error messages also include any text written to the error message output stream using the 'WRITEERR' function of IRXSAY.

ROSTORFL

The ROSTORFL flag is the read-only STORAGE flag. This flag controls the execution mode of the STORAGE external function. When this flag is set, STORAGE will execute in read-only mode.

0 — STORAGE function, if allowed, can both read and alter storage.
1 — STORAGE function, if allowed, can read but not alter storage.

This flag is meaningful only if the STORFL flag is OFF so that the STORAGE function itself is allowed.

Reserved

The remaining bits are reserved.

Module name table

The module name table contains the names of:

- The DDs for reading and writing data
- The DD from which to load REXX execs
- Replaceable routines
- Several exit routines

In the parameter block, the MODNAMET field points to the module name table (see "Characteristics of a Language Processor Environment" on page 317).

Table 57 on page 327 shows the format of the module name table. Each field is described in detail following the table. The end of the table is indicated by X'FFFFFFFFFFFFFFFF'. TSO/E provides a mapping macro IRXMODNT for the module name table. The mapping macro is in SYS1.MACLIB.

Table 57. Format of the module name table

Offset (decimal)	Number of bytes	Field name	Description
0	8	INDD	The DD from which the PARSE EXTERNAL instruction reads input data.
8	8	OUTDD	The DD to which data is written for either a SAY instruction, for REXX error messages, or when tracing is started.
16	8	LOADDD	The DD from which REXX execs are fetched.
24	8	IOROUT	The name of the input/output (I/O) replaceable routine.
32	8	EXROUT	The name of the exec load replaceable routine.
40	8	GETFREER	The name of the storage management replaceable routine.
48	8	EXECINIT	The name of the exec initialization exit routine.
56	8	ATTNROUT	The name of an attention handling exit routine.
64	8	STACKRT	The name of the data stack replaceable routine.
72	8	IRXEXECX	The name of the exit routine for the IRXEXEC routine.
80	8	IDROUT	The name of the user ID replaceable routine.
88	8	MSGIDRT	The name of the message identifier replaceable routine.
96	8	EXECTERM	The name of the exec termination exit routine.
152	8	—	The end of the module name table must be indicated by X'FFFFFFFFFFFFFFFF'.

Each field in the module name table is described below. You can specify some fields for any type of language processor environment. You can use other fields only for environments that are integrated into TSO/E or for environments that are not integrated into TSO/E. The description of each field below indicates the type of environment for which you can use the field. "Relationship of fields in module name table to types of environments" on page 330 summarizes the fields in the module name table and the environments for which you can specify each field.

INDD

Specifies the name of the DD from which the PARSE EXTERNAL instruction reads input data (in a language processor environment that is not integrated into TSO/E). The system default is SYSTSIN.

If the environment is integrated into TSO/E (the TSOFL flag is on), the system ignores any value you specify for INDD. In TSO/E foreground, TSO/E uses the terminal. In the background, TSO/E uses the input stream, which is SYSTSIN.

OUTDD

Specifies the name of the DD to which data is written for a SAY instruction, for REXX error messages, or when tracing is started (in a language processor environment that is not integrated into TSO/E). The system default is SYSTSPRT.

If the environment is integrated into TSO/E (the TSOFL flag is on), the system ignores any value you specify for OUTDD. In TSO/E foreground, TSO/E uses the terminal. In the background, TSO/E uses the output stream, which is SYSTSPRT.

LOADDD

Specifies the name of the DD from which REXX execs are loaded. The default is SYSEXEC. You can specify a ddname in any type of language processor environment (integrated or not integrated into TSO/E).

In TSO/E, you can store REXX execs in data sets that are allocated to SYSEXEC or SYSPROC. If you store an exec in a data set that is allocated to SYSPROC, the exec must start with a comment containing the characters **REXX** within the first line (line 1). This is required to distinguish REXX execs from CLISTs that are also stored in SYSPROC.

In data sets that are allocated to SYSEXEC, you can store REXX execs only, not CLISTs. If you store an exec in SYSEXEC, the exec does not need to start with a comment containing the characters "REXX". However, it is suggested that you start all REXX programs with a comment regardless of where you store them. SYSEXEC is useful for REXX execs that follow the SAA Procedures Language standards and that will be used on other SAA environments.

The NOLOADDD flag (see "Flags and corresponding masks" on page 322) controls whether the system searches the DD specified in the LOADDD field.

- If the NOLOADDD flag is off, the system searches the DD specified in the LOADDD field. If the language processor environment is integrated into TSO/E and the exec is not found, the system then searches SYSPROC.
- If the NOLOADDD flag is on, the system does not search the DD specified in the LOADDD field. However, if the language processor environment is integrated into TSO/E, the system searches SYSPROC.

In the default parameters module that is provided for TSO/E (IRXTSPRM), the NOLOADDD mask and flag settings indicate that SYSEXEC is searched before SYSPROC. (Note that before TSO/E 2.3, the default settings indicated that SYSPROC only was searched). In the default parameters module for ISPF (IRXISPRM), the defaults indicate that the environment inherits the values from the previous environment, which is the environment initialized for TSO/E. By default, the system searches the ddname specified in the LOADDD field (SYSEXEC). To use SYSPROC exclusively, you can provide your own parameters module or use the EXECUTIL SEARCHDD command. For more information, see "Using SYSPROC and SYSEXEC for REXX execs" on page 353.

IOROUT

Specifies the name of the routine that is called for input and output operations. The routine is called for:

- The PARSE EXTERNAL, SAY, and TRACE instructions when the exec is running in an environment that is not integrated into TSO/E
- The PULL instruction when the exec is running in an environment that is not integrated into TSO/E and the data stack is empty
- Requests from the EXECIO command
- Issuing REXX error messages

You can specify an I/O replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see "Input/Output routine" on page 403.

EXROUT

Specifies the name of the routine that is called to load and free a REXX exec. The routine returns the structure that is described in “The in-storage control block (INSTBLK)” on page 256. The specified routine is called to load and free this structure.

You can specify an exec load replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see “Exec load routine” on page 392.

GETFREER

Specifies the name of the routine that is called when storage is to be obtained or freed. If this field is blank, TSO/E storage routines handle storage requests and use the GETMAIN and FREEMAIN macros when larger amounts of storage must be handled.

You can specify a storage management replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see “Storage management routine” on page 420.

EXECINIT

Specifies the name of an exit routine that gets control after the system initializes the REXX variable pool for a REXX exec, but before the language processor processes the first clause in the exec. The exit differs from other standard TSO/E exits. The exit does not have a fixed name. You provide the exit and specify the routine's name in the EXECINIT field. “REXX exit routines” on page 426 describes the exec initialization exit.

You can provide an exec initialization exit in any type of language processor environment (integrated or not integrated into TSO/E).

ATTNROUT

Specifies the name of an exit routine that is invoked if a REXX exec is processing in the TSO/E address space (in an environment that is integrated into TSO/E), and an attention interruption occurs. The attention handling exit differs from other standard TSO/E exits. The exit does not have a fixed name. You provide the exit and specify the routine's name in the ATTNROUT field. “REXX exit routines” on page 426 describes the attention handling exit.

You can provide an attention handling exit only in a language processor environment that is integrated into TSO/E.

STACKRT

Specifies the name of the routine that the system calls to handle all data stack requests.

You can specify a data stack replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see “Data stack routine” on page 415.

IRXEXECX

Specifies the name of an exit routine that is invoked whenever the IRXEXEC routine is called to run an exec. You can use the exit to check the parameters specified on the call to IRXEXEC, change the parameters, or decide whether IRXEXEC processing should continue.

The exit differs from other standard TSO/E exits. The exit does not have a fixed name. You provide the exit and specify the routine's name in the IRXEXECX field.

You can provide an exit for the IRXEXEC routine in any type of language processor environment (integrated or not integrated into TSO/E). For more information about the exit, see “REXX exit routines” on page 426.

IDROUT

Specifies the name of a replaceable routine that the system calls to obtain the user ID. The USERID built-in function returns the result that the replaceable routine obtains.

You can specify a user ID replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see “User ID routine” on page 422.

MSGIDRT

Specifies the name of a replaceable routine that determines whether the system should display the message identifier (message ID) with a REXX error message.

You can specify a message identifier replaceable routine only in language processor environments that are not integrated into TSO/E. For more information about the replaceable routine, see "Message identifier routine" on page 425.

EXECTERM

Specifies the name of an exit routine that gets control after the language processor processes a REXX exec, but before the system terminates the REXX variable pool. The exit differs from other standard TSO/E exits. The exit does not have a fixed name. You provide the exit and specify the routine's name in the EXECTERM field. "REXX exit routines" on page 426 describes the exit in more detail.

You can provide an exec termination exit in any type of language processor environment (integrated or not integrated into TSO/E).

X'FFFFFFFFFFFFFFFF'

The end of the module name table must be indicated by X'FFFFFFFFFFFFFFFF'.

Relationship of fields in module name table to types of environments

You can specify certain fields in the module name table regardless of the type of language processor environment. You can define other fields only if the language processor environment is integrated into TSO/E or the environment is not integrated into TSO/E.

Table 58 on page 330 lists each field in the module name table and indicates the type of environment where you can specify the field. An *X* in the Integrated Into TSO/E column indicates you can use the field for a language processor environment that is integrated into TSO/E. An *X* in the Not Integrated Into TSO/E column indicates you can use the field for a language processor environment that is not integrated into TSO/E.

Table 58. Summary of fields in module name table and types of environments

Field name in module name table	**Integrated into TSO/E**	**Not integrated into TSO/E**
INDD – ddname from which PARSE EXTERNAL reads input.		X
OUTDD – ddname to which data is written.		X
LOADDD – ddname from which execs are fetched.	X	X
IOROUT – name of input/output (I/O) replaceable routine.		X
EXROUT – name of exec load replaceable routine.		X
GETFREER – name of storage management replaceable routine.		X
EXECINIT – name of exec initialization exit routine.	X	X
ATTNROUT – name of attention handling exit routine.	X	
STACKRT – name of data stack replaceable routine.		X
IRXEXECX – name of exec processing exit for the IRXEXEC routine.	X	X
IDROUT – name of user ID replaceable routine.		X
MSGIDRT – name of message ID replaceable routine.		X
EXECTERM – name of exec termination exit routine.	X	X

Host command environment table

The host command environment table contains the names of environments for processing commands. The table contains the names you can specify on the ADDRESS instruction. In the parameter block, the SUBCOMTB field points to the host command environment table (see Table 54 on page 319).

The table contains the environment names (for example, TSO, MVS, LINK, and ATTACH) that are valid for execs that run in the language processor environment. The table also contains the names of the routines that the system invokes to handle "commands" for each host command environment.

You can add, delete, update, and query entries in the host command environment table using the IRXSUBCM routine. For more information, see “Maintain entries in the host command environment table - IRXSUBCM” on page 280.

When a REXX exec runs, the exec has at least one active host command environment that processes host commands. When the REXX exec begins processing, a default environment is available. The default is specified in the host command environment table. In the REXX exec, you can use the ADDRESS instruction to change the host command environment. When the language processor processes a command, the language processor first evaluates the expression and then passes the command to the host command environment for processing. A specific routine that is defined for that host command environment then handles the command processing. “Commands to external environments” on page 22 describes how to issue commands to the host.

In the PARMBLOCK, the SUBCOMTB field points to the host command environment table. The table consists of two parts; the table header and the individual entries in the table. Table 59 on page 331 shows the format of the host command environment table header. The first field in the header points to the first host command environment entry in the table. Each host command environment entry is defined by one row in the table. Each row contains the environment name, corresponding routine to handle the commands, and a user token. TSO/E provides a mapping macro IRXSUBCT for the host command environment table. The mapping macro is in SYS1.MACLIB.

Table 59. Format of the host command environment table header

Offset (decimal)	Number of bytes	Field name	Description
0	4	ADDRESS	Specifies the address of the first entry in the table. The address is a fullword binary number. Table 60 on page 332 illustrates each row of entries in the table. Each row of entries in the table has an 8-byte field (NAME) that contains the name of the environment, a second 8-byte field (ROUTINE) that contains the name of the corresponding routine, followed by a 16-byte field (TOKEN) that is a user token.
4	4	TOTAL	Specifies the total number of entries in the table. This number is the total of the used and unused entries in the table and is a fullword binary number.
8	4	USED	Specifies the number of valid entries in the table. The number is a fullword binary number. All valid entries begin at the top of the table and are then followed by any unused entries. The unused entries must be on the bottom of the table.
12	4	LENGTH	Specifies the length of each entry in the table. This is a fullword binary number.

Table 59. Format of the host command environment table header (continued)

Offset (decimal)	Number of bytes	Field name	Description
16	4	INITIAL	Specifies the name of the initial host command environment. This is the default environment for any REXX exec that is invoked and that is not invoked as either a function or a subroutine. The INITIAL field is used only if you call the exec processing routine IRXEXEC to run a REXX exec and you do not pass an initial host command environment on the call. "Exec processing routines - IRXJCL and IRXEXEC" on page 245 describes the IRXEXEC routine and its parameters.
20	8	—	Reserved. The field is set to blanks.
28	8	—	The end of the table header must be indicated by X'FFFFFFFFFFFFFFFF'.

Table 60 on page 332 shows three rows (three entries) in the host command environment table. The NAME, ROUTINE, and TOKEN fields are described in more detail after the table.

Table 60. Format of entries in host command environment table

Offset (decimal)	Number of bytes	Field name	Description
0	8	NAME	The name of the first environment (entry) in the table.
8	8	ROUTINE	The name of the routine that the system invokes to handle the processing of host commands in the environment specified at offset +0.
16	16	TOKEN	A user token that is passed to the routine (at offset +8) when the routine is invoked.
32	8	NAME	The name of the second environment (entry) in the table.
40	8	ROUTINE	The name of the routine that the system invokes to handle the processing of host commands in the environment specified at offset +32.
48	16	TOKEN	A user token that is passed to the routine (at offset +40) when the routine is invoked.
64	8	NAME	The name of the third environment (entry) in the table.
72	8	ROUTINE	The name of the routine that the system invokes to handle the processing of host commands in the environment specified at offset +64.
80	16	TOKEN	A user token that is passed to the routine (at offset +72) when the routine is invoked.

The following describes each entry (row) in the table.

NAME

An 8-byte field that specifies the name of the host command environment defined by this row in the table. The string is eight characters long, left justified, and is padded with blanks.

If the REXX exec uses the

```
ADDRESS name
```

instruction, and the value *name* in not in the table, no error is detected. However, when the language processor tries to locate the entry in the table to pass a command and no corresponding entry is found, the language processor returns with a return code of -3, which indicates an error condition.

ROUTINE

An 8-byte field that specifies the name of a routine for the entry in the NAME field in the same row in the table. This is the routine to which a string is passed for this environment. The field is eight characters long, left justified, and is padded with blanks.

If the language processor locates the entry in the table, but finds this field blank or cannot locate the routine specified, the language processor returns with a return code of -3. This is equivalent to the language processor not being able to locate the host command environment name in the table.

TOKEN

A 16-byte field that is stored in the table for the user's use (a user token). The value in the field is passed to the routine specified in the ROUTINE field when the system calls the routine to process a command. The field is for the user's own use. The language processor does not use or examine this token field.

When a REXX exec is running in the language processor environment and a host command environment must be located, the system searches the entire host command environment table from bottom to top. The first occurrence of the host command environment in the table is used. If the name of the host command environment that is being searched for matches the name specified in the table (in the NAME field), the system calls the corresponding routine specified in the ROUTINE field of the table.

Function package table

The function package table contains information about the function packages that are available for the language processor environment.

An individual user or an installation can write external functions and subroutines. For faster access of a function or subroutine, you can group frequently used external functions and subroutines in *function packages*. A function package is a number of external functions and subroutines that are grouped together. Function packages are searched before load libraries and execs (see “Search order” on page 74).

There are three types of function packages:

- User function packages
- Local function packages
- System function packages

User function packages are searched before local packages. Local function packages are searched before any system packages.

To provide a function package, there are several steps you must perform, including writing the code for the external function or subroutine, providing a function package directory for each function package, and defining the function package directory name in the function package table. “External functions and subroutines, and function packages” on page 263 describes function packages in more detail and how you can provide user, local, and system function packages.

In the parameter block, the PACKTB field points to the function package table (see “Characteristics of a Language Processor Environment” on page 317). The table contains information about the user, local, and system function packages that are available for the language processor environment. The function package table consists of two parts; the table header and table entries. Table 61 on page 334 shows the format of the function package table header. The header contains the total number of user, local, and system packages, the number of user, local, and system packages that are used, and the length of each function package name, which is always 8. The header also contains three addresses that point to the first table entry for user, local, and system function packages. The table entries specify the individual names of the function packages.

The table entries are a series of eight-character fields that are contiguous. Each eight-character field contains the name of a function package, which is the name of a load module containing the directory for that function package. The function package directory specifies the individual external functions and subroutines that make up one function package. “Directory for function packages” on page 269 describes the format of the function package directory in detail.

Figure 21 on page 335 illustrates the eight-character fields that contain the function package directory names for the three types of function packages (user, local, and system).

TSO/E provides a mapping macro for the function package table. The name of the mapping macro is IRXPACKT. The mapping macro is in SYS1.MACLIB.

Table 61. Function package table header

Offset (decimal)	Number of bytes	Field name	Description
0	4	USER_FIRST	Specifies the address of the first user function package entry. The address points to the first field in a series of eight-character fields that contain the names of the function package directories for user packages. Figure 21 on page 335 shows the series of directory names.
4	4	USER_TOTAL	Specifies the total number of user package table entries. This is the total number of function package directory names that are pointed to by the address at offset +0. You can use the USER_TOTAL field to specify the maximum number of user function packages that can be defined for the environment. You can then use the USER_USED field at offset +8 to specify the actual number of packages that are available.
8	4	USER_USED	Specifies the total number of user package table entries that are used. You can specify a maximum number (total) in the USER_TOTAL field at offset +4 and specify the actual number of user function packages that are used in the USER_USED field.
12	4	LOCAL_FIRST	Specifies the address of the first local function package entry. The address points to the first field in a series of eight-character fields that contain the names of the function package directories for local packages. Figure 21 on page 335 shows the series of directory names.
16	4	LOCAL_TOTAL	Specifies the total number of local package table entries. This is the total number of function package directory names that are pointed to by the address at offset +12. You can use the LOCAL_TOTAL field to specify the maximum number of local function packages that can be defined for the environment. You can then use the LOCAL_USED field at offset +20 to specify the actual number of packages that are available.
20	4	LOCAL_USED	Specifies the total number of local package table entries that are used. You can specify a maximum number (total) in the LOCAL_TOTAL field at offset +16 and specify the actual number of local function packages that are used in the LOCAL_USED field.
24	4	SYSTEM_FIRST	Specifies the address of the first system function package entry. The address points to the first field in a series of eight-character fields that contain the names of the function package directories for system packages. Figure 21 on page 335 shows the series of directory names.

Table 61. Function package table header (continued)

Offset (decimal)	Number of bytes	Field name	Description
28	4	SYSTEM_TOTAL	Specifies the total number of system package table entries. This is the total number of function package directory names that are pointed to by the address at offset +24. You can use the SYSTEM_TOTAL field to specify the maximum number of system function packages that can be defined for the environment. You can then use the SYSTEM_USED field at offset +32 to specify the actual number of packages that are available.
32	4	SYSTEM_USED	Specifies the total number of system package table entries that are used. You can specify a maximum number (total) in the SYSTEM_TOTAL field at offset +28 and specify the actual number of system function packages that are used in the SYSTEM_USED field.
36	4	LENGTH	Specifies the length of each table entry, that is, the length of each function package directory name. The length is always 8.
40	8	—	The end of the table is indicated by X'FFFFFFFFFFFFFFFF'.

Figure 21 on page 335 shows the function package table entries that are the names of the directories for user, local, and system function packages.

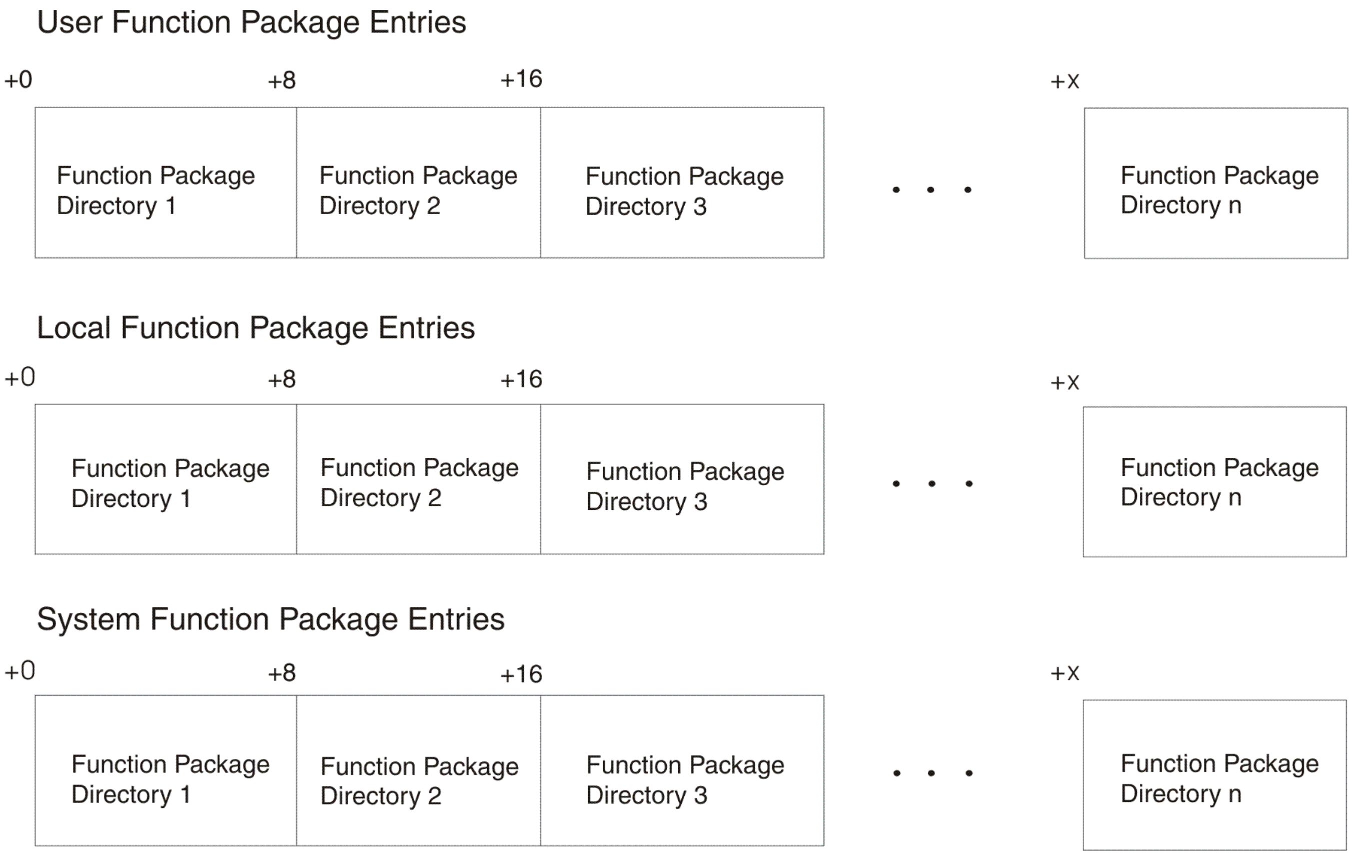

User Function Package Entries
+0
+8
+16
+X
Function Package Directory 1
Function Package Directory 2
Function Package Directory 3
. . .
Function Package Directory n
Local Function Package Entries
+0
+8
+16
+X
Function Package Directory 1
Function Package Directory 2
Function Package Directory 3
. . .
Function Package Directory n
System Function Package Entries
+0
+8
+16
+X
Function Package Directory 1
Function Package Directory 2
Function Package Directory 3
. . .
Function Package Directory n

Figure 21. Function package table entries – function package directories

The table entries are a series of eight-character fields. Each field contains the name of a function package directory. The directory is a load module that, when loaded, contains information about each external function and subroutine in the function package. "Directory for function packages" on page 269 describes the format of the function package directory in detail.

The function package directory names in each eight-character field must be left justified and padded with blanks.

Values provided in the three default parameters modules

Table 62 on page 336 shows the default values that TSO/E provides in each of the three default parameters modules. "Characteristics of a Language Processor Environment" on page 317 describes the structure of the parameters module in detail.

In Table 62 on page 336, the LANGUAGE field contains the language code ENU for US English in mixed case (upper and lowercase). The default parameters modules may contain a different language code depending on whether one of the language features has been installed on your system. See "Characteristics of a Language Processor Environment" on page 317 for information about the different language codes.

In the table, the value of each flag setting is followed by the value of its corresponding mask setting, in parentheses.

Note: Table 62 on page 336 shows the default values TSO/E provides in the parameters modules. It is *not* a mapping of a parameters module. For information about the format of a parameters module, see "Characteristics of a Language Processor Environment" on page 317. TSO/E provides the IRXPARMB mapping macro for the parameter block and the IRXMODNT, IRXSUBCT, and IRXPACKT mapping macros for the module name table, host command environment table, and function package table respectively.

Table 62. Values TSO/E provides in the three default parameters modules

Field name	IRXPARMS (MVS)	IRXTSPRM (TSO/E)	IRXISPRM (ISPF)
ID	IRXPARMS	IRXPARMS	IRXPARMS
VERSION	0200	0200	0200
LANGUAGE	ENU	ENU	
PARSETOK			
FLAGS (MASKS)			
TSOFL	0 (1)	1 (1)	1 (1)
CMDSOFL	0 (1)	0 (1)	0 (0)
FUNCSOFL	0 (1)	0 (1)	0 (0)
NOSTKFL	0 (1)	0 (1)	0 (0)
NOREADFL	0 (1)	0 (1)	0 (0)
NOWRTFL	0 (1)	0 (1)	0 (0)
NEWSTKFL	0 (1)	0 (1)	1 (1)
USERPKFL	0 (1)	0 (1)	0 (0)
LOCPKFL	0 (1)	0 (1)	0 (0)
SYSPKFL	0 (1)	0 (1)	0 (0)
NEWSCFL	0 (1)	0 (1)	0 (0)
CLOSEXFL	0 (1)	0 (1)	0 (0)
NOESTAE	0 (1)	0 (1)	0 (0)
RENTRANT	0 (1)	0 (1)	0 (0)
NOPMSGS	0 (1)	0 (1)	0 (0)

Table 62. Values TSO/E provides in the three default parameters modules (continued)

Field name	IRXPARMS (MVS)	IRXTSPRM (TSO/E)	IRXISPRM (ISPF)
ALTMSGS	1 (1)	1 (1)	0 (0)
SPSHARE	0 (1)	1 (1)	1 (1)
STORFL	0 (1)	0 (1)	0 (0)
NOLOADDD	0 (1)	0 (1)	0 (0)
NOMSGWTO	0 (1)	0 (1)	0 (0)
NOMSGIO	0 (1)	0 (1)	0 (0)
ROSTORFL	0 (1)	0 (1)	0 (0)
SUBPOOL	0	78	78
ADDRSPN	MVS	TSO/E	ISPF
—	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF
Field Name in Module Name Table	**IRXPARMS (MVS)**	**IRXTSPRM (TSO/E)**	**IRXISPRM (ISPF)**
INDD	SYSTSIN	SYSTSIN	
OUTDD	SYSTSPRT	SYSTSPRT	
LOADDD	SYSEXEC	SYSEXEC	
IOROUT			
EXROUT			
GETFREER			
EXECINIT			
ATTNROUT			
STACKRT			
IRXEXECX			
IDROUT			
MSGIDRT			
EXECTERM			
—	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF
Field Name in Host Command Environment Table	**IRXPARMS (MVS)**	**IRXTSPRM (TSO/E)**	**IRXISPRM (ISPF)**
TOTAL	9	11	13
USED	9	11	13
LENGTH	32	32	32
INITIAL	MVS	TSO	TSO
—	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF
Entry 1			
NAME	MVS	MVS	MVS
ROUTINE	IRXSTAM	IRXSTAM	IRXSTAM
TOKEN			
Entry 2			
NAME	LINK	TSO	TSO
ROUTINE	IRXSTAM	IRXSTAM	IRXSTAM

Table 62. Values TSO/E provides in the three default parameters modules (continued)

Field name	IRXPARMS (MVS)	IRXTSPRM (TSO/E)	IRXISPRM (ISPF)
TOKEN			
Entry 3			
NAME	ATTACH	LINK	LINK
ROUTINE	IRXSTAM	IRXSTAM	IRXSTAM
TOKEN			
Entry 4			
NAME	CPICOMM	ATTACH	ATTACH
ROUTINE	IRXAPPC	IRXSTAM	IRXSTAM
TOKEN			
Entry 5			
NAME	LU62	CONSOLE	ISPEXEC
ROUTINE	IRXAPPC	IRXSTAM	IRXSTAM
TOKEN			
Field Name in Host Command Environment Table	**IRXPARMS (MVS)**	**IRXTSPRM (TSO/E)**	**IRXISPRM (ISPF)**
Entry 6			
NAME	LINKMVS	CPICOMM	ISREDIT
ROUTINE	IRXSTAMP	IRXAPPC	IRXSTAM
TOKEN			
Entry 7			
NAME	LINKPGM	LU62	CONSOLE
ROUTINE	IRXSTAMP	IRXAPPC	IRXSTAM
TOKEN			
Entry 8			
NAME	ATTCHMVS	LINKMVS	CPICOMM
ROUTINE	IRXSTAMP	IRXSTAMP	IRXAPPC
TOKEN			
Entry 9			
NAME	ATTCHPGM	LINKPGM	LU62
ROUTINE	IRXSTAMP	IRXSTAMP	IRXAPPC
TOKEN			
Entry 10			
NAME		ATTCHMVS	LINKMVS
ROUTINE		IRXSTAMP	IRXSTAMP
TOKEN			
Entry 11			
NAME		ATTCHPGM	LINKPGM
ROUTINE		IRXSTAMP	IRXSTAMP
TOKEN			

Table 62. Values TSO/E provides in the three default parameters modules (continued)

Field name	IRXPARMS (MVS)	IRXTSPRM (TSO/E)	IRXISPRM (ISPF)
Entry 12			
NAME			ATTCHMVS
ROUTINE			IRXSTAMP
TOKEN			
Entry 13			
NAME			ATTCHPGM
ROUTINE			IRXSTAMP
TOKEN			
Field Name in Function Package Table	**IRXPARMS (MVS)**	**IRXTSPRM (TSO/E)**	**IRXISPRM (ISPF)**
USER_TOTAL	1	1	1
USER_USED	1	1	1
LOCAL_TOTAL	1	1	1
LOCAL_USED	1	1	1
SYSTEM_TOTAL	1	2	2
SYSTEM_USED	1	2	2
LENGTH	8	8	8
—	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFF
Entry 1			
NAME	IRXEFMVS	IRXEFMVS	IRXEFMVS
Entry 2			
NAME	IRXFLOC	IRXEFPCK	IRXEFPCK
Entry 3			
NAME	IRXFUSER	IRXFLOC	IRXFLOC
Entry 4			
NAME		IRXFUSER	IRXFUSER

How IRXINIT determines what values to use for the environment

When the system calls IRXINIT to automatically initialize a language processor environment, IRXINIT must first determine what values to use for the environment. IRXINIT uses the values that are defined in one of the three default parameters modules that TSO/E provides and the values that are defined for the previous language processor environment.

IRXINIT always identifies a previous language processor environment. If an environment has not been initialized in the address space, IRXINIT uses the values in the default parameters module IRXPARMS as the previous environment. The following topics describe how IRXINIT determines the values for a new environment when the system calls IRXINIT to automatically initialize an environment in the TSO/E and non-TSO/E address spaces. “Chains of environments and how environments are located” on page 340 describes how any TSO/E REXX routine locates a previous environment.

Result: If you call IRXINIT to initialize an environment, IRXINIT evaluates the parameters you pass on the call and the parameters defined for the previous environment. “Initialization routine - IRXINIT” on page 371 describes how IRXINIT determines what values to use when a user explicitly calls the IRXINIT routine.

Values IRXINIT uses to initialize environments

When the system calls IRXINIT to automatically initialize an environment in the TSO/E address space, IRXINIT determines what values to use for defining the environment from two sources:

- The default parameters module IRXTSPRM or IRXISPRM
- The previous environment

During logon processing, IRXINIT initializes a language processor environment for the TSO/E session. IRXINIT first checks the values in the default parameters module IRXTSPRM. If the value is provided (that is, the value is not null), IRXINIT uses that value. If the value in the parameters module is null, IRXINIT uses the value from the previous environment. In this case, an environment does not exist, so IRXINIT uses the value from the IRXPARMS parameters module. IRXINIT computes each individual value using this method and then initializes the environment.

The following types of parameter values are considered to be null:

- A character string is null if it contains only blanks or has a length of zero
- An address is null if the address is 0 (or X'80000000', when specified for the user field parameter in the IRXINIT parameter list)
- A binary number is null if it has the value X'80000000'
- A bit setting is null if its corresponding mask is 0

For example, in IRXTSPRM, the PARSETOK field is null. When IRXINIT determines what value to use for PARSETOK, it finds a null field in IRXTSPRM. IRXINIT then checks the PARSETOK field in the previous environment. A previous environment does not exist, so IRXINIT takes the value from the IRXPARMS module. In this case, the PARSETOK field in IRXPARMS is null, which is the value that IRXINIT uses for the environment. If an exec running in the environment contains the PARSE SOURCE instruction, the last token that PARSE SOURCE returns is a question mark.

After IRXINIT determines all of the values, IRXINIT initializes the new environment.

When a user invokes ISPF from the TSO/E session, the system calls IRXINIT to initialize a new language processor environment for ISPF. IRXINIT first checks the values provided in the IRXISPRM parameters module. If a particular parameter has a null value, IRXINIT uses the value from the previous environment. In this case, the previous environment is the environment that IRXINIT initialized for the TSO/E session. For example, in the IRXISPRM parameters module, the mask bit (CMDSOFL_MASK) for the command search order flag (CMDSOFL) is 0. A mask of 0 indicates that the corresponding flag bit is null. Therefore, IRXINIT uses the flag setting from the previous environment, which in this case is 0.

As the previous descriptions show, the parameters defined in all three parameters modules can have an effect on any language processor environment that is initialized in the address space.

When IRXINIT automatically initializes a language processor environment in a non-TSO/E address space, IRXINIT uses the values in the parameters module IRXPARMS only.

If you call the IRXINIT routine to initialize a language processor environment, you can pass parameters on the call that define the values for the environment. See Chapter 15, “Initialization and termination routines,” on page 371 for information about IRXINIT.

Chains of environments and how environments are located

As described in previous topics, many language processor environments can be initialized in one address space. A language processor environments is associated with an MVS task. There can be several language processor environments associated with a single task. This topic describes how non-reentrant environments are chained together in an address space.

Language processor environmentss are chained together in a hierarchical structure to form a *chain of environments*. The environments on one chain are interrelated and share system resources. For example, several language processor environments can share the same data stack. However, separate chains within a single address space are independent.

Although many language processor environments can be associated with a single MVS task, each individual environment is associated with only one task.

Figure 22 on page 341 illustrates three language processor environments that form one chain.

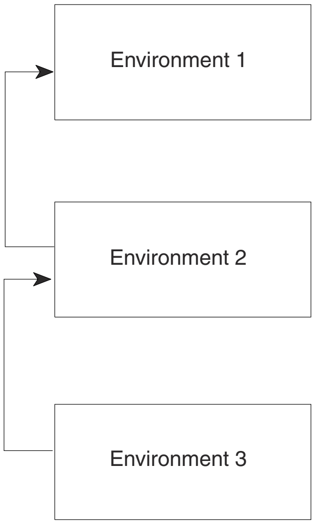


Figure 22. Three Language Processor Environments in a chain

The first environment initialized was environment 1. When IRXINIT initializes the second environment, the first environment is considered to be the previous environment (the parent environment). Environment 2 is chained to environment 1. Similarly, when IRXINIT initializes the third environment, environment 2 is considered to be the previous environment. Environment 2 is the parent environment for environment 3.

Different chains can exist in one address space. Figure 23 on page 341 illustrates two separate tasks, task 1 and task 2. Each task has a chain of environments. For task 1, the chain consists of two language processor environments. For task 2, the chain has only one language processor environment. The two environments on task 1 are interrelated and share system resources. The two chains are completely separate and independent.

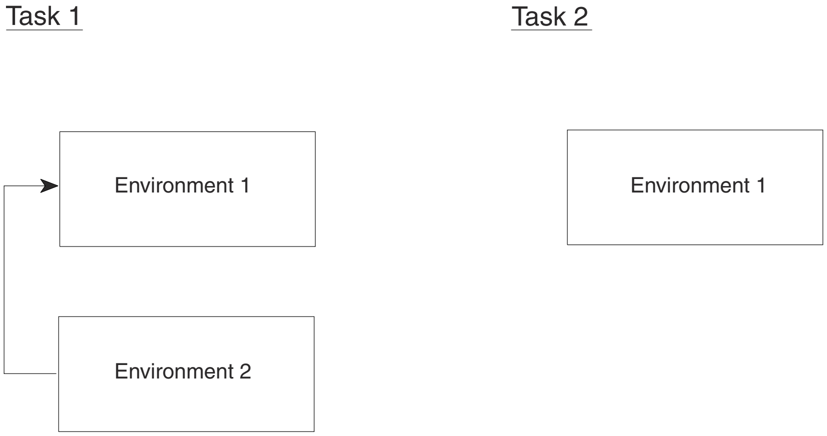


Figure 23. Separate chains on two different tasks

As discussed previously, language processor environments are associated with an MVS task. Under an MVS task, IRXINIT can initialize one or more language processor environments. The task can then attach another task. IRXINIT can be called under the second task to initialize a language processor environment. The new environment is chained to the last environment under the first task. Figure 24 on page 342 illustrates a task that has attached another task and how the language processor environments are chained together.

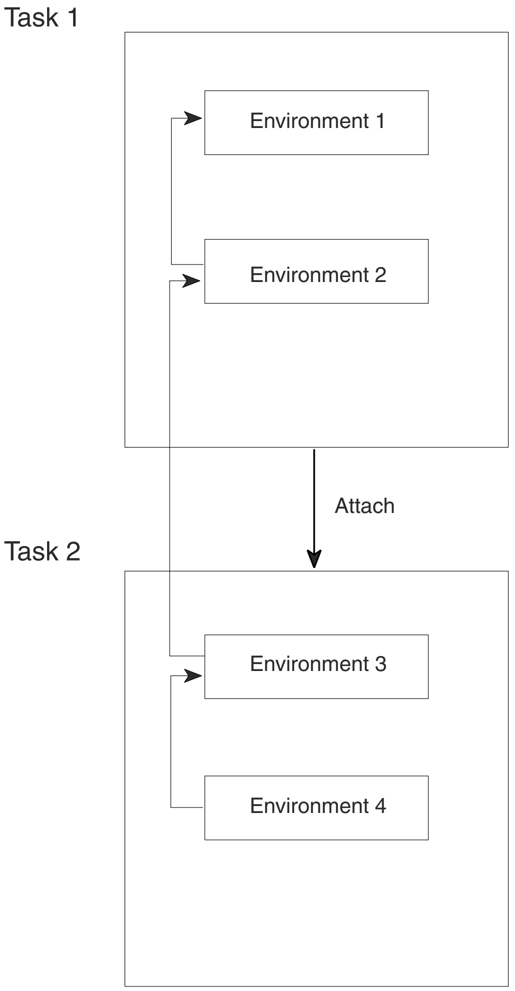


Figure 24. One chain of environments for attached tasks

As shown in Figure 24 on page 342, task 1 is started and IRXINIT initializes an environment (environment 1). IRXINIT is invoked again to initialize a second language processor environment under task 1 (environment 2). Environment 2 is chained to environment 1. If you invoke a REXX exec within task 1, the exec runs in environment 2.

Task 1 then attaches another task, task 2. IRXINIT is called to initialize an environment. IRXINIT locates the previous environment, which is environment 2, and chains the new environment (environment 3) to its parent (environment 2). When IRXINIT is called again, IRXINIT chains the fourth environment (environment 4) to its parent (environment 3). At this point, four language processor environments exist on the chain.

Locating a Language Processor Environment

A REXX environment is considered integrated into TSO/E if the address space is a TSO/E address space and the environment has the TSOFL flag set on. Otherwise the environment is not integrated into TSO/E (TSOFL flag off). For the following discussion, a REXX environment that is integrated into TSO/E will be referred to as a "TSO/E environment"; and a REXX environment that is not integrated into TSO/E will be referred to as a "non-TSO/E environment." For more information, see “Types of environments - integrated and not integrated into TSO/E” on page 316.

Whenever you invoke a REXX exec or routine, the exec or routine must run in a language processor environment. The one exception is the initialization routine, IRXINIT, which initializes environments.

In the TSO/E address space, the system always initializes a default language processor environment when you log on TSO/E and when you invoke ISPF. If you call a REXX programming routine from TSO/E, the routine runs in the environment in which you called it.

If you invoke an exec using the IRXJCL or IRXEXEC routine, a language processor environment may or may not already exist. If an environment does not exist on the

- Current task (non-TSO/E environment), or
- Current task or a parent task (TSO/E environment)

the system calls the IRXINIT routine to initialize an environment before the exec runs. Otherwise, the system locates the current non-reentrant environment and the exec runs in that environment.

IRXINIT always locates a previous language processor environment. If an environment does not exist on the current task (TSO/E or non-TSO/E environment) or on a parent task (TSO/E environment only), IRXINIT uses the values in the IRXPARMS parameters module as the previous environment.

A language processor environment must already exist if you call the TSO/E REXX programming routines IRXRLT, IRXSUBCM, IRXIC, IRXEXCOM, and IKJCT441 or the replaceable routines. These routines do not invoke IRXINIT to initialize a new environment. If an environment does not already exist and you call one of these routines, the routine completes unsuccessfully with a return code. See Chapter 12, “TSO/E REXX programming services,” on page 239 for information about the TSO/E REXX programming routines and Chapter 16, “Replaceable routines and exits,” on page 389 for information about the replaceable routines.

When IRXINIT initializes a new language processor environment, IRXINIT creates a number of control blocks that contain information about the environment and any REXX exec currently running in the environment. The main control block is the environment block (ENVBLOCK), which points to other control blocks, such as the parameter block (PARMBLOCK) and the work block extension. “Control blocks created for a Language Processor Environment” on page 357 describes the control blocks that IRXINIT creates for each language processor environment.

The environment block represents its language processor environment and is the anchor that the system uses on calls to all REXX programming service routines. Whenever you call a REXX programming service routine, you can pass the address of an environment block in register 0 on the call. By passing the address, you can specify in which language processor environment you want the routine to run. For example, suppose you invoke the initialization routine, IRXINIT, in a non-TSO/E address space. On return, IRXINIT returns the address of the environment block for the new environment in register 0. You can store that address for future use. Suppose you call IRXINIT several times to initialize a total of four environments in that address space. If you then want to call a TSO/E REXX programming service routine and have the routine run in the first environment on the chain, you can pass the address of the first environment's environment block on the call.

You can also pass the address of the environment block in register 0 to all REXX replaceable routines and exit routines. Some service routines also allow the address of the environment block to be passed as a parameter, in place of using register 0. If the parameter is used in these cases, register 0 is ignored.

When a programming service routine is called, the programming service routine must determine in which environment to run. The routine locates the environment as follows:

1. The routine checks for an environment block parameter (where allowed) or register 0 to determine whether the address of an environment block was passed on the call. If an address was passed, the routine determines whether the address points to a valid environment block. The environment block is valid if:

   - The environment is either a reentrant or non-reentrant environment on the current task (non-TSO/E environment)
   - The environment is either a reentrant or non-reentrant environment on the current task or on a parent task (TSO/E environment).

2. If the address of a valid environment block was not passed, the routine that is called:

   - Searches for a non-reentrant environment on the current task (non-TSO/E environment). If a non-reentrant environment is found, the routine runs in that environment.
   - Searches for a non-reentrant environment on the current task (TSO/E environment). If a non-reentrant environment is found on the current task, the routine runs in that environment. If a non-reentrant environment is not found on the current task, the parent tasks are searched for a

non-reentrant environment. If a non-reentrant TSO/E environment is found on any of these ancestor tasks, the routine runs in that environment.

3. If the routine could not find a valid environment using the previous steps, the next step depends on what routine was called.

   - If one of the REXX programming routines or the replaceable routines was called, a language processor environment is required in order for the routine to run. The routine ends in error. The same occurs for the termination routine, IRXTERM.
   - If IRXEXEC or IRXJCL were called, the routine invokes IRXINIT to initialize a new environment.
   - If IRXINIT was called, IRXINIT uses the IRXPARMS parameters module as the previous environment.

The IRXINIT routine initializes a new language processor environment. Therefore, IRXINIT does not need to locate an environment in which to run. However, IRXINIT does locate a previous environment to determine what values to use when defining the new environment. The following summarizes the steps IRXINIT takes to locate the previous environment:

1. If register 0 contains the address of a valid environment block, IRXINIT uses that environment as the previous environment.
2. If IRXINIT has not yet found an environment to use and a non-reentrant environment exists on the current task, IRXINIT uses the last non-reentrant environment on the task as the previous environment.
3. If IRXINIT has not yet found an environment to use and this is a request for a TSO/E environment, IRXINIT locates the last non-reentrant environment on the parent tasks. If a non-reentrant TSO/E environment is located on any of the parent tasks, IRXINIT uses it as the previous environment.
4. If IRXINIT cannot find an environment, IRXINIT uses the values in the default parameters module IRXPARMS as the previous environment.

"Initialization routine - IRXINIT" on page 371 describes how the IRXINIT routine determines what values to use when you explicitly call IRXINIT.

Changing the default values for initializing an environment

TSO/E provides default values in three parameters modules (load modules) for initializing language processor environments in non-TSO/E, TSO/E, and ISPF. In most cases, your installation probably need not change the default values. However, if you want to change one or more parameter values, you can provide your own load module that contains your values.

You can also call the initialization routine, IRXINIT, to initialize a new environment. On the call, you can pass the parameters whose values you want to be different from the previous environment. If you do not specifically pass a parameter, IRXINIT uses the value defined in the previous environment. See "Initialization routine - IRXINIT" on page 371 for more information.

This topic describes how to create a load module containing parameter values for initializing an environment. You should also refer to "Characteristics of a Language Processor Environment" on page 317 for information about the format of the parameters module.

To change one or more default values that IRXINIT uses to initialize a language processor environment, you can provide a load module containing the values you want. You must first write the code for a parameters module. TSO/E provides three samples in SYS1.SAMPLIB that are assembler code for the default parameters modules. The member names of the samples are:

- IRXREXX1 (for IRXPARMS — MVS)
- IRXREXX2 (for IRXTSPRM — TSO/E)
- IRXREXX3 (for IRXISPRM — ISPF)

When you write the code, be sure to include the correct default values for any parameters you are not changing. For example, suppose you are adding several function packages to the IRXISPRM module for ISPF. In addition to coding the function package table, you must also provide all of the other fields in the parameters module and their default values. “Values provided in the three default parameters modules” on page 336 shows the default parameter values for IRXPARMS, IRXTSPRM, and IRXISPRM.

After you create the code, you must assemble the code and then link- edit the object code. The output is a member of a partitioned data set. The member name must be either IRXPARMS, IRXTSPRM, or IRXISPRM depending on the load module you are providing. You must then place the data set with the IRXPARMS, IRXTSPRM, or IRXISPRM member in the search sequence for an MVS LOAD macro. The parameters modules that TSO/E provides are in the LPALIB, so you could place your data set in a logon STEPLIB, a JOBLIB, or in linklist.

If you provide an IRXPARMS load module, your module may contain parameter values that cannot be used in language processor environments that are integrated into TSO/E. When IRXINIT initializes an environment for TSO/E, IRXINIT uses the IRXTSPRM parameters module. However, if a parameter value in IRXTSPRM is null, IRXINIT uses the value from the IRXPARMS module. Therefore, if you provide your own IRXPARMS load module that contains parameters that cannot be used in TSO/E, you must place the data set in either a STEPLIB or JOBLIB that is not searched by the TSO/E session. For more information about the values you can specify for different types of environments, see “Specifying values for different environments” on page 348.

The new values you specify in your own load module are not available until the current language processor environment is terminated and a new environment is initialized. For example, if you provide a load module for TSO/E (IRXTSPRM), you must log on TSO/E again.

Providing your own parameters modules

There are various considerations for providing your own parameters modules. The different considerations depend on whether you want to change a parameter value only for an environment that is initialized for ISPF, for environments that are initialized for both the TSO/E and ISPF sessions, or for environments that are initialized in a non-TSO/E address space. The following topics describe changing the IRXISPRM, IRXTSPRM, and IRXPARMS values.

TSO/E provides the following samples in SYS1.SAMPLIB that you can use to code your own load modules:

- IRXREXX1 (for IRXPARMS — MVS)
- IRXREXX2 (for IRXTSPRM — TSO/E)
- IRXREXX3 (for IRXISPRM — ISPF)

Changing values for ISPF

If you want to change a default parameter value for language processor environments that are initialized for ISPF, you should provide your own IRXISPRM module. IRXINIT only locates the IRXISPRM load module when IRXINIT is initializing a language processor environment for ISPF. IRXINIT does not use IRXISPRM when initializing an environment for either a TSO/E session or for a non-TSO/E address space.

When you create the code for the load module, you must specify the new values you want for the parameters you are changing and the default values for all of the other fields. “Values provided in the three default parameters modules” on page 336 shows the defaults that TSO/E provides in the IRXISPRM parameters module.

After you assemble and link-edit the code, place the data set with the IRXISPRM member in the search sequence for an MVS LOAD. For example, you can put the data set in a logon STEPLIB or linklist. The new values are not available until IRXINIT initializes a new language processor environment for ISPF. For example, if you are currently using ISPF, you must return to TSO/E READY mode and then invoke ISPF again. When the system calls IRXINIT to initialize an environment for ISPF, IRXINIT locates your load module and initializes the environment using your values.

There are many fields in the parameters module that are intended for use only if an environment is not being integrated into TSO/E. There are also several flag settings that you must not change in the IRXISPRM parameters module for ISPF. See “Specifying values for different environments” on page 348 for information about which fields you can and cannot specify.

Changing values for TSO/E

If you want to change a default parameter value for environments that IRXINIT initializes for TSO/E only, you probably have to code both a new IRXTSPRM module (for TSO/E) and a new IRXISPRM module (for ISPF). This is because most of the fields in the default IRXISPRM parameters module are null, which means that IRXINIT uses the value from the previous environment. The previous environment is the one that IRXINIT initializes for the TSO/E session.

For example, in the default IRXTSPRM module (for TSO/E), the USERPKFL, LOCPKFL and SYSPKFL flags are 0. This means the user, local, and system function packages defined for the previous environment are also available to the environment IRXINIT initializes for the TSO/E session. In the default IRXISPRM module (for ISPF), the masks for these three flags are 0, which means IRXINIT uses the flag settings from the previous environment. IRXINIT initialized the previous environment (TSO/E) using the IRXTSPRM module. Suppose you do not want the function packages from the previous environment available to an environment that IRXINIT initializes for TSO/E. However, when IRXINIT initializes an environment for ISPF, the function packages defined for the TSO/E environment should also be available in ISPF. You must code a new IRXTSPRM module and specify a setting of 1 for the USERPKFL, LOCPKFL, and SYSPKFL flags. You must code a new IRXISPRM module and specify a setting of 1 for the following mask fields:

- USERPKFL_MASK
- LOCPKFL_MASK
- SYSPKFL_MASK

When you code the new load modules, you must include the default values for all of the other parameters. "Values provided in the three default parameters modules" on page 336 shows the defaults TSO/E provides.

Changing values for TSO/E and ISPF

If you want to change a default parameter value for language processor environments that IRXINIT initializes for TSO/E and ISPF, you may be able to simply provide your own IRXTSPRM module for TSO/E and use the default IRXISPRM module for ISPF. Whether you need to create one or two parameters modules depends on the specific parameter value you want to change and whether that field is null in the IRXISPRM default module. If the field is null in IRXISPRM, when IRXINIT initializes a language processor environment for ISPF, IRXINIT uses the value from the previous environment (TSO/E), which is the value in the IRXTSPRM module.

For example, suppose you want to change the setting of the NOLOADDD flag so that the system searches SYSPROC only when you invoke an exec. The value in the default IRXTSPRM (TSO/E) module is 0, which means the system searches SYSEXEC followed by SYSPROC. In the default IRXISPRM (ISPF) module, the mask for the NOLOADDD flag is 0, which means IRXINIT uses the value defined in the previous environment. You can code a IRXTSPRM load module and specify 1 for the NOLOADDD flag. You do not need to create a new IRXISPRM module. When IRXINIT initializes a language processor environment for ISPF, IRXINIT uses the value from the previous environment.

You may need to code two parameters modules for IRXTSPRM and IRXISPRM depending on the parameter you want to change and the default value in IRXISPRM. For example, suppose you want to change the language code. You must code two modules because the value in both default modules is ENU. Code a new IRXTSPRM module and specify the language code you want. Code a new IRXISPRM module and specify either a null or the specific language code. If you specify a null, IRXINIT uses the language code from the previous environment, which is TSO/E.

You also need to code both an IRXTSPRM and IRXISPRM load module if you want different values for TSO/E and ISPF.

If you provide your own load modules, you must also include the default values for all of the other fields as provided in the default modules. "Values provided in the three default parameters modules" on page 336 shows the defaults provided in IRXTSPRM and IRXISPRM.

After you assemble and link-edit the code, place the data set with the IRXTSPRM member (and IRXISPRM member if you coded both modules) in the search sequence for an MVS LOAD. For example, you can put the data sets in a logon STEPLIB or linklist. The new values are not available until IRXINIT initializes a new language processor environment for TSO/E and for ISPF. You must log on TSO/E again. During logon, IRXINIT uses your IRXTSPRM load module to initialize the environment. Similarly, IRXINIT uses your IRXISPRM module when you invoke ISPF.

There are many fields in the parameters module that you must not change for certain parameters modules. See “Specifying values for different environments” on page 348 for information about the values you can specify.

Changing values for Non-TSO/E

If you want to change a default parameter value for language processor environments that IRXINIT initializes in non-TSO/E address spaces, code a new IRXPARMS module. In the code, you must specify the new values you want for the parameters you are changing and the default values for all of the other fields. “Values provided in the three default parameters modules” on page 336 shows the defaults TSO/E provides in the IRXPARMS parameters module.

There are many fields in the parameters module that are intended for use in language processor environments that are *not* integrated into TSO/E. If you provide IRXPARMS with values that cannot be used in TSO/E, provide the IRXPARMS module only for non-TSO/E address spaces. When you assemble the code and link-edit the object code, you must name the output member IRXPARMS. You must then place the data set with IRXPARMS in either a STEPLIB or JOBLIB that is not searched by the TSO/E session. You can do this using JCL. You must ensure that the data set is not searched by the TSO/E session.

If you provide your own IRXPARMS module that contains parameters values that must not be used by environments that are integrated into TSO/E (for example, TSO/E and ISPF), and IRXINIT locates the module when initializing a language processor environment in the TSO/E address space, IRXINIT may terminate or errors may occur when TSO/E users log on TSO/E or invoke ISPF. For example, you can provide your own replaceable routines only in language processor environments that are not integrated into TSO/E. The values for the replaceable routines in the three default parameters modules are null. You can code your own IRXPARMS load module and specify the names of one or more replaceable routines. However, your module must not be in the TSO/E search order. When IRXINIT is invoked to initialize a language processor environment for TSO/E, IRXINIT finds a null value for the replaceable routine in the IRXTSPRM parameters module. IRXINIT then uses the value from the previous environment, which, in this case, is the value in IRXPARMS.

In the TSO/E address space, you can call IRXINIT and initialize an environment that is not integrated into TSO/E. See “Types of environments - integrated and not integrated into TSO/E” on page 316 about the two types of environments.

For more information about the parameters you can use in different language processor environments, see “Specifying values for different environments” on page 348.

Considerations for providing parameters modules

The previous topics describe how to change the default parameter values that IRXINIT uses to initialize a language processor environment. You can provide your own IRXISPRM, IRXTSPRM, and IRXPARMS modules for ISPF, TSO/E, and non-TSO/E. Generally, if you want to change environment values for REXX execs that run from ISPF, you can simply provide your own IRXISPRM parameters module. To change values for TSO/E only or for TSO/E and ISPF, you may have to create only a IRXTSPRM module or both the IRXTSPRM and IRXISPRM modules. The modules you have to provide depend on the parameter you are changing and the value in the IRXISPRM default module.

If you provide an IRXPARMS module and your module contains parameter values that cannot be used in environments that are integrated into TSO/E, you must ensure that the module is available only to non-TSO/E address spaces, not to TSO/E and ISPF.

Before you code your own parameters module, review the default values that TSO/E provides. In your code, you must include the default values for any parameters you are not changing. In the ISPF module IRXISPRM, many parameter values are null, which means IRXINIT obtains the value from the previous environment. In this case, the previous environment was defined using the IRXTSPRM values. If you provide a IRXTSPRM module for TSO/E, check how the module affects the definition of environments for ISPF.

TSO/E provides three samples in SYS1.SAMPLIB that are assembler code samples for the three parameters modules. The member names of the samples are:

- IRXREXX1 (for IRXPARMS — MVS)
- IRXREXX2 (for IRXTSPRM — TSO/E)
- IRXREXX3 (for IRXISPRM — ISPF)

Specifying values for different environments

As described in the previous topic ("Changing the default values for initializing an environment" on page 344), you can change the default parameter values IRXINIT uses to initialize a language processor environment by providing your own parameters modules. You can also call the initialization routine, IRXINIT, to initialize a new environment. When you call IRXINIT, you can pass parameter values on the call. Chapter 15, "Initialization and termination routines," on page 371 describes IRXINIT and its parameters and return codes.

Whether you provide your own load modules or invoke IRXINIT directly, you cannot change some parameters. You can use other parameters only in language processor environments that are not integrated into TSO/E or in environments that are integrated into TSO/E. In addition, there are some restrictions on parameter values depending on the values of other parameters in the same environment and on parameter values that are defined for the previous environment. This topic describes the parameters you can and cannot use in the two types of language processor environments. The topic also describes different considerations for using the parameters. For more information about the parameters and their descriptions, see "Characteristics of a Language Processor Environment" on page 317.

Parameters you cannot change

There are two parameters that have fixed values and that you cannot change. The parameters are:

ID
The value must be IRXPARMS. If you provide your own load module, you must specify IRXPARMS for the ID. If you call IRXINIT, IRXINIT ignores any value you pass and uses the default IRXPARMS.

VERSION
The value must be 0200. If you provide your own load module or call IRXINIT, specify 0200 for the version.

Parameters you can use in any Language Processor Environment

There are several parameters that you can specify in any language processor environment. That is, you can use these parameters in environments that are integrated into TSO/E and in environments that are not integrated into TSO/E. The following describes the parameters and any considerations for specifying them.

LANGUAGE
The language code. The default is ENU for US English in mixed case (upper and lowercase).

PARSETOK
The token for the PARSE SOURCE instruction. The default is a blank.

ADDRSPN
The name of the address space. TSO/E provides the following defaults:

- IRXPARMS – MVS
- IRXTSPRM – TSO/E
- IRXISPRM – ISPF

You can change the address space name for any type of language processor environment. If you write applications that examine the PARMBLOCK for an environment and perform processing based on the address space name, you must ensure that any changes you make to the ADDRSPN field do not affect your application programs.

FLAGS

The FLAGS field is a fullword of bits that are used as flags. You can specify any of the flags in any environment. However, the value you specify for each flag depends on the purpose of the flag. In addition, there are some restrictions for various flag settings depending on the flag setting in the previous environment.

The following explains the different considerations for the setting of some flags. See "Characteristics of a Language Processor Environment" on page 317 for details about each flag.

Note: If your installation uses ISPF, there are several considerations about the flag settings for language processor environments that are initialized for ISPF. For more information, see "Flag settings for environments initialized for TSO/E and ISPF" on page 353.

TSOFL

The TSOFL flag indicates whether the new environment is integrated into TSO/E.

If IRXINIT is initializing an environment in a non-TSO/E address space, the flag must be off (set to 0). The TSOFL flag must also be off if the environment is being initialized as a reentrant environment. You can initialize reentrant environments only by explicitly calling the IRXINIT routine.

If IRXINIT is initializing an environment in the TSO/E address space, the TSOFL flag can be on or off. If the flag is on, the environment is integrated into TSO/E. REXX execs that run in the environment can use TSO/E commands, such as ALLOCATE and PRINTDS, and TSO/E programming services that are described in *z/OS TSO/E Programming Services* (for example, the parse service routine and TSO/E I/O service routines, such as PUTGET). The exec can also use ISPF services and can call and be called by TSO/E CLISTs.

If the flag is off, the environment is not integrated into TSO/E. In this case, REXX execs cannot use TSO/E commands, TSO/E programming services, or ISPF services, or interact with CLISTs. If the exec contains these type of services, unpredictable results can occur.

If the TSOFL flag is on (the environment is integrated into TSO/E), then:

- The RENTRANT flag must be off (set to 0)
- The names of the replaceable routines in the module name table must be blank. You cannot provide replaceable routines in environments that are integrated into TSO/E.

  Note that the module name table also includes several fields for the names of REXX exit routines (for example, EXECINIT, ATTNROUT, IRXEXECX, and EXECTERM). If the environment is integrated into TSO/E (TSOFL flag is on), you can specify the exits in the module name table.
- The INDD and OUTDD fields in the module name table must be the defaults SYSTSIN and SYSTSPRT
- The subpool number in the SUBPOOL field must be 78, in decimal.

The TSOFL flag cannot be on (set to 1) if a previous language processor environment in the environment chain has the TSOFL flag off.

NEWSTKFL

The NEWSTKFL flag indicates whether IRXINIT initializes a new data stack for the new environment.

If you set the NEWSTKFL off for the new environment that IRXINIT is initializing, you must ensure that the SPSHARE flag is on in the previous environment. The SPSHARE flag determines whether the subpool is shared across MVS tasks. If the NEWSTKFL flag is off for the new environment and the SPSHARE flag is off in the previous environment, an error occurs when IRXINIT tries to initialize the new environment.

Module Name Table

The module name table contains the ddnames for reading and writing data and for loading REXX execs, and the names of replaceable routines and exit routines. The fields you can specify in any address space are described below. You can use the replaceable routines only in:

- Non-TSO/E address spaces
- The TSO/E address space if the language processor environment is initialized with the TSOFL flag off (the environment is not integrated with TSO/E).

The module name table also contains fields for several REXX exits. The fields are EXECINIT for the exec initialization exit, ATTNROUT for the attention handling exit, IRXEXECX for the exec processing exit (for the IRXEXEC routine), and EXECTERM for the exec termination exit. You can specify exits for exec initialization (EXECINIT), exec processing (IRXEXECX), and exec termination (EXECTERM) in any type of language processor environment. You can provide an attention handling exit (ATTNROUT) only for environments that are integrated into TSO/E.

LOADDD

The name of the DD from which the system loads REXX execs. The default TSO/E provides in all three parameters modules is SYSEXEC. (See "Using SYSPROC and SYSEXEC for REXX execs" on page 353 for more information about SYSEXEC in the TSO/E address space).

The DD from which the system loads REXX execs depends on the name specified in the LOADDD field and the setting of the TSOFL and NOLOADDD flags. If the TSOFL flag is on, the language processor environment is initialized in the TSO/E address space and is integrated into TSO/E (see "Flags and corresponding masks" on page 322). In TSO/E, you can store REXX execs in data sets that are allocated to SYSPROC or to the DD specified in the LOADDD field (the default is SYSEXEC). The NOLOADDD flag (see "Flags and corresponding masks" on page 322) indicates whether the system searches SYSPROC only or whether the system searches the DD specified in the LOADDD field (SYSEXEC) first, followed by SYSPROC.

If the TSOFL flag is off, the system loads REXX execs from the DD specified in the LOADDD field.

Note: For the default parameters modules IRXTSPRM and IRXISPRM, the NOLOADDD flag is off (0). Therefore, the system searches SYSEXEC followed by SYSPROC. To have the system search SYSPROC exclusively, you can provide your own parameters module. TSO/E users can also use the EXECUTIL command to dynamically change the search order. "EXECUTIL" on page 216 describes the EXECUTIL command.

The system opens the specified DD the first time a REXX exec is loaded. The DD remains open until the environment under which it was opened is terminated. If you want the system to close the DD after each REXX exec is fetched, you must set the CLOSEXFL flag on (see "Flags and corresponding masks" on page 322). Users can also use the EXECUTIL command to dynamically close the DD. Note that the system may close the data set at certain points.

See "Using SYSPROC and SYSEXEC for REXX execs" on page 353 for more information about SYSPROC and SYSEXEC.

EXECINIT

The name of an exit routine that gets control after the system initializes the REXX variable pool for a REXX exec, but before the language processor starts processing the exec.

IRXEXECX

The name of an exit routine that is invoked whenever the IRXEXEC routine is called.

EXECTERM

The name of an exit routine that is invoked after a REXX exec has completed processing, but before the system terminates the REXX variable pool.

Host Command Environment Table

The table contains the names of the host command environments that are valid for the language processor environment and the names of the routines that the system calls to process commands for the host command environment.

When IRXINIT creates the host command environment table for a new language processor environment, IRXINIT checks the setting of the NEWSCFL flag. The NEWSCFL flag indicates whether the host command environments that are defined for the previous language processor environment are added to the table that is specified for the new environment. If the NEWSCFL flag is 0, IRXINIT creates the table by copying the host command environment table from the previous environment and concatenating the entries specified for the new environment. If the NEWSCFL flag is 1, IRXINIT creates the table using only the entries specified for the new environment.

Function Package Table

The function package table contains information about the user, local, and system function packages that are available in the language processor environment. "Function package table" on page 333 describes the format of the table in detail.

When IRXINIT creates the function package table for a new language processor environment, IRXINIT checks the settings of the USERPKFL, LOCPKFL, and SYSPKFL flags. The three flags indicate whether the user, local, and system function packages that are defined for the previous language processor environment are added to the function package table that is specified for the new environment. If a particular flag is 0, IRXINIT copies the function package table from the previous environment and concatenates the entries specified for the new environment. If the flag is 1, IRXINIT creates the function package table using only the entries specified for the new environment.

Parameters you can use for environments that are integrated into TSO/E

There is one parameter that you can use only if a language processor environment is initialized in the TSO/E address space and the TSOFL flag is on. The parameter is the ATTNROUT field in the module name table. The ATTNROUT field specifies the name of an exit routine for attention processing. The exit gets control if a REXX exec is running in the TSO/E address space and an attention interruption occurs. "REXX exit routines" on page 426 describes the attention handling exit.

The ATTNROUT field must be blank if the new environment is not being integrated into TSO/E, that is, the TSOFL flag is off.

Parameters you can use for environments that are not integrated into TSO/E

There are several parameters that you can specify only if the environment is not integrated into TSO/E (the TSOFL flag is off). The following describes the parameters and any considerations for specifying them.

SUBPOOL

The subpool number in which storage is allocated for the entire language processor environment. In the parameters module IRXPARMS, the default is 0. You can specify a number from 0 – 127.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, the subpool number must be 78, in decimal.

Module Name Table

The module name table contains the names of DDs for reading and writing data and for loading REXX execs, and the names of replaceable routines and exit routines. The fields you can specify if the environment is not integrated into TSO/E (the TSOFL flag is off) are described below.

INDD

The name of the DD from which the PARSE EXTERNAL instruction reads input data. The default is SYSTSIN.

If IRXINIT initializes the environment in the TSO/E address space and the TSOFL flag is on, IRXINIT ignores the ddname.

If the specified DD is opened by a previous language processor environment, even an environment on a higher task, and the INDD value for the new environment is obtained from the previous environment, the new environment uses the DCB of the previous environment. Sharing of the DCB in this way means:

- A REXX exec running in the new environment reads the record that follows the record the previous environment read.
- If the previous environment runs on a higher task and that environment is terminated, the new environment reopens the DD. However, the original position in the DD is lost.

OUTDD

The name of the DD to which data is written for a SAY instruction, when tracing is started, or for REXX error messages. The default is SYSTSPRT.

If IRXINIT initializes the environment in the TSO/E address space and the TSOFL flag is on, IRXINIT ignores the ddname.

If you initialize two environments by calling IRXINIT and explicitly pass the same ddname for the two different environments, when the second environment opens the DD, the open fails. The open fails because the data set can only be opened once. The OPEN macro issues an ENQ exclusively for the ddname.

IOROUT

The name of the input/output (I/O) replaceable routine. “Input/Output routine” on page 403 describes the routine in detail.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, this field must be blank.

EXROUT

The name of the load exec replaceable routine. “Exec load routine” on page 392 describes the routine in detail.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, this field must be blank.

GETFREER

The name of the storage management replaceable routine. “Storage management routine” on page 420 describes the routine in detail.

If more than one language processor environment is initialized on the same task and the environments specify a storage management replaceable routine, the name of the routine must be the same. If the name of the routine is different for two environments on the same task, an error occurs when IRXINIT tries to initialize the new environment.

If the environment is initialized in the TSO/E address space and the TSOFL is on, the GETFREER field must be blank.

STACKRT

The name of the data stack replaceable routine. “Data stack routine” on page 415 describes the routine in detail.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, this field must be blank.

IDROUT

The name of the user ID replaceable routine. The system calls the routine whenever an exec uses the USERID built-in function. “User ID routine” on page 422 describes the routine in detail.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, this field must be blank.

MSGIDRT

The name of the message identifier replaceable routine. The system calls the routine to determine whether message IDs are displayed. “Message identifier routine” on page 425 describes the routine in detail.

If the environment is initialized in the TSO/E address space and the TSOFL flag is on, this field must be blank.

Flag settings for environments initialized for TSO/E and ISPF

If your installation uses ISPF, there are several considerations about flag settings for language processor environments that are initialized for TSO/E and ISPF. In the default IRXISPRM parameters module for ISPF, most of the mask settings for the flags parameters are 0, which means IRXINIT uses the values from TSO/E (IRXTSPRM module). If you provide your own IRXISPRM load module, you should not change the mask values for the following flags. The mask values for these flags should be 0.

- CMDSOFL — command search order flag
- FUNCSOFL — function and subroutine search order flag
- NOSTKFL — no data stack flag
- NOREADFL — no read (input file) flag
- NOWRTFL — no write (output file) flag
- NEWSTKFL — new data stack flag
- NOESTAE — recovery ESTAE flag
- RENTRANT — reentrant/non-reentrant flag
- SPSHARE — subpool sharing flag

The values for these flags in ISPF should be the same as the values that IRXINIT uses when initializing an environment for the TSO/E session. When IRXINIT initializes an environment for ISPF, IRXINIT uses the values defined for the previous environment (TSO/E) because the mask settings are 0. Using the same values for these flags for both TSO/E and ISPF prevents any processing problems between the ISPF and TSO/E sessions.

If you do want to change one of the flag values, change the value in the IRXTSPRM parameters module for TSO/E. The change is inherited by ISPF when IRXINIT initializes an environment for the ISPF screen. For example, suppose you want to change the search order the system uses for locating external functions and subroutines. The FUNCSOFL flag controls the search order. You can provide a IRXTSPRM parameters module for TSO/E and change the flag setting. ISPF inherits the changed flag setting when IRXINIT initializes an environment.

Using SYSPROC and SYSEXEC for REXX execs

In the module name table, the LOADDD field contains the name of the DD from which REXX execs are fetched. The default TSO/E provides for non-TSO/E, TSO/E, and ISPF is SYSEXEC. If you customize REXX processing either by providing your own parameters modules or explicitly calling IRXINIT to initialize an environment, it is suggested that you use the ddname SYSEXEC. The TSO/E REXX documentation refers to this DD as SYSEXEC. For a description of LOADDD field, see "Module name table" on page 326.

In TSO/E, you can store both interpreted and compiled REXX execs in data sets that are allocated to either SYSPROC or SYSEXEC. You can use SYSPROC for both TSO/E CLISTs and REXX execs. SYSEXEC is for REXX execs only. If an exec is in a data set that is allocated to SYSPROC, the exec must start with a comment containing the characters REXX within the first line (line 1). This is referred to as the REXX identifier and is required in order for the TSO/E EXEC command to distinguish REXX execs from CLISTs. *z/OS TSO/E REXX User's Guide* describes how to allocate execs to SYSPROC and SYSEXEC. For information about compiled execs, see the appropriate compiler publications.

In the parameters module, the NOLOADDD flag (see "Flags and corresponding masks" on page 322) controls the search order for REXX execs. The flag indicates whether the system searches the DD specified in the LOADDD field (SYSEXEC). With the defaults that TSO/E provides, the system searches SYSEXEC first, followed by SYSPROC. The system searches SYSPROC only if the language processor environment is integrated into TSO/E.

If your installation plans to use REXX, store your execs in data sets that are allocated to SYSEXEC, instead of using SYSPROC. Using SYSEXEC makes it easier to maintain your REXX execs. If your installation uses many CLISTs and does not plan to have a large number of REXX execs, you may want to use SYSPROC only and not use SYSEXEC. To use SYSPROC only, you might provide your own IRXTSPRM parameters module for TSO/E or use the EXECUTIL SEARCHDD command.

If you provide your own IRXTSPRM parameters module, specify the following values for the NOLOADDD mask and flag fields:

- NOLOADDD_MASK — 1
- NOLOADDD_FLAG — 1

With these values, the system does not search SYSEXEC and searches SYSPROC only. You can make your parameters module available on a system-wide basis for your entire installation. You can also make your module available only to a specific group of users by making it available only on a logon level. You can place your IRXTSPRM module in a data set specified in the STEPLIB concatenation in the logon procedure. You must ensure that the data set is higher in the concatenation than any other data set that contains IRXTSPRM. See *z/OS TSO/E Customization* for more information about logon procedures.

You need not provide your own IRXISPRM parameters module for ISPF because the NOLOADDD mask value in the default IRXISPRM module is 0, which means IRXINIT uses the flag setting from the previous environment. In this case, the previous environment is the value from the IRXTSPRM module you provide.

You can also use the EXECUTIL command with the SEARCHDD operand to change the search order and have the system search SYSPROC only. You can use EXECUTIL SEARCHDD(NO) in a start-up CLIST or REXX exec that is part of a logon procedure. Users can also use EXECUTIL SEARCHDD(NO) to dynamically change the search order during their TSO/E and ISPF sessions. For more information about the EXECUTIL command, see Chapter 10, “TSO/E REXX commands,” on page 201.

In TSO/E, you can also use the TSO/E ALTLIB command to define alternate exec libraries in which to store implicitly executed REXX execs. Using ALTLIB, you can specify alternate libraries on the user, application, or system level and activate and deactivate individual exec libraries as needed. For more information about using ALTLIB, see *z/OS TSO/E REXX User's Guide*.

Compressing REXX execs

Compression provides a potential performance benefit by reducing the size of the in-storage image of the exec. This could have benefits for all users on a system for execs that are stored in VLF. If you do not want certain execs to be compressed, you can do any one of the following:

- allocate the exec data set to the SYSPROC user level file using the TSO/E ALTLIB command, or to a file that can contain only REXX execs, such as SYSEXEC
- include the character string `SOURCELINE` in the exec, outside of a comment
- specify the compression indicator COMMENT.

REXX execs in the SYSPROC system level, or a CLIST application level library as defined by ALTLIB, are eligible for compression. TSO/E REXX can automatically compress an exec, or you can request that a REXX exec be compressed.

In general, compression eliminates comment text and leading and trailing blanks, and replaces blank lines with null lines, which preserves the line numbering in the exec. For comments, the system removes the comment text but keeps the beginning and ending comment delimiters `/*` and `*/`. This preserves the exec line numbering if the comment spans more than one line. Blanks and comments within literal strings (delimited by either single or double quotation marks) are not removed. Blanks or comments within a Double-Byte Character Set (DBCS) string are not removed.

- Automatic Compression
  - If the system automatically compresses the exec, it replaces the first line of the exec (the comment line containing the characters "REXX") with the comment `/*%NOCOMMENT*/`. If you review a dump of VLF, the `/*%NOCOMMENT*/` comment is an indicator that the exec is compressed. For example, if the initial line 1 comment contains:

    ```
    /* REXX */
    ```

    then after compression, line 1 contains:

    ```
    /*%NOCOMMENT*/
    ```

However, if the line 1 comment also contains other special options (the comment starts with /*%), TSO/E REXX inserts the option NOCOMMENT ahead of the other keyword options in line 1. The remaining keyword options of line 1 are not removed (compressed) from the line 1 comment. For example, if the initial line 1 comment contains:

```
/*% REXX xxxxxxx yyyyyyy */
```

then after compression line 1 contains:

```
/*%NOCOMMENT REXX xxxxxxx yyyyyyy */
```

If the system finds an explicit occurrence of the characters `SOURCELINE` outside of a comment in the exec, it does not compress the exec. For example, if you use the SOURCELINE built-in function, the exec is not compressed. If you use a variable called "ASOURCELINE1", 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 compresses the exec because the characters `SOURCELINE` are not explicitly found.

- Controlled Compression

  – To request compression of the REXX exec, the exec must begin with a special comment in line 1. A comment is called a special comment if it is the first comment in line 1 and the begin comment delimiter `/*` is immediately followed by the special comment trigger character `%`. That is,

```
/*%
```

The trigger character is followed by one or more keyword options. If you specify more than one keyword option, separate them by one or more blanks. You can also use blanks between the trigger character and the first keyword option for better readability. Keyword options can be lower case, upper case, or mixed case. The following are some examples of using a special comment. Note that there are no changes to the special comment after compression.

```
/*% REXX xxxx yyyy NOCOMMENT */
```

then after compression the line is unchanged, except trailing blanks are removed, and the line contains:

```
/*% REXX xxxx yyyy NOCOMMENT*/
```

The scan for keyword options ends by:

- an end comment delimiter (`*/`)
- another begin comment delimiter `/*` to start a nested comment
- the end of the line.

Keyword options cannot continue onto the second physical line, even if the comment itself continues to line two. If TSO/E REXX does not recognize a keyword option, it is ignored.

Rule: DBCS characters cannot be used within the special comment in line 1 to specify keyword options. All keyword options must be specified using the EBCDIC character set.

The compression indicator keyword options are COMMENT and NOCOMMENT, where:

COMMENT
indicates the exec is NOT to be compressed (the exec will contain comments).

NOCOMMENT
indicates the exec is to be compressed (the exec will NOT contain comments).

The COMMENT and NOCOMMENT compression indicators are only valid for execs which are invoked implicitly and are found in either a SYSPROC library or an application level CLIST library.

The following are some examples of using COMMENT and NOCOMMENT. In all of these examples the exec is invoked implicitly and is found in the SYSPROC CLIST library.

- To indicate an exec is to be compressed, the author of the REXX exec might code the following:

```
/*%NOCOMMENT   REXX */
Say 'This exec will be compressed'
x = 5       /* This comment will be removed from the exec */
say 'x is ' x
exit 0      /* leave the exec */
```

- To indicate an exec is NOT to be compressed, the author of the REXX exec might code the following:

```
/*%COMMENT   REXX */
Say 'This exec will not be compressed'
x = 5       /* This comment will not be removed from the exec */
say 'x is ' x
exit 0      /* leave the exec */
```

  Recommendation: The REXX identifier must still be specified within a comment in the first line of any interpreted exec found in SYSPROC. Otherwise, the procedure is treated as a CLIST.

- If you specify the NOCOMMENT keyword option, the NOCOMMENT keyword option, and any other options are not removed from the special comment in line 1 after compression. For example, if the initial line 1 comment contains:

```
/*% xxxxxxx yyyyyyy NOCOMMENT /*REXX*/ */
```

  then after compression line 1 contains:

```
/*% xxxxxxx yyyyyyy NOCOMMENT*/
```

  Note: The REXX identifier was removed from the line 1 comment because it is not a keyword option in this instance. The nested delimiter `/*` ended the scan for special keyword options.

If a compression indicator is specified, the exec is not scanned for the string SOURCELINE. TSO/E REXX determines whether to compress the exec based only on the value of the compression indicator.

The following are examples of using the compression indicator option. In all of these examples the exec is invoked implicitly and is found in the SYSPROC CLIST library.

- Example of **correct** placement of the trigger character:

  In the following example NOCOMMENT is a valid compression indicator option recognized by TSO/E REXX and indicates that the exec should be compressed. The string REXX is seen both as a keyword option and as the REXX REXX identifier.

```
/*% NOCOMMENT  REXX */
```

- Examples of **incorrect** placement of the trigger character:

  In each of these examples NOCOMMENT is not treated as a compression indicator option because the trigger character (%) does not immediately follow the first begin comment delimiter `/*`. The string REXX in the first comment is seen as a valid REXX identifier.

```
/*REXX*/  /*%NOCOMMENT */
```

```
/*REXX  /*%NOCOMMENT */ */
```

```
/* %NOCOMMENT REXX */
```

- Example of specifying the compression indicator in mixed case:

  In the following example the compression indicator option is specified in mixed case. The compression indicator option can be specified in uppercase, lowercase, or mixed case. The "NOcomment" indicator is a valid keyword option and indicates that the exec should be compressed. The start of the nested comment ends scanning for keyword options. The string "REXX" is seen as a REXX identifier and not as a keyword option. The words within the nested comment are *not* treated as keyword options.

```
/*% NOcomment       /*REXX - Remove any comment text*/ */
```

- Example of using two compression keyword options:

  If both COMMENT and NOCOMMENT are specified in the set of keyword options in the comment in line 1, TSO/E REXX uses the last value specified. In this example, the exec sees the last compression indicator COMMENT and the exec is not compressed.

  ```
  /*% NOCOMMENT  COMMENT  - REXX */
  ```

- Example of SOURCELINE that does not prevent compression:

  In the following example the string SOURCELINE is not apparent in the source text of the exec because it is formed dynamically during the processing of the INTERPRET statement. Use of the COMMENT keyword option informs TSO/E REXX not to compress this exec. Without the COMMENT indicator this exec would be compressed and this would have caused the SOURCELINE built-in function to obtain a null comment, that is `/**/`, for the source text of line 2.

  ```
  /*%COMMENT    REXX */
  /* Second line of exec */
  interpret 'line2 = SOURCE'||'LINE(2)'
  if substr(line2,1,25) = '/* Second line of exec */' then
    do
      say 'Found the correct line 2 source text'
      say '... Exec executed correctly'
    end
   else
    do
      say 'Did not find correct line 2 source text'
      say '... Exec executed incorrectly'
    end
   exit 0
  ```

- Example of a missing REXX identifier:

  In this example, NOCOMMENT is specified as a keyword option but is not recognized because no REXX identifier appears in the line 1 comment. Any procedure invoked implicitly from a SYSPROC library that does not contain the REXX identifier is treated as a CLIST. Therefore, no scan for keyword options is performed and the system attempts to run this exec as a CLIST.

  ```
  /*%NOCOMMENT      */
  ```

- Example of **incorrectly** specified compression indicator:

  In the following example, the NOCOMMENT indicator is not recognized. The REXX identifier should be separated from the NOCOMMENT indicator by at least one blank. TSO/E REXX views NOCOMMENTREXX NOCOMMENTOREXX as a single unrecognized exec keyword option. Note, however, the word REXX imbedded within the keyword option NOCOMMENTREXX is seen as a valid REXX identifier string.

  ```
  /*%NOCOMMENTREXX*/
  ```

  Rule: Avoid concatenating data sets of unlike RECFM attributes to the SYSEXEC or SYSPROC libraries or to any of the REXX exec or CLIST libraries that are defined by using the ALTLIB command. In addition, if RECFM=FB data sets are used in the library allocated to SYSEXEC, or in any exec library allocated by using the ALTLIB command, all the data sets which comprise that library concatenation should have the same LRECL. Follow these directives or various failures, including abends or loops, during REXX exec load processing, can result.

Control blocks created for a Language Processor Environment

When IRXINIT initializes a new language processor environment, IRXINIT creates a number of control blocks that contain information about the environment. The main control block is the *environment block* (ENVBLOCK). The environment block contains pointers to:

- The parameter block (PARMBLOCK), which is a control block containing the parameters IRXINIT used to define the environment. The parameter block IRXINIT creates has the same format as the parameters module.
- The user field that was passed on the call to IRXINIT if IRXINIT was explicitly invoked by a user.
- The work block extension, which is a control block that contains information about the REXX exec that is currently running.
- The REXX vector of external entry points, which contains the addresses of the REXX routines TSO/E provides, such as IRXINIT, IRXTERM, REXX programming routines, and replaceable routines. For replaceable routines, the vector contains the addresses of both the system-supplied routines and any user-supplied routines.
- The TSO/E REXX routine that encountered the first error and issued the first error message in the environment.
- The compiler programming table, which identifies compiler runtime processors and corresponding compiler interface routines.

Rule About Changing Any Control Blocks: You can obtain information from the control blocks. However, you *must not change* any of the control blocks. If you do, unpredictable results may occur.

Format of the environment block (ENVBLOCK)

Table 63 on page 358 shows the format of the environment block. TSO/E provides a mapping macro, IRXENVB, for the environment block. The mapping macro is in SYS1.MACLIB.

When IRXINIT initializes a new language processor environment, IRXINIT returns the address of the new environment block in register 0 and in parameter 6 in the parameter list. You can use the environment block to locate information about a specific environment. For example, the environment block points to the REXX vector of external entry points that contains the addresses of routines that perform system services, such as I/O, data stack, and exec load. Using the control blocks lets you easily call one of the routines.

Table 63. Format of the environment block

Offset (decimal)	Number of bytes	Field name	Description
0	8	ID	An eight-character field that identifies the environment block. The field contains the characters ‘ENVBLOCK’.
8	4	VERSION	A 4-byte field that contains the character representation of the version number of the environment block. The version number is 0100.
12	4	LENGTH	The length of the environment block. The number is 320, in decimal.
16	4	PARMBLOCK	The address of the parameter block (PARMBLOCK). See “Format of the parameter block (PARMBLOCK)” on page 360 for more information.
20	4	USERFIELD	The address of the user field that is passed to IRXINIT if you explicitly called IRXINIT. You pass the user field in parameter 4 (see “Initialization routine - IRXINIT” on page 371 for information about the parameters). You can use this field for your own processing. The TSO/E REXX services do not use this field.

Table 63. Format of the environment block (continued)

Offset (decimal)	Number of bytes	Field name	Description
24	4	WORKBLOK_EXT	The address of the current work block extension. If an exec is not currently running in the environment, the address is 0. See "Format of the work block extension" on page 360 for details about the work block extension.
28	4	IRXEXTE	The address of the REXX vector of external entry points. See "Format of the REXX vector of external entry points" on page 362 for details about the vector.
32	4	ERROR_CALL@	The address of the TSO/E REXX routine that encountered the first error in the language processor environment and that issued the first error message. The error could have occurred while an exec was running or when a particular service was requested in the environment.
36	4	—	Reserved.
40	8	ERROR_MSGID	An eight-character field that contains the message ID of the first error message the system issued in the language processor environment. The message relates to the error encountered by the routine that is pointed to at offset +32.
48	80	PRIMARY_ERROR_MESSAGE	An 80-character field that contains the primary error message (the message text) for the message ID at offset +40.
128	160	ALTERNATE_ERROR_MESSAGE	A 160-character field that contains the alternate error message (the message text) for the message ID at offset +40.
288	4	COMPGMTB	The address of the compiler programming table for the language processor environment. The table identifies a compiler runtime processor and corresponding compiler interface routines. If a compiler programming table is not available to the language processor environment, this field is 0. For information about the compiler programming table, see *z/OS TSO/E Customization*.
292	4	ATTNROUT_PARMPTR	The address of an attention handling routine control block. The attention handling exit can optionally use this control block to communicate with REXX attention processing. For more information about the control block, see *z/OS TSO/E Customization*.
296	4	ECTPTR	The address of the ECT under which this environment is anchored. This field is only used for environments which are integrated into TSO/E. Otherwise, it is zero.
300	4	INFO_FLAGS	A fullword of bits that gives status of this environment block. Bit 0 is the only bit that is used. Bits 1 through 31 are reserved. • Bit 0 (TERMA_CLEANUP). This bit is on if the environment is undergoing abnormal termination. (See Appendix B, "IRXTERMA routine," on page 445 for information about abnormal termination.

The following topics describe the format of the parameter block (PARMBLOCK), the work block extension, and the vector of external entry points.

Format of the parameter block (PARMBLOCK)

The parameter block (PARMBLOCK) contains information about the parameters that IRXINIT used to define the environment. The environment block points to the parameter block.

Table 64 on page 360 shows the format of the parameter block. TSO/E provides a mapping macro, IRXPARMB, for the parameter block. The mapping macro is in SYS1.MACLIB.

The parameter block has the same format as the parameters module. See “Characteristics of a Language Processor Environment” on page 317 for information about the parameters module and a complete description of each field.

Table 64. Format of the parameter block (PARMBLOCK)

Offset (decimal)	Number of bytes	Field name	Description
0	8	ID	An eight-character field that identifies the parameter block. The field contains the characters ‘IRXPARMS’.
8	4	VERSION	A 4-byte field that contains the version number of the parameter block in EBCDIC. The version number is 0200.
12	3	LANGUAGE	Language code for REXX messages.
15	1	—	Reserved.
16	4	MODNAMET	Address of the module name table. See “Module name table” on page 326 for a description of the table.
20	4	SUBCOMTB	Address of the host command environment table. See “Host command environment table” on page 331 for a description of the table.
24	4	PACKTB	Address of the function package table. See “Function package table” on page 333 for a description of the table.
28	8	PARSETOK	Token for the PARSE SOURCE instruction.
36	4	FLAGS	A fullword of bits that represent the flags that IRXINIT used in defining the environment. The flags in the parameter block are in the same order as in the parameters module. See “Flags and corresponding masks” on page 322 for a complete description of the flags.
40	4	MASKS	A fullword of bits that represent the mask settings of the flag bits that IRXINIT used in defining the environment. The masks are in the same order as in the parameters module. See “Flags and corresponding masks” on page 322 for a complete description of the flags and their corresponding masks.
44	4	SUBPOOL	Number of the subpool for storage allocation.
48	8	ADDRSPN	Name of the address space.
56	8	—	The end of the parameter block is indicated by X'FFFFFFFFFFFFFFFF'.

Format of the work block extension

The work block extension contains information about the REXX exec that is currently running. The environment block points to the work block extension.

When IRXINIT first initializes a new environment and creates the environment block, the address of the work block extension in the environment block is 0. The address is 0 because a REXX exec is not yet running in the environment. At this point, IRXINIT is only initializing the environment.

When an exec starts running in the environment, the environment block is updated to point to the work block extension describing the exec. If an exec is running and invokes another exec, the environment block is updated to point to the work block extension for the second exec. The work block extension for the first exec still exists, but the environment block does not point to it. When the second exec completes and returns control to the first exec, the environment block is changed again to point to the work block extension for the original exec.

The work block extension contains the parameters that are passed to the IRXEXEC routine to invoke the exec. You can call IRXEXEC explicitly to invoke an exec and pass the parameters on the call. If you use IRXJCL, implicitly or explicitly invoke an exec in TSO/E, or run an exec in TSO/E background, the IRXEXEC routine always gets control to run the exec. "Exec processing routines - IRXJCL and IRXEXEC" on page 245 describes the IRXEXEC routine in detail and each parameter that IRXEXEC receives.

Table 65 on page 361 shows the format of the work block extension. TSO/E provides a mapping macro, IRXWORKB, for the work block extension. The mapping macro is in SYS1.MACLIB.

Table 65. Format of the work block extension

Offset (decimal)	Number of bytes	Field name	Description
0	4	EXECBLK	The address of the exec block (EXECBLK). See "The exec block (EXECBLK)" on page 254 for a description of the control block.
4	4	ARGTABLE	The address of the arguments for the exec. The arguments are arranged as a vector of address/length pairs followed by X'FFFFFFFFFFFFFFFF'. See "Format of argument list" on page 255 for a description of the argument list.
8	4	FLAGS	A fullword of bits that IRXEXEC uses as flags. IRXEXEC uses bits 0, 1, 2, and 3 only. The remaining bits are reserved. Bits 0, 1, and 2 are mutually exclusive. • Bit 0 – If the bit is on, the exec was invoked as a "command" (that is, the exec was not invoked from another exec as an external function or subroutine). • Bit 1 – If the bit is on, the exec was invoked as an external function (a function call). • Bit 2 – If the bit is on, the exec was invoked as a subroutine using the CALL instruction. • Bit 3 – If the bit is on and a syntax error occurs, IRXEXEC returns a return code from 20001 – 20099. If the bit is off and a syntax error occurs, IRXEXEC returns with return code 0. See Table 17 on page 251 for more information about bit 3.
12	4	INSTBLK	The address of the in-storage control block (INSTBLK). See "The in-storage control block (INSTBLK)" on page 256 for a description of the control block.
16	4	CPPLPTR	The address of the command processor parameter list (CPPL) if you invoked the exec from the TSO/E address space. If you invoked the exec from a non-TSO/E address space, the address is 0.

Table 65. Format of the work block extension (continued)

Offset (decimal)	Number of bytes	Field name	Description
20	4	EVALBLOCK	The address of the evaluation block (EVALBLOCK). See "The evaluation block (EVALBLOCK)" on page 259 for a description of the control block.
24	4	WORKAREA	The address of an 8-byte field that defines a work area for the IRXEXEC routine. See Table 17 on page 251 for more information about the work area.
28	4	USERFIELD	The address of the user field that is passed to IRXEXEC if you explicitly called IRXEXEC. You pass the address of the user field in parameter 8 (see "The IRXEXEC routine" on page 249 for information about the parameters). You can use this field for your own processing. Any of the REXX services do not use this field.
32	4	RTPROC	A fullword that is available for use by a REXX compiler runtime processor. This field allows a compiler runtime processor to have an *anchor* that is unique for each compiled REXX exec that runs within a language processor environment. A compiler runtime processor can use this field for its own purpose. TSO/E REXX does not check or change this field.
36	4	SOURCE_ADDRESS	The address of the PARSE SOURCE string for the exec currently processing. This is the string that the PARSE SOURCE instruction would return.
40	4	SOURCE_LENGTH	The length of the PARSE SOURCE string that is pointed to by the SOURCE_ADDRESS field at offset +36 (decimal).

Format of the REXX vector of external entry points

The REXX vector of external entry points is a control block that contains the addresses of REXX programming routines and replaceable routines. The environment block points to the vector. Table 66 on page 363 shows the format of the vector of external entry points. TSO/E provides a mapping macro, IRXEXTE, for the vector. The mapping macro is in SYS1.MACLIB.

The vector allows you to easily access the address of a particular TSO/E REXX routine to call the routine. The table contains the number of entries in the table followed by the entry points (addresses) of the routines.

Each REXX external entry point has an alternate entry point to permit FORTRAN programs to call the entry point. The external entry points and their alternates are:

Primary entry point name	Alternate entry point name
IRXINIT	IRXINT
IRXLOAD	IRXLD
IRXSUBCM	IRXSUB
IRXEXEC	IRXEX
IRXINOUT	IRXIO

Primary entry point name	Alternate entry point name
IRXJCL	IRXJCL (same)
IRXRLT	IRXRLT (same)
IRXSTK	IRXSTK (same)
IRXTERM	IRXTRM
IRXIC	IRXIC (same)
IRXUID	IRXUID (same)
IRXTERMA	IRXTMA
IRXMSGID	IRXMID
IRXEXCOM	IRXEXC
IRXSAY	IRXSAY (same)
IRXERS	IRXERS (same)
IRXHST	IRXHST (same)
IRXHLT	IRXHLT (same)
IRXTXT	IRXTXT (same)
IRXLIN	IRXLIN (same)
IRXRTE	IRXRTE (same)

For the replaceable routines, the vector provides two addresses for each routine. The first address is the address of the replaceable routine the user supplied for the language processor environment. If a user did not supply a replaceable routine, the address points to the default system routine. The second address points to the default system routine. Chapter 16, “Replaceable routines and exits,” on page 389 describes replaceable routines in detail.

Table 66. Format of REXX vector of external entry points

Offset (decimal)	Number of bytes	Field name	Description
0	4	ENTRY_COUNT	The total number of entry points included in the vector. The number is 26.
4	4	IRXINIT	The address of the initialization routine, IRXINIT.
8	4	LOAD_ROUTINE	The address of the user-supplied exec load replaceable routine for the language processor environment. This is the routine that is specified in the EXROUT field of the module name table. If a replaceable routine is not specified, the address points to the system-supplied exec load routine, IRXLOAD.
12	4	IRXLOAD	The address of the system-supplied exec load routine, IRXLOAD.
16	4	IRXEXCOM	The address of the variable access routine, IRXEXCOM.
20	4	IRXEXEC	The address of the exec processing routine, IRXEXEC.

Table 66. Format of REXX vector of external entry points (continued)

Offset (decimal)	Number of bytes	Field name	Description
24	4	IO_ROUTINE	The address of the user-supplied I/O replaceable routine for the language processor environment. This is the routine that is specified in the IOROUT field of the module name table. If a replaceable routine is not specified, the address points to the system-supplied I/O routine, IRXINOUT.
28	4	IRXINOUT	The address of the system-supplied I/O routine, IRXINOUT.
32	4	IRXJCL	The address of the IRXJCL routine.
36	4	IRXRLT	The address of the IRXRLT (get result) routine.
40	4	STACK_ROUTINE	The address of the user-supplied data stack replaceable routine for the language processor environment. This is the routine that is specified in the STACKRT field of the module name table. If a replaceable routine is not specified, the address points to the system-supplied data stack routine, IRXSTK.
44	4	IRXSTK	The address of the system-supplied data stack handling routine, IRXSTK.
48	4	IRXSUBCM	The address of the host command environment routine, IRXSUBCM.
52	4	IRXTERM	The address of the termination routine, IRXTERM.
56	4	IRXIC	The address of the trace and execution control routine, IRXIC.
60	4	MSGID_ROUTINE	The address of the user-supplied message ID replaceable routine for the language processor environment. This is the routine that is specified in the MSGIDRT field of the module name table. If a replaceable routine is not specified, the address points to the system-supplied message ID routine, IRXMSGID.
64	4	IRXMSGID	The address of the system-supplied message ID routine, IRXMSGID.
68	4	USERID_ROUTINE	The address of the user-supplied user ID replaceable routine for the language processor environment. This is the routine that is specified in the IDROUT field of the module name table. If a replaceable routine is not specified, the address points to the system-supplied user ID routine, IRXUID.
72	4	IRXUID	The address of the system-supplied user ID routine, IRXUID.
76	4	IRXTERMA	The address of the termination routine, IRXTERMA.
80	4	IRXSAY	The address of the SAY instruction routine, IRXSAY.

Table 66. Format of REXX vector of external entry points (continued)

Offset (decimal)	Number of bytes	Field name	Description
84	4	IRXERS	The address of the external routine search routine, IRXERS. The IRXERS routine is a REXX compiler programming routine and is described in *z/OS TSO/E Customization*.
88	4	IRXHST	The address of the host command search routine, IRXHST. The IRXHST routine is a REXX compiler programming routine and is described in *z/OS TSO/E Customization*.
92	4	IRXHLT	The address of the halt condition routine, IRXHLT.
96	4	IRXTXT	The address of the text retrieval routine, IRXTXT.
100	4	IRXLIN	The address of the LINESIZE built-in function routine, IRXLIN.
104	4	IRXRTE	The address of the exit routing routine, IRXRTE. The IRXRTE routine is a REXX compiler programming routine and is described in *z/OS TSO/E Customization*.

Changing the maximum number of environments in an address space

Within an address space, language processor environments are chained together to form a chain of environments. There can be many environments on a single chain. You can also have more than one chain of environments in a single address space. There is a maximum number of environments that can be initialized at one time in an address space. The maximum is not a specific number because the maximum depends on the number of chains in an address space and the number of environments on each chain. The default maximum TSO/E provides should be sufficient for any address space. However, if IRXINIT initializes a new environment and the maximum number of environments has been reached, IRXINIT completes unsuccessfully and returns with a return code of 20 and a reason code of 24. If this error occurs, you can change the maximum value.

The maximum number of environments the system can initialize in an address space depends on the maximum number of entries defined in the environment table known as IRXANCHR, and on the kind of environments being initialized. To change the number of environment table entries, you can use the IRXTSMPE sample that TSO/E provides in SYS1.SAMPLIB or you can create your own IRXANCHR load module. The IRXTSMPE sample is a System Modification Program/Extended (SMP/E) user modification (USERMOD) to change the number of language processor environments in an address space. The prolog of IRXTSMPE has instructions for using the sample job. The SMP/E code that is included in the IRXTSMPE sample handles the installation of the load module.

Guideline: To determine the number of entries needed in IRXANCHR and to ensure that at least "n" environments can be initialized in an address space, use the following formula:

```
      Number_of_entries = (2*n) + 1
```

For example, if you require 100 environments, set the number of entries to 201. You can then initialize at least 100 environments.

If you create your own IRXANCHR load module, you must assemble the code and then link-edit the module as non-reentrant and reusable. You can place the data set in a STEPLIB or JOBLIB, or in the linklist. The data set cannot be in the LPALIB. When running in a TSO/E address space, IRXANCHR must come from an APF-authorized library.

Table 67 on page 366 describes the environment table. TSO/E provides a mapping macro, IRXENVT, for the environment table. The mapping macro is in SYS1.MODGEN.

The environment table consists of a table header followed by table entries. The header contains the ID, version, total number of entries, number of used entries, and the length of each entry. Following the header, each entry is 40 bytes long.

Table 67. Format of the environment table

Offset (decimal)	Number of bytes	Field name	Description
0	8	ID	An eight-character field that identifies the environment table. The field contains the characters ‘IRXANCHR’.
8	4	VERSION	The version of the environment table. The value must be 0100 in EBCDIC.
12	4	TOTAL	Specifies the total number of entries in the environment table.
16	4	USED	Specifies the total number of entries in the environment table that are used.
20	4	LENGTH	Specifies the length of each entry in the environment table. The length of each entry is 40 bytes.
24	8	—	Reserved.
32	40	FIRST	The first environment table entry. Each entry is 40 bytes long. The remaining entries follow.

Using the data stack in different environments

The data stack is a repository for storing data for use by a REXX exec. You can place elements on the data stack using the PUSH and QUEUE instructions, and take elements off the data stack using the PULL instruction. You can also use TSO/E REXX commands to manipulate the data stack. For example, you can use the MAKEBUF command to create a buffer on the data stack and then add elements to the data stack. You can use the QELEM command to query how many elements are currently on the data stack above the most recently created buffer. Chapter 10, “TSO/E REXX commands,” on page 201 describes the REXX commands for manipulating the data stack. *z/OS TSO/E REXX User's Guide* describes how to use the data stack and associated commands.

The data stack is associated with one or more language processor environments. The data stack is shared among all REXX execs that run within a specific language processor environment.

A data stack may or may not be available to REXX execs that run in a particular language processor environment. Whether a data stack is available depends on the setting of the NOSTKFL flag (see “Flags and corresponding masks” on page 322). When IRXINIT initializes an environment and the NOSTKFL flag is on, IRXINIT does not create a data stack or make a data stack available to the language processor environment. Execs that run in the environment cannot use a data stack.

If the NOSTKFL flag is off, either IRXINIT initializes a new data stack for the new environment or the new environment shares a data stack that was initialized for a previous environment. Whether IRXINIT initializes a new data stack for the new environment depends on:

- The setting of the NEWSTKFL (new data stack) flag
- Whether the environment is the first environment that IRXINIT is initializing on a chain

Restriction: The NOSTKFL flag takes precedence over the NEWSTKFL flag. If the NOSTKFL flag is on, IRXINIT does not create a data stack or make a data stack available to the new environment regardless of the setting of the NEWSTKFL flag.

If the environment is the first environment on a chain, IRXINIT automatically initializes a new data stack regardless of the setting of the NEWSTKFL flag.

Restriction: If the NOSTKFL flag is on, IRXINIT does not initialize a data stack.

If the environment is not the first one on the chain, IRXINIT determines the setting of the NEWSTKFL flag. If the NEWSTKFL flag is off, IRXINIT does not create a new data stack for the new environment. The language processor environment shares the data stack that was most recently created for one of the parent environments. If the NEWSTKFL flag is on, IRXINIT creates a new data stack for the language processor environment. Any REXX execs that run in the new environment can access only the new data stack for this environment. Execs cannot access any data stacks that IRXINIT created for any parent environment on the chain.

Environments can only share data stacks that were initialized by environments that are higher on a chain.

If IRXINIT creates a data stack when it initializes an environment, the system deletes the data stack when that environment is terminated. The data stack is deleted at environment termination regardless of whether any elements are on the data stack. All elements on the data stack are lost.

Figure 25 on page 367 shows three environments that are initialized on one chain. Each environment has its own data stack, that is, the environments do not share a data stack.

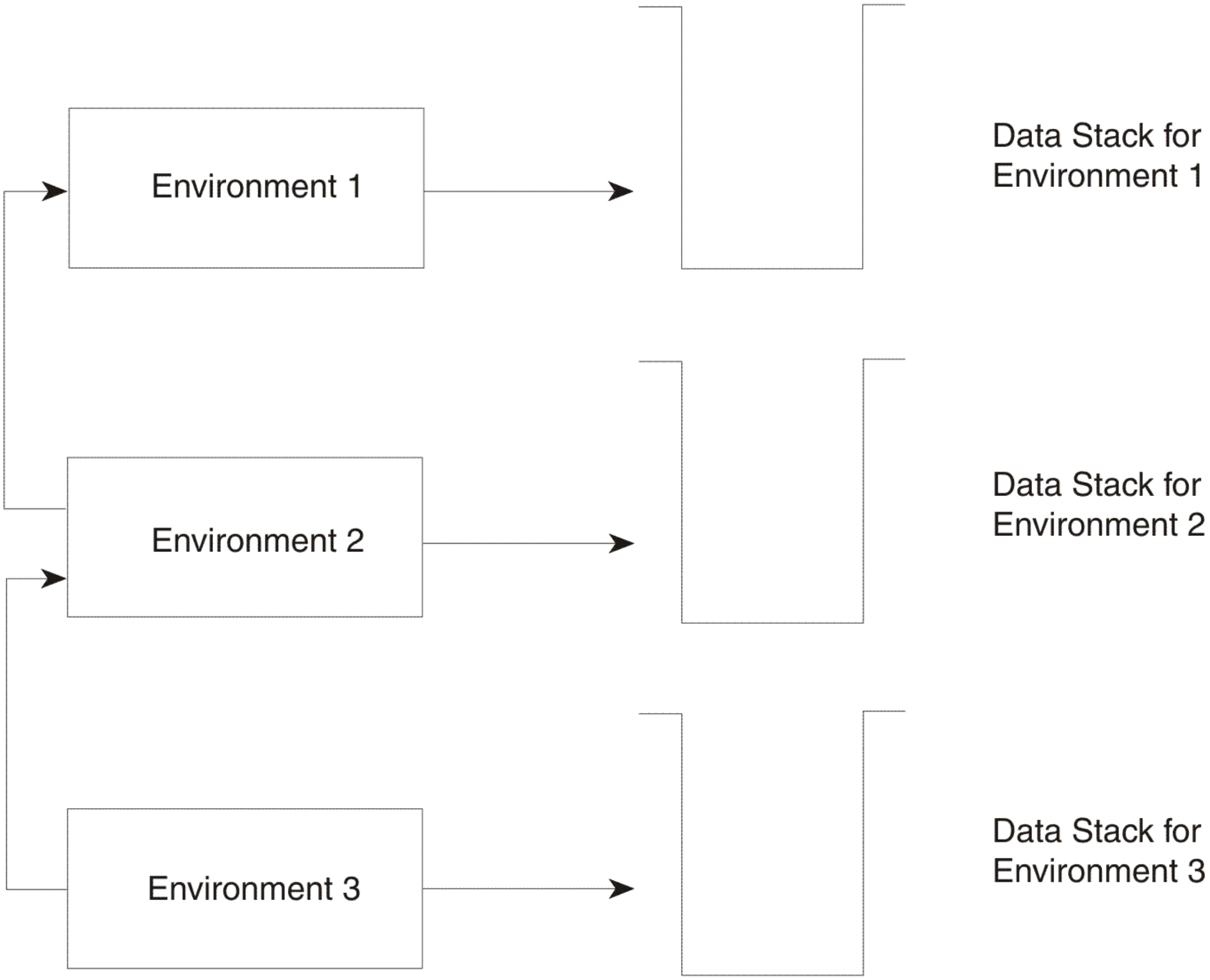

Environment 1
Data Stack for Environment 1
Environment 2
Data Stack for Environment 2
Environment 3
Data Stack for Environment 3

Figure 25. Separate data stacks for each environment

When environment 1 was initialized, it was the first environment on the chain. Therefore, a data stack was automatically created for environment 1. Any REXX execs that execute in environment 1 access the data stack associated with environment 1.

When environment 2 and environment 3 were initialized, the NEWSTKFL flag was set on, indicating that a data stack was to be created for the new environment. The data stack associated with each environment is separate from the stack for any of the other environments. If an exec executes, it executes in the most current environment (environment 3) and only has access to the data stack for environment 3.

Figure 26 on page 368 shows two environments that are initialized on one chain. The two environments share one data stack.

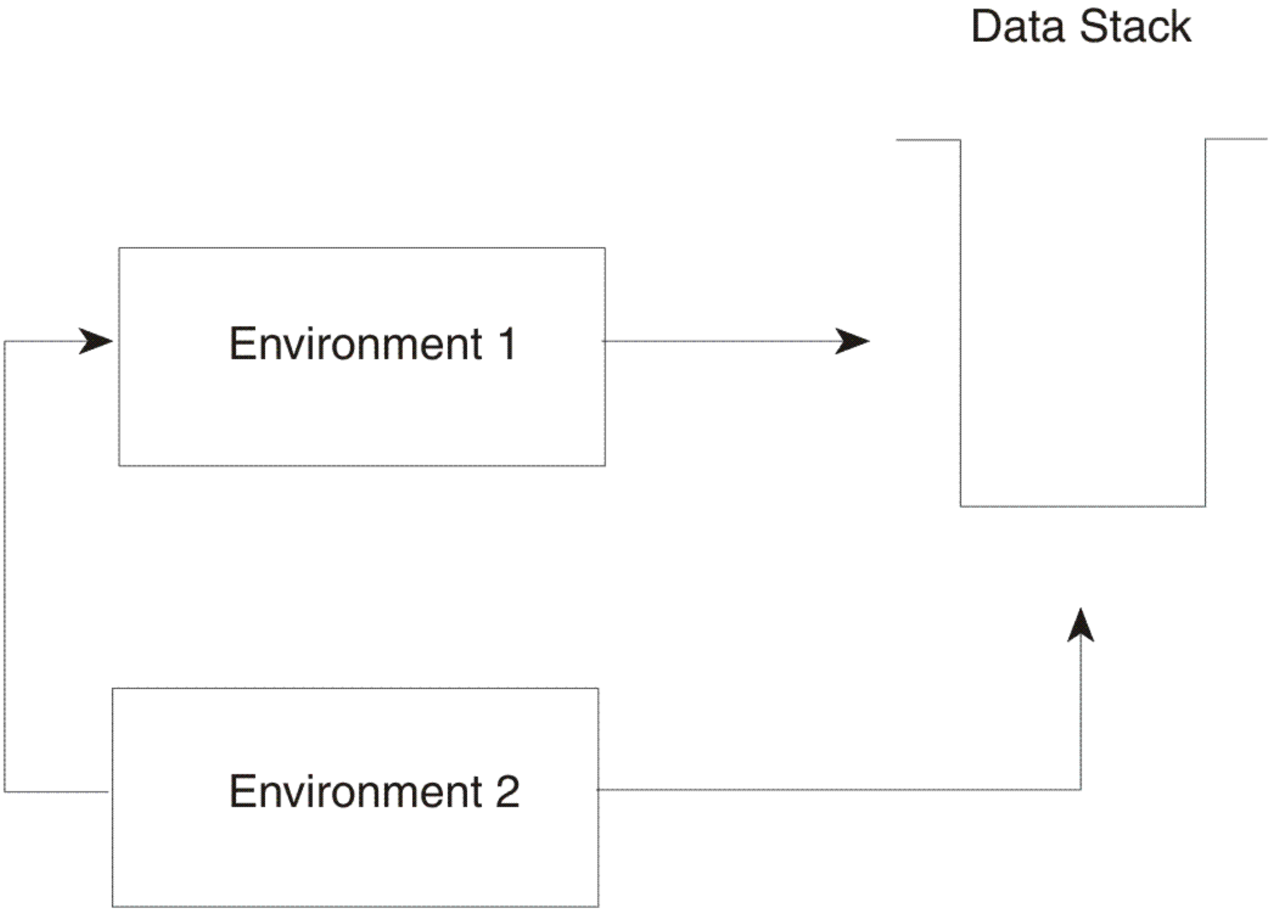


Figure 26. Sharing of the data stack between environments

When environment 1 was initialized, it was the first environment on the chain. Therefore, a data stack was automatically created. When environment 2 was initialized, the NEWSTKFL flag was off indicating that a new data stack should not be created. Environment 2 shares the data stack that was created for environment 1. Any REXX execs that execute in either environment use the same data stack.

Suppose a third language processor environment was initialized and chained to environment 2. If the NEWSTKFL flag is off for the third environment, it would use the data stack that was most recently created on the chain. That is, it would use the data stack that was created when environment 1 was initialized. All three environments would share the same data stack.

As described, several language processor environments can share one data stack. On a single chain of environments, one environment can have its own data stack and other environments can share a data stack. Figure 27 on page 369 shows three environments on one chain. When environment 1 was initialized, a data stack was automatically created because it is the first environment on the chain. Environment 2 was initialized with the NEWSTKFL flag on, which means a new data stack was created for environment 2. Environment 3 was initialized with the NEWSTKFL flag off, so it uses the data stack that was created for environment 2.

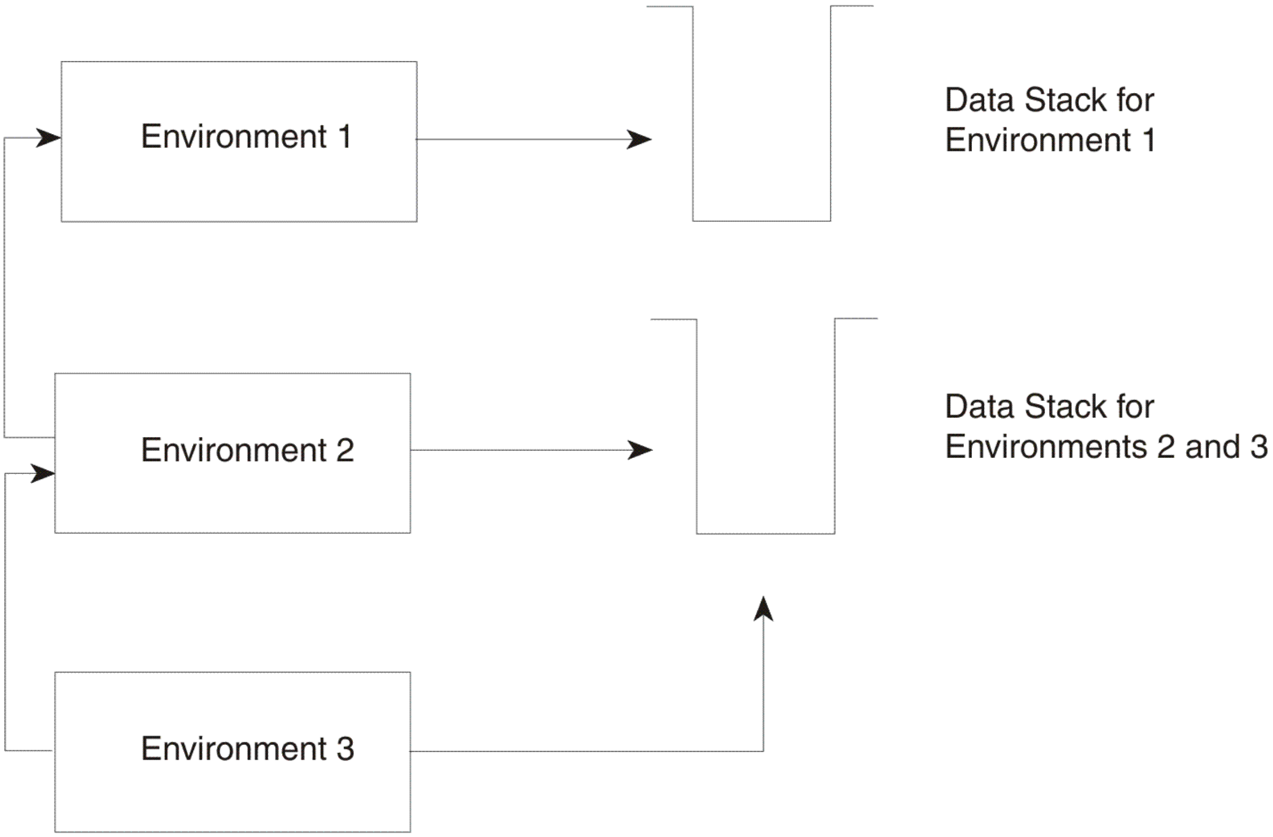

Environment 1
Data Stack for Environment 1
Environment 2
Data Stack for Environments 2 and 3
Environment 3

Figure 27. Separate data stack and sharing of a data stack

Environments can be created without having a data stack, that is, the NOSTKFL flag is on. Referring to Figure 27 on page 369, suppose environment 2 was initialized with the NOSTKFL flag on, which means a new data stack was not created and the environment does not share the first environment's (environment 1) data stack. If environment 3 is initialized with the NOSTKFL flag off (meaning a data stack should be available to the environment), and the NEWSTKFL flag is off (meaning a new data stack is not created for the new environment), environment 3 shares the data stack created for environment 1.

When a data stack is shared between multiple language processor environments, any REXX execs that execute in any of the environments use the same data stack. This sharing can be useful for applications where a parent environment needs to share information with another environment that is lower on the environment chain. At other times, a particular exec may need to use a data stack that is not shared with any other execs that are executing on different language processor environments. TSO/E REXX provides the NEWSTACK command that creates a new data stack and that basically hides or isolates the original data stack. Suppose two language processor environments are initialized on one chain and the second environment shares the data stack with the first environment. If a REXX exec executes in the second environment, it shares the data stack with any execs that are running in the first environment. The exec in environment 2 may need to access its own data stack that is private. In the exec, you can use the NEWSTACK command to create a new data stack. The NEWSTACK command creates a new data stack and hides all previous data stacks that were originally accessible and all data that is on the original stacks. The original data stack is referred to as the *primary stack*. The new data stack that was created by the NEWSTACK command is known as the *secondary stack*. Secondary data stacks are private to the language processor environment in which they were created. That is, they are not shared between two different environments.

Figure 28 on page 370 shows two language processor environmentss that share one primary data stack. When environment 2 was initialized, the NEWSTKFL flag was off indicating that it shares the data stack created for environment 1. When an exec was executing in environment 2, it issued the NEWSTACK command to create a secondary data stack. After NEWSTACK is issued, any data stack requests are only performed against the new secondary data stack. The primary stack is isolated from any execs executing in environment 2.

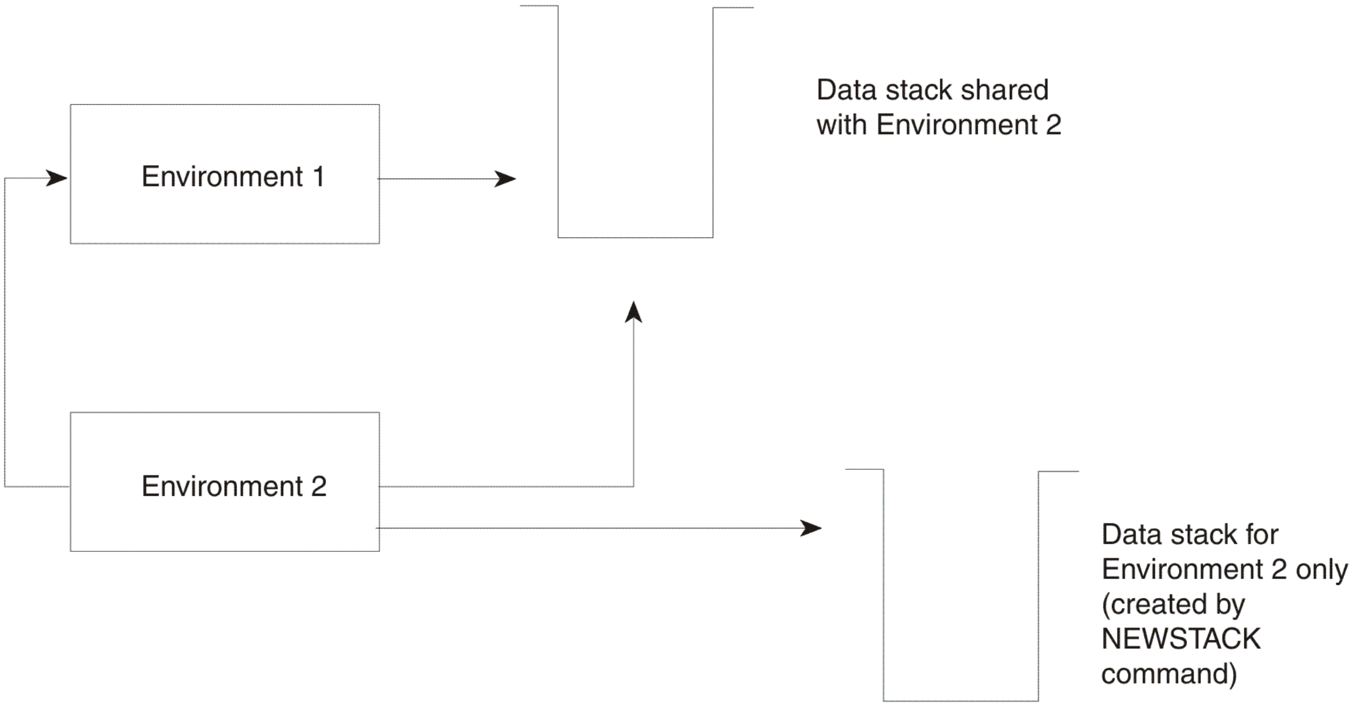

Data stack shared with Environment 2
Environment 1
Environment 2
Data stack for Environment 2 only (created by NEWSTACK command)

Figure 28. Creating a new data stack with the NEWSTACK command

If an exec executing in environment 1 issues the NEWSTACK command to create a secondary data stack, the secondary data stack is available only to REXX execs that execute in environment 1. Any execs that execute in environment 2 cannot access the new data stack created for environment 1.

TSO/E REXX also provides the DELSTACK command that you use to delete any secondary data stacks that were created using NEWSTACK. When the secondary data stack is no longer required, the exec can issue DELSTACK to delete the secondary stack. At this point, the primary data stack that is shared with environment 1 is accessible.

TSO/E REXX provides several other commands you can use for data stack functions. For example, an exec can use the QSTACK command to find out the number of data stacks that exist for the language processor environment. Chapter 10, “TSO/E REXX commands,” on page 201 describes the different stack-oriented commands that TSO/E REXX provides, such as NEWSTACK and DELSTACK.

Chapter 15. Initialization and termination routines

This chapter provides information about how to use the initialization routine, IRXINIT, and the termination routine, IRXTERM.

Use the initialization routine, IRXINIT, to either initialize a language processor environment or obtain the address of the environment block for the current non-reentrant environment. Use the termination routine, IRXTERM, to terminate a language processor environment. Chapter 8, “Using REXX in different address spaces,” on page 187 provides general information about how the initialization and termination of environments relates to REXX processing. Chapter 14, “Language processor environments,” on page 311 describes the concept of a language processor environment in detail, the various characteristics you can specify when initializing an environment, the default parameters modules, and information about the environment block and the format of the environment block.

Initialization routine - IRXINIT

Use IRXINIT to either initialize a new language processor environment or obtain the address of the environment block for the current non-reentrant environment. The "current" environment is the last non-reentrant environment created on this task control block (TCB) or a parent TCB. A non-reentrant environment found on a parent TCB is only considered "current" if it is integrated into TSO/E (TSOFL='1'B).

Tip: To permit FORTRAN programs to call IRXINIT, TSO/E provides an alternate entry point for the IRXINIT routine. The alternate entry point name is IRXINT.

If you use IRXINIT to obtain the address of the current environment block, IRXINIT returns the address in register 0 and also in the sixth parameter.

If you use IRXINIT to initialize a language processor environment, the characteristics for the new environment are based on parameters that you pass on the call and values that are defined for the previous environment. Generally, if you do not pass a specific parameter on the call, IRXINIT uses the value from the previous environment.

IRXINIT always locates a previous environment as follows. On the call to IRXINIT, you can pass the address of an environment block in register 0. IRXINIT then uses this environment as the previous environment if the environment is valid. If register 0 does not contain the address of an environment block, IRXINIT locates the previous environment. If IRXINIT locates a previous environment, IRXINIT uses that environment as the previous environment. If IRXINIT cannot locate an environment, IRXINIT uses the load module IRXPARMS as the previous environment.

“Chains of environments and how environments are located” on page 340 describes in detail how IRXINIT locates a previous environment. A previous environment is always identified regardless of the parameters you specify on the call to IRXINIT.

Using IRXINIT, you can initialize a reentrant or a non-reentrant environment, which is determined by the setting of the RENTRANT flag bit. If you use IRXINIT to initialize a reentrant environment and you want to chain the new environment to a previous reentrant environment, you must pass the address of the environment block for the previous reentrant environment in register 0.

If you use IRXINIT to locate a previous environment, you can locate only the current non-reentrant environment. IRXINIT does not locate a reentrant environment.

You can use IRXINIT CHEKENVB to verify that an address is a valid ENVBLOCK:

- under the current task
- under a parent task
- is not under the current task or under a parent task.

Entry specifications

For the IRXINIT initialization routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional for FINDENVB and INITENVB, required for CHEKENVB)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

You can pass the address of an environment block in register 0. In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter.

The first seven parameters are required. Parameter 8 and parameter 9 are optional. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. If IRXINIT does not find the high-order bit set on in either the address for parameter 7 or in the address of one of the optional parameters from 8 to 11, IRXINIT does not initialize the environment and returns with a return code of 20 and a reason code of 27. See "Output parameters" on page 381 for more information.

Table 68 on page 373 describes the parameters for IRXINIT. For general information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 68. Parameters for IRXINIT

Parameter	Number of bytes	Description
Parameter 1	8	The function IRXINIT is to perform: **INITENVB** To initialize a new environment. **FINDENVB** To obtain the address of the environment block for the current non-reentrant environment. IRXINIT returns the address of the environment block in register 0 and in parameter 6. IRXINIT does not initialize a new environment. **CHEKENVB** To validate that a given address is a REXX ENVBLOCK address. CHEKENVB will return a return code to indicate that the ENVBLOCK address was initialized: • under the current task • under a parent task • elsewhere in the address space If IRXINIT CHEKENVB was called with an incorrect ENVBLOCK address in register 0, IRXINIT will return the current, non-reentrant ENVBLOCK address in parameter 6. IRXINIT does not initialize a new environment.
Parameter 2	8	The name of a parameters module that contains the values for initializing the new environment. The module is described in "Parameters module and in-storage parameter list" on page 379. If the name of the parameters module is blank, IRXINIT assumes that all fields in the parameters module are null. IRXINIT provides two ways in which you can pass parameter values; the parameters module and the address of an in-storage parameter list, which is parameter 3. A complete description of how IRXINIT computes each parameter value and the flexibility of passing parameters is described in "How IRXINIT determines what values to use for the environment" on page 378.
Parameter 3	4	The address of an *in-storage parameter list*, which is an area in storage containing parameters that are equivalent to the parameters in the parameters module. The format of the in-storage list is identical to the format of the parameters module. "Parameters module and in-storage parameter list" on page 379 describes the parameters module and in-storage parameter list. For parameter 3, you can specify an address of 0 for the address of the in-storage parameter list. However, the address in the address list that points to this parameter cannot be 0. If the address of parameter 3 is 0, IRXINIT assumes that all fields in the in-storage parameter list are null.

Table 68. Parameters for IRXINIT (continued)

Parameter	Number of bytes	Description
Parameter 4	4	The address of a user field. IRXINIT does not use or check the user field pointed to by this address. You can use this parameter and the user field to which it points for your own processing. If the value of this parameter is X'80000000', the address of the user field is inherited from the previous environment. Otherwise, the value of this parameter is used as specified. The resulting user field address is available in ENVBLOCK_USERFIELD.
Parameter 5	4	Reserved. This parameter must be set to 0, but the address that points to this parameter cannot be 0.
Parameter 6	4	IRXINIT uses this parameter for output only. The parameter contains the address of the environment block. If you use the FINDENVB function (parameter 1) to locate an environment, parameter 6 contains the address of the environment block for the current non-reentrant environment. If you use the INITENVB function (parameter 1) to initialize a new environment, IRXINIT returns the address of the environment block for the newly created environment in parameter 6. For either FINDENVB or INITENVB, IRXINIT also returns the address of the environment block in register 0. Parameter 6 lets high-level languages obtain the environment block address to examine information in the environment block. For CHEKENVB this parameter will return the current, non-reentrant environment if the ENVBLOCK address in register 0 is not found (that is, return code 12 is returned from IRXINIT).
Parameter 7	4	IRXINIT uses this parameter for output only. IRXINIT returns a reason code for the IRXINIT routine in this field that indicates why the requested function did not complete successfully. Table 70 on page 382 describes the reason codes that IRXINIT returns.
Parameter 8	4	Parameter 8 is an optional parameter that lets you specify how REXX obtains storage in the language processor environment. Specify 0 if you want the system to reserve a default amount of storage workarea. If you want to pass a storage workarea to IRXINIT, specify the address of an *extended parameter list*. The extended parameter list consists of the address (a fullword) of the storage workarea and the length (a fullword) of the workarea, followed by X'FFFFFFFFFFFFFFFF'. For more information about parameter 8 and storage, see "Specifying how REXX obtains storage in the environment" on page 376. Although parameter 8 is optional, it is suggested that you specify an address of 0 if you do not want to pass a storage workarea to IRXINIT.

Table 68. Parameters for IRXINIT (continued)

Parameter	Number of bytes	Description
Parameter 9	4	A 4-byte field that IRXINIT uses to return the return code. The return code parameter is optional. If you use this parameter, IRXINIT returns the return code in the parameter and also in register 15. Otherwise, IRXINIT uses register 15 only. If the parameter list is incorrect, the return code is returned in register 15 only. "Return codes" on page 384 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.
Parameter 10	4	Parameter 10 is an optional parameter that is the address of a pointer to the TSO/E environment control table (ECT) under which the REXX environment is to be initialized. This field is only used when initializing a REXX environment that is integrated into TSO/E. In all other cases, including initializing a non-integrated environment and finding the current environment, this parameter is ignored, if specified. Valid values for this parameter are: • ECT address = the caller's current ECT • '00000000'X - IRXINIT assumes that the primary ECT, the ECT created at TSO/E logon time or TMP initialization, is the caller's current ECT. When this parameter contains '00000000'X upon input, the field is updated to contain the address of the primary ECT. When parameter 10 is not specified and you are initializing a REXX environment that is integrated into TSO/E, the ECT created at TSO/E logon time or TMP initialization is assumed.

Table 68. Parameters for IRXINIT (continued)

Parameter	Number of bytes	Description
Parameter 11	4	Parameter 11 is an optional parameter that is the address of a message buffer (MSGBUF) in which IRXINIT can return an error message if IRXINIT completes with return code 20. The MSGBUF consists of a fullword length header field (MSGHDR) followed by the message text return area (MSGTEXT). The MSGHDR word is divided into two halfword length fields: • The first halfword, MSGTLEN, is used by the caller to pass the length of the MSGTEXT area, excluding the length of the 4-byte header. This value is not changed by IRXINIT. • The second halfword, MSGRLEN, is used by IRXINIT to indicate the actual length of the message text returned in the MSGTEXT area, or 0 if none. (The caller should initialize this halfword to 0 before calling IRXINIT). If a message buffer address is passed and IRXINIT encounters an error for which a message is returned, no message is issued by IRXINIT. Instead, IRXINIT just returns the message. If parm11 is 0, or if this parameter is not passed because fewer than 11 parameters are passed, and if IRXINIT completes with return code 20, a message will be issued to the TSO/E user and/or to the CONSOLE containing information about the error. (This is the default for callers not using this parameter). If the caller of IRXINIT does not want IRXINIT to issue an error message nor return a message when an error occurs, specify parm11 pointing to a buffer with a length header word of X'00000000'. This indicates a buffer of length 0 is passed, so no message will be returned and no message will be issued if an error should occur. If a MSGBUF buffer is passed, the MSGTEXT area should be at least 124 bytes long. If the message being returned is longer than the message buffer space provided, it will be truncated to the size of the buffer provided. Messages issued or returned by IRXINIT are not translated by MMS message processing.

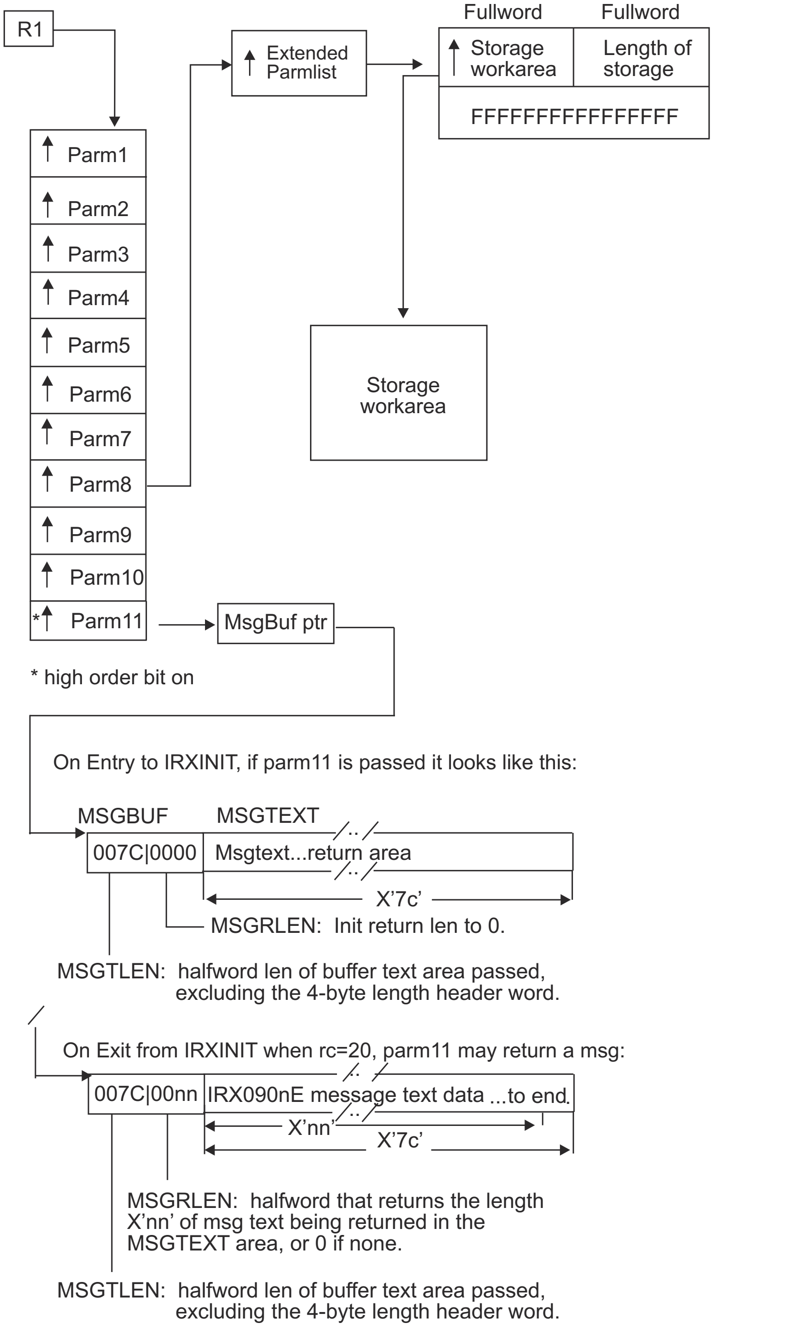
Specifying how REXX obtains storage in the environment

On the call to IRXINIT, parameter 8 is an optional parameter. You can use parameter 8 to specify how REXX obtains storage in the language processor environment for the processing of REXX execs.

If you specify 0 for parameter 8, during the initialization of the environment, the system reserves a default amount of storage for the storage workarea. If you have provided your own storage management replaceable routine, the system calls your routine to obtain this storage workarea. Otherwise, the system obtains storage using GETMAIN. When the environment that IRXINIT is initializing is terminated, the system automatically frees the storage. The system frees the storage by either calling your storage management replaceable routine or using FREEMAIN, depending on how the storage was obtained.

You can also pass a storage workarea to IRXINIT. For parameter 8, specify an address that points to an *extended parameter list*. The extended parameter list is an address/length pair that contains the address (a fullword) of the storage workarea and the length (a fullword) of the storage area, in bytes. The address/

length pair must be followed by X'FFFFFFFFFFFFFFFF' to indicate the end of the extended parameter list. Figure 29 on page 377 shows the extended parameter list.


R1
Extended Parmlist
Fullword
Fullword
Storage workarea
Length of storage
FFFFFFFFFFFFFFFF
Parm1
Parm2
Parm3
Parm4
Parm5
Parm6
Parm7
Parm8
Parm9
Parm10
*Parm11
Storage workarea
MsgBuf ptr
* high order bit on
On Entry to IRXINIT, if parm11 is passed it looks like this:
MSGBUF
MSGTEXT
007C|0000
Msgtext...return area
X'7c'
MSGRLEN: Init return len to 0.
MSGTLEN: halfword len of buffer text area passed, excluding the 4-byte length header word.
On Exit from IRXINIT when rc=20, parm11 may return a msg:
007C|00nn
IRX090nE message text data ...to end.
X'nn'
X'7c'
MSGRLEN: halfword that returns the length X'nn' of msg text being returned in the MSGTEXT area, or 0 if none.
MSGTLEN: halfword len of buffer text area passed, excluding the 4-byte length header word.

Note: To prevent IRXINIT from issuing a message or returning a message, code X'00000000' in the length header word of the MSGBUF passed to IRXINIT. The first halfword (MSGTLEN) is X'0000', which indicates that the length of the MSGTEXT return area is 0, so if an error does occur resulting in return code 20, no message will be returned, and no message will be issued.

To allow IRXINIT to issue an error message if it is unable to initialize a new REXX environment and would otherwise return a return code 20, simply end the parameter list before parm11, or pass parm11 with an address of 0 (no buffer).

Figure 29. Extended parameter list – parameter 8

The storage workarea you pass to IRXINIT is then available for REXX processing in the environment that you are initializing. The storage workarea must remain available to the environment until the environment is terminated. After you terminate the language processor environment, you must also free the storage workarea. The system does not free the storage you pass to IRXINIT when you terminate the environment.

You can also specify that a reserved storage workarea should not be initialized for the environment. The system then obtains and frees storage whenever storage is required. To specify that a storage workarea should not be initialized, for parameter 8, specify the address of the extended parameter list as described above. In the extended parameter list, specify 0 for the address of the storage workarea and 0 for the length of the storage workarea. Again, the address/length pair must be followed by X'FFFFFFFFFFFFFFFF' to indicate the end of the extended parameter list.

Do not run REXX without a reserved storage workarea because of possible performance degradation. However, you might consider this option if available storage is low and you want to initialize a language processor environment with a minimal amount of storage at initialization time.

In the extended parameter list, you can also specify 0 for the address of the storage workarea and -1 for the length of the workarea. This is considered a null entry and IRXINIT ignores the extended parameter list entry. This is equivalent to specifying an address of 0 for parameter 8, and the system reserves a default amount of workarea storage.

In general, 3 pages (12K) of storage is needed for the storage workarea for normal exec processing, for each level of exec nesting. If there is insufficient storage available in the storage workarea, REXX calls the storage management routine to obtain additional storage if you provided a storage management replaceable routine. Otherwise, the system uses GETMAIN and FREEMAIN to obtain and free storage. For more information about the replaceable routine, see "Storage management routine" on page 420.

How IRXINIT determines what values to use for the environment

IRXINIT first determines the values to use to initialize the environment. After all of the values are determined, IRXINIT initializes the new environment using the values.

On the call to IRXINIT, you can pass parameters that define the environment in two ways. You can specify the name of a parameters module (a load module) that contains the values IRXINIT uses to initialize the environment. In addition to the parameters module, you can also pass an address of an area in storage that contains the parameters. This area in storage is called an in-storage parameter list and the parameters it contains are equivalent to the parameters in the parameters module.

The two methods of passing parameter values give you flexibility when calling IRXINIT. You can store the values on disk or build the parameter structure in storage dynamically. The format of the parameters module and the in-storage parameter list is the same. You can pass a value for the same parameter in both the parameters module and the in-storage parameter list.

When IRXINIT computes what values to use to initialize the environment, IRXINIT takes values from four sources using the following hierarchical search order:

1. The in-storage list of parameters that you pass on the call.

   If you pass an in-storage parameter list and the value in the list is not null, IRXINIT uses this value. Otherwise, IRXINIT continues.
2. The parameters module, the name of which you pass on the call.

   If you pass a parameters module and the value in the module is not null, IRXINIT uses this value. Otherwise, IRXINIT continues.
3. The previous environment.

   IRXINIT copies the value from the previous environment.
4. The IRXPARMS parameters module if a previous environment does not exist.

If a parameter has a null value, IRXINIT continues to search until it finds a non-null value. The following types of parameters are defined to be null:

- A character string is null if it either contains only blanks or has a length of zero
- An address is null if its value is 0 (or X'80000000', when specified for the user field parameter in the IRXINIT parameter list)
- A binary number is null if it has the value X'80000000'
- A given bit is null if its corresponding mask is 0.

On the call to IRXINIT, if the address of the in-storage parameter list is 0, all values in the list are defined as null. Similarly, if the name of the parameters module is blank, all values in the parameters module are defined as null.

You need not specify a value for every parameter in the parameters module or the in-storage parameter list. If you do not specify a value, IRXINIT uses the value defined for the previous environment. You need only specify the parameters whose values you want to be different from the previous environment.

Parameters module and in-storage parameter list

The parameters module is a load module that contains the values you want IRXINIT to use to initialize a new language processor environment. TSO/E provides three default parameters modules (IRXPARMS, IRXTSPRM, and IRXISPRM) for initializing environments in non-TSO/E, TSO/E, and ISPF. "Characteristics of a Language Processor Environment" on page 317 describes the parameters modules.

On the call to the IRXINIT, you can optionally pass the name of a parameters module that you have created. The parameters module contains the values you want IRXINIT to use to initialize the new language processor environment. On the call, you can also optionally pass the address of an in-storage parameter list. The format of the parameters module and the in-storage parameter list is identical.

Table 69 on page 379 shows the format of a parameters module and in-storage list. The format of the parameters module is identical to the default modules TSO/E provides. "Characteristics of a Language Processor Environment" on page 317 describes the parameters module and each field in detail. The end of the table must be indicated by X'FFFFFFFFFFFFFFFF'.

Table 69. Parameters module and in-storage parameter list

Offset (decimal)	Number of bytes	Field name	Description
0	8	ID	Identifies the parameter block (PARMBLOCK).
8	4	VERSION	Identifies the version of the parameter block. The value must be 0200.
12	3	LANGUAGE	Language code for REXX messages.
15	1	RESERVED	Reserved.
16	4	MODNAMET	Address of module name table. The module name table contains the names of DDs for reading and writing data and fetching REXX execs, the names of the replaceable routines, and the names of several exit routines.
20	4	SUBCOMTB	Address of host command environment table. The table contains the names of the host command environments that are available and the names of the routines that process commands for each host command environment.
24	4	PACKTB	Address of function package table. The table defines the user, local, and system function packages that are available to REXX execs running in the environment.
28	8	PARSETOK	Token for PARSE SOURCE instruction.
36	4	FLAGS	A fullword of bits used as flags to define characteristics for the environment.
40	4	MASKS	A fullword of bits used as a mask for the setting of the flag bits.

Table 69. Parameters module and in-storage parameter list (continued)

Offset (decimal)	Number of bytes	Field name	Description
44	4	SUBPOOL	Number of the subpool for storage allocation.
48	8	ADDRSPN	Name of the address space.
56	8	—	The end of the parameter block must be X'FFFFFFFFFFFFFFFF'.

Specifying values for the new environment

If you use IRXINIT to initialize a new language processor environment, the parameters you can specify on the call depend on:

- Whether the environment is being initialized in a non-TSO/E address space or in the TSO/E address space
- If the environment is being initialized in the TSO/E address space, whether the environment is to be integrated into TSO/E (TSOFL flag setting)

You can use many parameters only if the environment is initialized in a non-TSO/E address space or if the environment is initialized in TSO/E, but is not integrated into TSO/E (the TSOFL flag is off). Other parameters are intended only for use in the TSO/E address space where the environment is integrated into TSO/E (the TSOFL flag is on). The following information highlights different parameters. For more information about the values you can and cannot specify and various considerations for parameter values, see "Specifying values for different environments" on page 348.

When you call IRXINIT, you cannot specify the ID and VERSION. If you pass values for the ID or VERSION parameters, IRXINIT ignores the value and uses the default.

At offset +36 in the parameters module, the field is a fullword of bits that IRXINIT uses as flags. The flags define certain characteristics for the new language processor environment and how the environment and execs running in the environment operate. In addition to the flags field, the parameter following the flags is a mask field that works together with the flags. The mask field is a string that has the same length as the flags field. Each bit position in the mask field corresponds to a bit in the same position in the flags field. IRXINIT uses the mask field to determine whether it should use or ignore the corresponding flag bit.

The description of the mask field of "Characteristics of a Language Processor Environment" on page 317 describes the bit settings for the mask field in detail. Table 55 on page 321 summarizes each flag. "Flags and corresponding masks" on page 322 describes each of the flags in more detail and the bit settings for each flag.

For a given bit position, if the value in the mask field is:

- 0 – IRXINIT ignores the corresponding bit in the flags field (that is, IRXINIT considers the bit to be null)
- 1 – IRXINIT uses the corresponding bit in the flags field

When you call IRXINIT, the flag settings that IRXINIT uses depend on the:

- Bit settings in the flag and mask fields you pass in the in-storage parameter list
- Bit settings in the flag and mask fields you pass in the parameters module
- Flags defined for the previous environment
- Flags defined in IRXPARMS if a previous environment does not exist.

IRXINIT uses the following order to determine what value to use for *each* flag bit:

- IRXINIT first checks the mask setting in the in-storage parameter list. If the mask is 1, IRXINIT uses the flag value from the in-storage parameter list.

- If the mask in the in-storage parameter list is 0, IRXINIT then checks the mask setting in the parameters module. If the mask in the parameters module is 1, IRXINIT uses the flag value from the parameters module.
- If the mask in the parameters module is 0, IRXINIT uses the flag value defined for the previous environment.
- If a previous environment does not exist, IRXINIT uses the flag setting from IRXPARMS.

If you call IRXINIT to initialize an environment that is not integrated into TSO/E (the TSOFL flag is off), you can specify a subpool number (SUBPOOL field) from 0 – 127. IRXINIT does not check the number you provide. If the number is not 0 – 127, IRXINIT does not fail. However, when storage is used in the environment, an error occurs.

If you call IRXINIT to initialize an environment in the TSO/E address space and the environment is integrated into TSO/E, you must provide a subpool number of 78 (decimal). If the number is not 78, IRXINIT returns with a reason code of 7 in parameter 7.

For detailed information about the parameters you can specify for initializing a language processor environment, see "Specifying values for different environments" on page 348.

The end of the parameter block must be indicated by X'FFFFFFFFFFFFFFFF'.

Return specifications

For the IRXINIT initialization routine, the contents of the registers on return are:

Register 0

Contains the address of the new environment block if IRXINIT initialized a new environment, or the address of the environment block for the current non-reentrant environment that IRXINIT located.

If you called IRXINIT to initialize a new environment and IRXINIT could not initialize the environment, register 0 contains the same value as on entry. If you called IRXINIT to find an environment and IRXINIT could not locate the environment, register 0 contains a 0.

If IRXINIT returns with return code 100 or 104, register 0 contains the abend and reason code. "Return codes" on page 384 describes the return codes and how IRXINIT returns the abend and reason codes for return codes 100 and 104.

Register 1

Address of the parameter list.

IRXINIT uses three parameters (parameters 6, 7, and 9) for output only (see Table 68 on page 373). "Output parameters" on page 381 describes the three output parameters.

Registers 2-14

Same as on entry

Register 15

Return code

Output parameters

The parameter list for IRXINIT contains three parameters that IRXINIT uses for output only (parameters 6, 7, and 9). Parameter 6 contains the address of the environment block. If you called IRXINIT to locate an environment, parameter 6 contains the address of the environment block for the current non-reentrant environment. If you called IRXINIT to initialize an environment, parameter 6 contains the address of the environment block for the new environment. Parameter 6 lets high-level programming languages obtain the address of the environment block to examine information in the environment block.

Parameter 9 is an optional parameter you can use to obtain the return code. If you specify parameter 9, IRXINIT returns the return code in parameter 9 and also in register 15.

Parameter 7 contains a reason code for IRXINIT processing. The reason code indicates whether IRXINIT completed successfully. If IRXINIT processing was not successful, the reason code indicates the error. Table 70 on page 382 describes the reason codes IRXINIT returns. Note that these reason codes are not the same as the reason codes that are returned because of a system or user abend. A system or user abend results in a return code of 100 or 104 and an abend code and abend reason code in register 0. See "Return codes" on page 384 for a description of return codes 100 and 104.

If parameter 11 is specified and IRXINIT is returning with return code 20, the message buffer pointed to by parameter 11 will return a message associated with the error, usually containing the return code and reason code. If parameter 11 is not passed when IRXINIT completes with return code 20, the message is issued to the TSO/E user and/or to the CONSOLE. The caller of IRXINIT can set parm11 to point to a buffer with length header word X'00000000' to prevent IRXINIT from either issuing or returning a message when the IRXINIT return code is 20.

Table 70. Reason codes for IRXINIT processing

Reason code	Description
0	Successful processing.
1	Unsuccessful processing. The type of function to be performed (parameter 1) was not valid. The valid functions are INITENVB, FINDENVB, and CHEKENVB.
2	Unsuccessful processing. The TSOFL flag is on, but TSO/E is not active. IRXINIT evaluated all of the parameters for initializing the new environment. This reason code indicates that the environment is being initialized in a non-TSO/E address space, but the TSOFL flag is on. The TSOFL flag must be off for environments initialized in non-TSO/E address spaces.
3	Unsuccessful processing. A reentrant environment was specified for an environment that was being integrated into TSO/E. If you are initializing an environment in TSO/E and the TSOFL flag is on, the RENTRANT flag must be off. In this case, both the TSOFL and RENTRANT flags were on.
4	Unsuccessful processing. The environment being initialized was to be integrated into TSO/E (the TSOFL flag was on). However, a routine name was specified in the module name table that cannot be specified if the environment is being integrated into TSO/E. If the TSOFL flag is on, you can specify only the following routines in the module name table: • An attention exit (ATTNROUT field) • An exit for IRXEXEC (IRXEXECX field) • An exec initialization exit (EXECINIT field) • An exec termination exit (EXECTERM field).
5	Unsuccessful processing. The value specified in the GETFREER field in the module name table does not match the GETFREER value in the current language processor environment under the current task. If more than one environment is initialized on the same task and the environments specify a storage management replaceable routine (GETFREER field), the name of the routine must be the same for the environments.
6	Unsuccessful processing. The value specified for the length of each entry in the host command environment table is incorrect. This is the value specified in the SUBCOMTB_LENGTH field in the table. See "Host command environment table" on page 331 for information about the table.
7	Unsuccessful processing. An incorrect subpool number was specified for an environment being integrated into TSO/E. The subpool number must be 78 (decimal).
8	Unsuccessful processing. The TSOFL flag for the new environment is on. However, the flag in the previous environment is off. The TSOFL flag cannot be on if a previous environment in the chain has the TSOFL flag off.

Table 70. Reason codes for IRXINIT processing (continued)

Reason code	Description
9	Unsuccessful processing. The new environment specified that the data stack is to be shared (NEWSTKFL is off), but the SPSHARE flag in the previous environment is off, which means that storage is not to be shared across tasks. If you have the NEWSTKFL off for the new environment, you must ensure that the SPSHARE flag in the previous environment is on.
10	Unsuccessful processing. The IRXINITX exit routine returned a non-zero return code. IRXINIT stops initialization.
11	Unsuccessful processing. The IRXITTS exit routine returned a non-zero return code. IRXINIT stops initialization.
12	Unsuccessful processing. The IRXITMV exit routine returned a non-zero return code. IRXINIT stops initialization.
13	Unsuccessful processing. The REXX I/O routine or the replaceable I/O routine is called to initialize I/O when IRXINIT is initializing a new language processor environment. The I/O routine returned a non-zero return code.
14	Unsuccessful processing. The REXX data stack routine or the replaceable data stack routine is called to initialize the data stack when IRXINIT is initializing a new language processor environment. The data stack routine returned a non-zero return code.
15	Unsuccessful processing. The REXX exec load routine or the replaceable exec load routine is called to initialize exec loading when IRXINIT is initializing a new language processor environment. The exec load routine returned a non-zero return code.
16	Unsuccessful processing. REXX failed to initialize the TSO service facility command/program invocation platform. **Note:** A possible cause for this initialization failure may include, but is not restricted to, a GETMAIN failure where storage could not be obtained.
17	Unsuccessful processing. The ECT parameter, parameter 10, was not valid when initializing an environment that is integrated with TSO/E. The following are restrictions on the use of alternative ECTs (that is, ECTs other than the primary ECT created at either logon time or TMP initialization): **Note:** 1. When TSO/E processes an authorized command from a REXX exec and an alternate ECT is used, it is not possible for REXX to trap the command output from the authorized command. To use command output trapping by using the OUTTRAP function, the REXX exec must be executing on the primary ECT. 2. When TSO/E processes an authorized command from a REXX exec and an alternative ECT is being used, it is not possible for REXX to satisfy a prompt from the data stack, other than from the data stack associated with the language processor environment that is anchored in the primary ECT. That is, when TSO/E is processing an authorized command and that command prompts for input (by using the GETLINE or PUTGET service routines) the prompt can only be satisfied from the language processor environment anchored in the primary ECT.
20	Unsuccessful processing. Storage could not be obtained.
21	Unsuccessful processing. A module could not be loaded into storage. Refer to message IRX0901E for the name of this module. This message may be issued to the user and/or console, or it may be returned to the caller of IRXINIT in parm11.

Table 70. Reason codes for IRXINIT processing (continued)

Reason code	Description
22	Unsuccessful processing. The IRXINIT routine could not obtain serialization for a system resource.
23	Unsuccessful processing. A recovery ESTAE could not be established.
24	Unsuccessful processing. The maximum number of environments has already been initialized in the address space. The number of environments is defined in the environment table. See "Changing the maximum number of environments in an address space" on page 365 for more information about the environment table.
25	Unsuccessful processing. The extended parameter list (parameter 8) passed to IRXINIT was not valid. The end of the extended parameter list must be indicated by X'FFFFFFFFFFFFFFFF'.
26	Unsuccessful processing. The values specified in the extended parameter list (parameter 8) were incorrect. Either the address of the storage workarea or the length of the storage workarea was 0, or the length was a negative value. Reason code 26 is not returned if: • Both the address and length of the storage workarea are 0, which are valid values. • The address of the storage workarea is 0 and the length is -1, which is considered a valid null entry.
27	Unsuccessful processing. An incorrect number of parameters were passed to IRXINIT. IRXINIT returns reason code 27 if it cannot find the high-order bit on in the last address of the parameter list. In the parameter list, you must set the high-order bit on in either the address of parameter 7 or in the address of parameter 8 or parameter 9, which are optional parameters. **Note:** If you set the high-order bit on in a parameter before parameter 7, IRXINIT does not return reason code 27. The high-order bit indicates the end of the parameter list. Because IRXINIT detects the end of the parameter list before parameter 7, it cannot return a reason code because parameter 7 is the reason code parameter. In this case, IRXINIT returns only a return code of 20 in register 15 indicating an error.
28	Unsuccessful processing. An attempt was made to pass an 8th parameter, but the address of parameter 8 itself in the IRXINIT parameter list was zero. This is not valid. **Note:** An 8th parameter may be passed with a value of zero, but the address of the 8th parameter must not be zero. If the value of the 8th parameter (the extended parameter list ptr) is zero, REXX reserves a default amount of storage for the storage work area. If the address of that 8th parameter is zero, REXX sets reason code 28 and fails the initialization request.

Return codes

IRXINIT returns different return codes for finding an environment and for initializing an environment. IRXINIT returns the return code in register 15. If you specify the return code parameter (parameter 9), IRXINIT also returns the return code in the parameter.

Table 71 on page 385 shows the return codes if you call IRXINIT to find an environment.

Table 71. IRXINIT return codes for finding an environment

Return code	Description
0	Processing was successful. IRXINIT located the current non-reentrant environment. IRXINIT initialized the environment under the current task.
4	Processing was successful. IRXINIT located the current non-reentrant environment. IRXINIT initialized the environment under a previous task.
20	Processing was not successful. An error occurred. Check the reason code that IRXINIT returns in parameter 7.
28	Processing was successful. There is no current non-reentrant environment.
100	Processing was not successful. A system abend occurred while IRXINIT was locating the environment. The environment is not found. The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXINIT returns the abend code in the two low-order bytes of register 0. IRXINIT returns the abend reason code in the two high-order bytes of register 0. If the abend reason code is greater than two bytes, IRXINIT returns only the two low-order bytes of the abend reason code. See *z/OS MVS System Codes* for information about the abend codes and reason codes.
104	Processing was not successful. A user abend occurred while IRXINIT was locating the environment. The environment is not found. The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXINIT returns the abend code in the two low-order bytes of register 0. IRXINIT returns the abend reason code in the two high-order bytes of register 0. If the abend reason code is greater than two bytes, IRXINIT returns only the two low-order bytes of the abend reason code. See *z/OS MVS System Codes* for information about the abend codes and reason codes.

Table 72 on page 385 shows the return codes if you call IRXINIT to check an environment.

Table 72. IRXINIT return codes for checking an environment

Return code	Description
0	The environment block address provided in register 0 is an environment on the current task.
4	The environment block address provided in register 0 is an environment on a parent task.
8	The environment block address provided in register 0 is an environment in the address space, but not on the current or a parent task.
12	The environment block address provided in register 0 was not found in the address space. Parameter 6 contains the address of the current, non-reentrant environment block.
24	The environment table could not be located. The environment block address provided in register 0 could not be checked.

Table 73 on page 386 shows the return codes if you call IRXINIT to initialize an environment.

Table 73. IRXINIT return codes for initializing an environment

Return code	Description
0	Processing was successful. IRXINIT initialized a new language processor environment. The new environment is not the first environment under the current task.
4	Processing was successful. IRXINIT initialized a new language processor environment. The new environment is the first environment under the current task.
20	Processing was not successful. An error occurred. Check the reason code that IRXINIT returns in the parameter list.
100	Processing was not successful. A system abend occurred while IRXINIT was initializing the environment. The environment is not initialized. The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXINIT returns the abend code in the two low-order bytes of register 0. IRXINIT returns the abend reason code in the two high-order bytes of register 0. If the abend reason code is greater than two bytes, IRXINIT returns only the two low-order bytes of the abend reason code. See *z/OS MVS System Codes* for information about the abend codes and reason codes.
104	Processing was not successful. A user abend occurred while IRXINIT was initializing the environment. The environment is not initialized. The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXINIT returns the abend code in the two low-order bytes of register 0. IRXINIT returns the abend reason code in the two high-order bytes of register 0. If the abend reason code is greater than two bytes, IRXINIT returns only the two low-order bytes of the abend reason code. See *z/OS MVS System Codes* for information about the abend codes and reason codes.

Termination routine - IRXTERM

Use the IRXTERM routine to terminate a language processor environment.

Note: To permit FORTRAN programs to call IRXTERM, TSO/E provides an alternate entry point for the IRXTERM routine. The alternate entry point name is IRXTRM.

You can optionally pass the address of the environment block in register 0 that represents the environment you want terminated. IRXTERM then terminates the language processor environment pointed to by register 0. The environment must have been initialized on the current task.

If you do not specify an environment block address in register 0, IRXTERM locates the last environment that was created under the current task and terminates that environment.

When IRXTERM terminates the environment, IRXTERM closes all open data sets that were opened under that environment. IRXTERM also deletes any data stacks that you created under the environment using the NEWSTACK command.

IRXTERM does not terminate an environment under any one of the following conditions:

- The environment was not initialized under the current task
- An active exec is currently running in the environment
- The environment was the first environment initialized under the task and other environments are still initialized under the task.

The first environment initialized on a task must be the last environment terminated on that task. The first environment is the *anchor* environment because all subsequent environments that are initialized on the same task share information from the first environment. Therefore, all other environments on a task must be terminated before you terminate the first environment. If you use IRXTERM to terminate the first environment and other environments on the task still exist, IRXTERM does not terminate the environment and returns with a return code of 20.

Entry specifications

For the IRXTERM termination routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Registers 1-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

You can optionally pass the address of the environment block for the language processor environment you want to terminate in register 0. There is no parameter list for IRXTERM.

Return specifications

For the IRXTERM termination routine, the contents of the registers on return are:

Register 0
If you passed the address of an environment block, IRXTERM returns the address of the environment block for the previous environment. If you did not pass an address, register 0 contains the same value as on entry.

If IRXTERM returns with return code 100 or 104, register 0 contains the abend and reason code. "Return codes" on page 387 describes the return codes and how IRXTERM returns the abend and reason codes for return codes 100 and 104.

Registers 1-14
Same as on entry

Register 15
Return code

Return codes

Table 74 on page 387 shows the return codes for the IRXTERM routine.

Table 74. Return codes for IRXTERM

Return code	Description
0	IRXTERM successfully terminated the environment. The terminated environment was not the last environment on the task.
4	IRXTERM successfully terminated the environment. The terminated environment was the last environment on the task.
20	IRXTERM could not terminate the environment.
28	The environment could not be found.

Table 74. Return codes for IRXTERM (continued)

Return code	Description
100	A system abend occurred while IRXTERM was terminating the language processor environment. The system tries to terminate the environment again. If termination is still unsuccessful, the environment cannot be used. The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXTERM returns the abend code in the two low-order bytes of register 0. IRXTERM returns the abend reason code in the high-order two bytes of register 0. If the abend reason code is greater than two bytes, IRXTERM returns only the two low-order bytes of the abend reason code. See *z/OS MVS System Codes* for information about the abend codes and reason codes.
104	A user abend occurred while IRXTERM was terminating the language processor environment. The system tries to terminate the environment again. If termination is still unsuccessful, the environment cannot be used. The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXTERM returns the abend code in the two low-order bytes of register 0. IRXTERM returns the abend reason code in the two high-order bytes of register 0. If the abend reason code is greater than two bytes, IRXTERM returns only the two low-order bytes of the abend reason code. See *z/OS MVS System Codes* for information about the abend codes and reason codes.

Chapter 16. Replaceable routines and exits

When a REXX exec runs, different system services are used for obtaining and freeing storage, handling data stack requests, loading and freeing the exec, and I/O. TSO/E provides routines for these system services. The routines are called *replaceable routines* because you can provide your own routines that replace the system-supplied routines. You can provide your own routines for non-TSO/E address spaces. In the TSO/E address space, you can provide your own routines only if the language processor environment is initialized with the TSOFL flag off. The TSOFL flag (see "Flags and corresponding masks" on page 322) indicates whether the language processor environment is integrated with TSO/E services. "Types of environments - integrated and not integrated into TSO/E" on page 316 describes the two types of environments.

In addition to defining your own replaceable routines to replace the routines that TSO/E provides, you can use the interfaces as described in this chapter to call any of the TSO/E-supplied routines to perform system services. You can call the routines in any address space, that is, in any type of language processor environment. You can also write your own routine to perform a system service using the interfaces described for the routine. A program can then call your own routine in any address space to perform that particular service.

In addition to replaceable routines, TSO/E also provides several exits you can use to customize REXX processing. The exits let you customize the initialization and termination of language processor environments, exec processing itself, and attention interrupts. Unlike the replaceable routines that you can replace only in language processor environments that are *not* integrated into TSO/E, you can provide REXX exits in any type of environment (integrated and not integrated into TSO/E). One exception is the attention handling exit for attention interrupts. The exit applies only to TSO/E, so you can specify the exit only in an environment that is integrated into TSO/E.

This chapter describes each of the replaceable routines and the exits that TSO/E provides for REXX processing.

Replaceable Routines: If you provide a replaceable routine that will replace the system- supplied routine, your routine can perform some pre-processing and then call the system-supplied routine to actually perform the service request. If the replaceable routine you provide calls the system-supplied routine, your replaceable routine must act as a *filter* between the call to your routine and your routine calling the system-provided routine. Pre-processing can include checking the request for the specific service, changing the request, or terminating the request. Your routine can also perform the requested service itself and not call the system-supplied routine.

The routines that you can replace and the functions your routine must perform, if you replace the system-supplied routine, are summarized below. "Replaceable routines" on page 390 describes each routine in more detail.

Exec Load
Called to load an exec into storage and free an exec when the exec completes processing. The exec load routine is also called to determine whether an exec is currently loaded and to close a specified data set.

I/O
Called to read a record from or write a record to a specified ddname. The I/O routine is also called to open a specified DD. For example, the routine is called for the SAY and PULL instructions (if the environment is not integrated into TSO/E) and for the EXECIO command.

Host Command Environment
Called to process all host commands for a specific host command environment.

Data Stack
Called to handle any requests for data stack services.

Storage Management
Called to obtain and free storage.

User ID

Called to obtain the user ID. The USERID built-in function returns the result that the user ID routine obtains.

Message Identifier

Called to determine whether the message identifier (message ID) is displayed with a REXX error message.

Replaceable routines are defined on a language processor environment basis. You define the names of the routines in the module name table. To define your own replaceable routine to replace the system-supplied routine, you must do the following:

- Write the code for the routine. The individual topics in this chapter describe the interfaces to each replaceable routine.
- Define the routine name to a language processor environment. For environments that are initialized in non-TSO/E address spaces, you can provide your own IRXPARMS parameters module that IRXINIT uses instead of the default IRXPARMS module. In your module, specify the names of your replaceable routines. You can also call IRXINIT to initialize an environment and pass the name of your module name table that includes the names of your replaceable routines.

  In the TSO/E address space, you can call IRXINIT to initialize an environment and pass the name of your module name table that includes the names of the replaceable routines. When you call IRXINIT, the TSOFL flag in the parameters module must be off, so the environment is not integrated into TSO/E.

  "Changing the default values for initializing an environment" on page 344 describes how to provide your own parameters module. "Initialization routine - IRXINIT" on page 371 describes IRXINIT.

You can also call any of the system-supplied replaceable routines from a program to perform a system service. You can also write your own routine that user-written programs can call to perform a service. This chapter describes the interfaces to the system routines.

Exit Routines: In addition to the replaceable routines, there are several exits you can use to customize REXX processing. Some of the exits have fixed names. Other exits do not have a fixed name. You name the exit yourself and then specify the name in the module name table. The exits are briefly described below. "REXX exit routines" on page 426 describes each exit in more detail.

- Pre-environment initialization – use to customize processing before the IRXINIT initialization routine initializes a language processor environment.
- Post-environment initialization – use to customize processing after the IRXINIT initialization routine has initialized an environment, but before IRXINIT completes processing.
- Environment termination – use to customize processing when a language processor environment is terminated.
- Exec initialization – use to customize processing after the variable pool has been created and before the exec begins processing.
- Exec termination – use to customize processing after an exec completes processing and before the variable pool is deleted.
- Exec processing – use to customize exec processing before an exec is loaded and runs.
- Attention handling – use to customize attention interrupt processing in TSO/E.

Unlike the replaceable routines, which you can define only in language processor environments that are not integrated into TSO/E, you can provide the exits in any type of environment. One exception is the attention handling routine, which is only applicable to the TSO/E address space (in an environment that is integrated into TSO/E). See "REXX exit routines" on page 426 for more information about the exits.

Replaceable routines

The following topics describe each of the TSO/E REXX replaceable routines. The documentation describes how the system-supplied routines work, the input they receive, and the output they return. If you provide your own routine that replaces the system-supplied routine, your routine must handle all of the functions that the system-supplied routine handles.

The replaceable routines that TSO/E provides are programming routines that you can call from a program in any address space. The only requirement for invoking one of the system-supplied routines is that a language processor environment must exist in which the routine runs. The language processor environment can either be integrated or not integrated into TSO/E. For example, an application program can call the system-supplied data stack routine to perform data stack operations or call the I/O routine to perform I/O.

You can also write your own routines to handle different system services. For example, if you write your own exec load routine, a program can call your routine to load an exec before calling IRXEXEC to invoke the REXX exec. Similar to the system-supplied routines, if you write your own routine, an application program can call your routine in any address space as long as a language processor environment exists in which the routine can run. The environment can either be integrated or not integrated into TSO/E.

You could also write your own routine that application programs can call to perform a system service, and have your routine call the system- supplied routine. Your routine could act as a *filter* between the call to your routine and your routine calling the system-supplied routine. For example, you could write your own exec load routine that verifies a request, allocates a system load file, and then invokes the system-supplied exec load routine to actually load the exec.

General considerations

This topic provides general information about the replaceable routines.

- If you provide your own replaceable routine, your routine is called in 31 bit addressing mode. Your routine may perform the requested service itself and not call the system-supplied routine. Your routine can perform pre-processing, such as checking or changing the request or parameters, and then call the corresponding system-supplied routine. If your routine calls the system routine to actually perform the request, your routine must call the system routine in 31 bit addressing mode also.
- When the system calls your replaceable routine, your routine can use any of the system-supplied replaceable routines to request system services.
- The addresses of the system-supplied and any user-supplied replaceable routines are stored in the REXX vector of external entry points (see "Format of the REXX vector of external entry points" on page 362). This allows a caller external to REXX to call any of the replaceable routines, either the system-supplied or user-supplied routines. For example, if you want to preload a REXX exec in storage before using the IRXEXEC routine to invoke the exec, you can call the IRXLOAD routine to load the exec. IRXLOAD is the system-supplied exec load routine. If you provide your own exec load routine, you can also use your routine to preload the exec.
- When a replaceable routine is invoked by the system or by an application program, the contents of register 0 may or may not contain the address of the environment block. For more information, see "Using the environment block address" on page 391.

Using the environment block address

If you provide a user-supplied replaceable routine that replaces a system-supplied replaceable routine, when the system calls your routine, it passes the address of the environment block for the current environment in register 0. If your user-supplied routine then invokes the system-supplied routine, it is suggested that you pass the environment block address you received to the system-supplied routine. When you invoke the system-supplied routine, you can pass the environment block address in register 0. Some replaceable routines also have an optional environment block address parameter that you can use.

If your user-supplied routine passes the environment block address in the parameter list, the system-supplied routine uses the address you specify and ignores register 0. Additionally, the system-supplied routine does not validate the address you pass. Therefore, you must ensure that your user-supplied routine passes the same address it received in register 0 when it got control.

If your user-supplied routine does not specify an address in the environment block address parameter or the replaceable routine does not support the parameter, the system-supplied routine checks register 0 for the environment block address. If register 0 contains the address of a valid environment block, the system-supplied routine runs in that environment. If the address in register 0 is not valid, the system-supplied routine locates and runs in the current non-reentrant environment.

If your user-supplied routine does not pass the environment block address it received to the system-supplied routine, the system-supplied routine locates the current non-reentrant environment and runs in that environment. This may or may not be the environment in which you want the routine to run. Therefore, it is suggested that you pass the environment block address when your user-supplied routine invokes the system-supplied routine.

An application program running in any address space can call a system-supplied or user-supplied replaceable routine to perform a specific service. On the call, the application program can optionally pass the address of an environment block that represents the environment in which the routine runs. The application program can pass the environment block address in register 0 or in the environment block address parameter if the replaceable routine supports the parameter. Note the following for application programs that invoke replaceable routines:

- If an application program invokes a system-supplied replaceable routine and does not pass an environment block address, the system-supplied routine locates the current non-reentrant environment and runs in that environment.
- If an application program invokes a user-supplied routine, either the application program must provide the environment block address or the user-supplied routine must locate the current environment in which to run.

Installing replaceable routines

If you write your own replaceable routine, you must link-edit the routine as a separate load module. You can link-edit all your replaceable routines in a separate load library or in an existing library that contains other routines. The routines can reside in:

- The link pack area (LPA)
- Linklist (LNKLIST)
- A logon STEPLIB

The replaceable routines must be reentrant, refreshable, and reusable. The characteristics for the routines are:

- State: Problem program
- Not APF-authorized
- AMODE(31), RMODE(ANY)

Exec load routine

The system calls the exec load routine to load and free REXX execs. The system also calls the routine:

- To close any input files from which execs are loaded
- To check whether an exec is currently loaded in storage
- When a language processor environment is initialized and terminated

The name of the system-supplied exec load routine is IRXLOAD.

Tip: To permit FORTRAN programs to call IRXLOAD, TSO/E provides an alternate entry point for the IRXLOAD routine. The alternate entry point name is IRXLD.

When the exec load routine is called to load an exec, the routine reads the exec from the DD and places the exec into a data structure called the *in-storage control block* (INSTBLK). “Format of the in-storage control block” on page 400 describes the format of the in-storage control block. When the exec load routine is called to free an exec, the exec frees the storage that the previously loaded exec occupied.

The name of the exec load routine is specified in the EXROUT field in the module name table for a language processor environment. “Module name table” on page 326 describes the format of the module name table.

The system calls the exec load routine when:

- A language processor environment is initialized. During environment initialization, the exec load routine initializes the REXX exec load environment.
- The IRXEXEC routine is called and the exec is not preloaded. See "The IRXEXEC routine" on page 249 for information about using IRXEXEC.
- The exec that is currently running calls an external function or subroutine and the function or subroutine is an exec. (This is an internal call to the IRXEXEC routine.)
- An exec that was loaded needs to be freed.
- The language processor environment that originally opened the DD from which execs are loaded is terminating and all files associated with the environment must be closed.

The system-supplied load routine, IRXLOAD, tests for numbered records in the file. If the records of a file are numbered, the routine removes the numbers when it loads the exec. A record is considered to be numbered if:

- The record format of the file is variable and the first eight characters of the first record are numeric
- The record format of the file is fixed and the last eight characters of the first record are numeric

If the first record of the file is not numbered, the routine loads the exec without making any changes.

Any user-written program can call IRXLOAD to perform the functions that IRXLOAD supports. You can also write your own exec load routine and call the routine from an application program in any address space. For example, if you have an application program that calls the IRXEXEC routine to run a REXX exec, you may want to preload the exec into storage before calling IRXEXEC. To preload the exec, your application program can call IRXLOAD. The program can also call your own exec load routine.

TSO/E REXX tries to reuse a previously loaded exec if it appears that the exec has not been changed. Otherwise, EXECUTIL EXECDD (CLOSEDD) causes a new copy of the exec to be reloaded each time it is needed.

If you are writing an exec load routine that will be used in environments in which compiled REXX execs run, note that your exec load routine may want to invoke a compiler interface load routine. For information about the compiler interface load routine and when it can be invoked, see *z/OS TSO/E Customization*.

Entry specifications

For the exec load replaceable routine, the contents of the registers on entry are described below. The address of the environment block can be specified in either register 0 or in the environment block address parameter in the parameter list. For more information, see "Using the environment block address" on page 391.

Register 0
Address of the current environment block

Register 1
Address of the parameter list

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 75 on page 394 describes the parameters for the exec load routine.

Table 75. Parameters for the exec load routine

Parameter	Number of bytes	Description
Parameter 1	8	The function to be performed. The function name is left justified, in uppercase, and padded to the right with blanks. The valid functions are: • INIT • LOAD • TSOLOAD • FREE • STATUS • CLOSEDD • TERM The functions are described in "Functions you can specify for parameter 1" on page 396.
Parameter 2	4	Specifies the address of the exec block (EXECBLK). The exec block is a control block that describes the exec to be loaded (LOAD or TSOLOAD), to be checked (STATUS), or the DD to be closed (CLOSEDD). "Format of the exec block" on page 397 describes the exec block. For the LOAD, TSOLOAD, STATUS, and CLOSEDD functions, this parameter must contain a valid exec block address. For the other functions, this parameter is ignored.

Table 75. Parameters for the exec load routine (continued)

Parameter	Number of bytes	Description
Parameter 3	4	Specifies the address of the in-storage control block (INSTBLK), which defines the structure of a REXX exec in storage. The in-storage control block contains pointers to each record in the exec and the length of each record. “Format of the in-storage control block” on page 400 describes the control block. The exec load routine uses this parameter as an input parameter for the FREE function only. The routine uses the parameter as an output parameter for the LOAD, TSOLOAD, STATUS, and FREE functions. The parameter is ignored for the INIT, TERM, and CLOSEDD functions. As an input parameter for the FREE function, the parameter contains the address of the in-storage control block that represents the exec to be freed. As an output parameter for the FREE function, the parameter contains a 0 indicating the exec was freed. If the exec could not be freed, the return code in either register 15 or the return code parameter (parameter 5) indicates the error condition. “Return codes” on page 403 describes the return codes. As an output parameter for the LOAD, TSOLOAD, or STATUS functions, the parameter returns the address of the in-storage control block that represents the exec that was: • Just loaded (LOAD or TSOLOAD function) • Previously loaded (STATUS function) For the LOAD, TSOLOAD, and STATUS functions, the routine returns a value of 0 if the exec is not loaded.
Parameter 4	4	The address of the environment block that represents the environment in which you want the exec load replaceable routine to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, the exec load routine uses the value you specify and ignores register 0. However, the routine does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see “Using the environment block address” on page 391.

Table 75. Parameters for the exec load routine (continued)

Parameter	Number of bytes	Description
Parameter 5	4	A 4-byte field that the exec load replaceable routine uses to return the return code. The return code parameter is optional. If you use this parameter, the exec load routine returns the return code in the parameter and also in register 15. Otherwise, the routine uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 403 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

Functions you can specify for parameter 1

The functions that can be specified in parameter 1 are described below.

INIT

The routine performs any initialization that is required. During the initialization of a language processor environment, the system calls the exec load routine to initialize load processing.

LOAD

The routine loads the exec specified in the exec block from the ddname specified in the exec block. "Format of the exec block" on page 397 describes the exec block.

The routine returns the address of the in-storage control block (parameter 3) that represents the loaded exec. "Format of the in-storage control block" on page 400 shows the format of the in-storage control block.

Note: The TSO/E IRXLOAD routine reuses an existing copy of a previously loaded exec if one already exists in storage, and if it appears that the exec did not change since the exec was originally loaded. However, if the CLOSEXFL flag is on, indicating the SYSEXEC data set should be closed after each exec is loaded from SYSEXEC, IRXLOAD will not reuse a previously loaded exec copy to satisfy a load request. Instead, each load request results in a new copy of the exec being read into storage. For more information about the CLOSEXFL flag, see "Flags and corresponding masks" on page 322.

TSOLOAD

The routine loads the exec specified in the exec block from the current list of ddnames that TSO/E is using to search for REXX execs. For example, the routine may search load libraries, any exec libraries as defined by the TSO/E ALTLIB command, and SYSEXEC and SYSPROC. See "Search order" on page 74 for a description of the complete search order.

You can use the TSOLOAD function only in the TSO/E address space in a language processor environment that is integrated into TSO/E. TSOLOAD requires an environment that is integrated into TSO/E because TSOLOAD requests that the exec load routine use the current TSO/E search order to locate the exec.

The TSOLOAD function is intended for use if you call the system-supplied exec load routine (IRXLOAD) in TSO/E. TSOLOAD gives you the flexibility to search more than one DD to locate a REXX exec compared to the LOAD function, which only searches the DD specified in the exec block. You can also use the TSOLOAD function if you write your own exec load routine and then call your routine from application programs running in TSO/E.

TSOLOAD is not intended for language processor environments that are not integrated into TSO/E. Therefore, if you provide an exec load routine to replace the system- supplied exec load routine in the module name table, your routine that replaces the system routine need not handle the TSOLOAD request. This is because you can replace the system-supplied exec load routine only in environments that are not integrated into TSO/E.

For the TSOLOAD function, the exec load routine returns the:

- DD from which the exec was loaded. The routine returns the ddname in the exec block (at offset +24) that you provide on the call.
- Address of the in-storage control block in parameter 3 of the parameter list. The control block represents the loaded exec.

Note: The TSO/E IRXLOAD routine reuses an existing copy of a previously loaded exec if one already exists in storage, and if it appears that the exec did not change since the exec was originally loaded. However, if the CLOSEXFL flag is on, indicating the SYSEXEC data set should be closed after each exec is loaded from SYSEXEC, IRXLOAD will not reuse a previously loaded exec copy to satisfy a load request. Instead, each load request results in a new copy of the exec being read into storage. For more information about the CLOSEXFL flag, see "Flags and corresponding masks" on page 322.

FREE

The routine frees the exec represented by the in-storage control block that is pointed to by parameter 3.

Rule: If a user written load routine calls IRXLOAD to load an exec, the user written load routine must also call IRXLOAD to free the exec. If TSO/E REXX loads an exec, it also frees the exec. For information about the IRXEXEC routine, see "The IRXEXEC routine" on page 249.

STATUS

The routine determines whether the exec specified in the exec block is currently loaded in storage from the ddname specified in the exec block. If the exec is loaded, the routine returns the address of the in-storage control block in parameter 3. The address that the routine returns is the same address that was returned for the LOAD function when the routine originally loaded the exec into storage.

TERM

The routine performs any cleanup before termination of the language processor environment. When the last language processor environment under the task that originally opened the DD terminates, all files associated with the environment are closed. In TSO/E REXX, when terminating the last language processor environment under a task, IRXLOAD frees any execs that were loaded by any language processor environment under the task but which were not yet freed.

CLOSEDD

The routine closes the data set specified in the exec block, it does not free any execs that have been loaded from this data set.

The CLOSEDD function allows you to free and reallocate data sets. Only data sets that were opened on the current task can be closed.

Format of the exec block

The exec block (EXECBLK) is a control block that describes the:

- Exec to be loaded (LOAD or TSOLOAD function)
- Exec to be checked (STATUS function)
- DD to be closed (CLOSEDD function)

If a user-written program calls IRXLOAD or your own exec load routine, the program must build the exec block and pass the address of the exec block on the call. TSO/E provides a mapping macro, IRXEXECB, for the exec block. The mapping macro is in SYS1.MACLIB. Table 76 on page 398 describes the format of the exec block.

Table 76. Format of the exec block

Offset (decimal)	Number of bytes	Field name	Description
0	8	ACRYN	An eight-character field that identifies the exec block. The field must contain the character string 'IRXEXECB'.
8	4	LENGTH	Specifies the length of the exec block, in bytes.
12	4	—	Reserved.
16	8	MEMBER	Specifies the member name of the exec if the exec is in a partitioned data set. If the exec is in a sequential data set, this field is blank. For the TSOLOAD function, the member name is required.
24	8	DDNAME	For a LOAD request, the field specifies the ddname from which the exec is to be loaded. For a TSOLOAD request, this field is used only for output; it is ignored on input. On output, the field contains the ddname from which the exec was loaded. For a STATUS request, the field specifies the ddname from which the exec being checked was loaded. For a CLOSEDD request, the field specifies the ddname to be closed. An exec cannot be loaded from a DD that has not been allocated. The ddname specified must be allocated to a data set containing REXX execs or to a sequential data set that contains an exec. For the LOAD and STATUS functions, this field can be blank. In these cases, the ddname in the LOADDD field of the module name table is used.
32	8	SUBCOM	Specifies the name of the initial host command 4 environment when the exec starts running. If this field is blank, the environment specified in the INITIAL field of the host command environment table is used.
40	4	DSNPTR	Specifies the address of a data set name that the PARSE SOURCE instruction returns. The name usually represents the name of the exec load data set. The name can be up to 54 characters long (44 characters for the fully qualified data set name, 8 characters for the member name, and 2 characters for the left and right parentheses). The field can be blank. **Note:** For concatenated data sets, the field may contain the name of the first data set in the sequence, although the exec was loaded from a data set other than the first one in the sequence.
44	4	DSNLEN	Specifies the length of the data set name that is pointed to by the address at offset +40. The length can be 0-54. If no data set name is specified, the length is 0.

Table 76. Format of the exec block (continued)

Offset (decimal)	Number of bytes	Field name	Description
48	4	EXTNAME_PTR	Pointer to the extended execname. This field can be used to pass an execname if greater than eight characters. For example, this field may be used to pass *pathname/filename* of a UNIX file to a replaceable load routine that handles UNIX file system files. (This name is not used by the TSO/E REXX load routine.) (This field is only valid if PTF for APAR OW28404 is applied.)
52	4	EXTNAME_LEN	Specifies the length of the extended name pointed to by EXTNAME_PTR, or 0 if no extended name is specified. The maximum length of an extended name is 4096 (x'1000'). Any length larger than this maximum value should be treated as 0 (that is, as no extended name specified).
56	8	------	Reserved

An exec cannot be loaded from a data set that has not been allocated. The ddname specified (at offset +24) must be allocated to a data set containing REXX execs or to a sequential data set that contains an exec. The fields at offset +40 and +44 in the exec block are used only for input to the PARSE SOURCE instruction and are for informational purposes only.

For the LOAD and STATUS functions, if a ddname is not specified in the exec block (at offset +24), the routine uses the ddname in the LOADDD field in the module name table for the language processor environment. The environment block (ENVBLOCK) points to the PARMBLOCK, which contains the address of the module name table.

Loading execs using an extended exec name

With the PTF for APAR OW28404 applied, the replaceable Exec LOAD routine interface is enhanced to allow extended execnames, that is, exec names that are longer than eight characters or that are case sensitive. Note that the new fields, EXECBLK_EXTNAME_PTR and EXECBLK_EXTNAME_LEN in the EXECBLK, and INSTBLK_EXTNAME_PTR and INSTBLK_EXTNAME_LEN in the INSTBLK, can only be utilized if the PTF for OW28404 is applied. The following describes the extension to the LOAD interface when OW28404 is applied.

For a LOAD request, the caller may request an exec with a name longer than eight characters, or an exec with a name that is case sensitive. If so, that extended name may be specified in a field pointed to by the EXEC_EXTNAME_PTR field of the EXECBLK. A name in the MEMBER field may also be specified. It is optional whether the LOAD routine will handle an extended execname. The LOAD routine may use the extended name pointed to by EXEC_EXTNAME_PTR (if present), or it may ignore this name and continue to use the EXEC_MEMBER field as the name of the exec to be loaded.

You can use the EXEC_EXTNAME_PTR field as a way to pass a UNIX file system path name, or a case sensitive exec identifier to a replaceable LOAD routine written specifically to handle such LOADs. However, note that the TSO/E default LOAD routine (IRXLOAD) does not load execs specified by an extended name. It loads only from a sequential data set or a partitioned data set (PDS) specified by EXEC_DDNAME, with a PDS member specified by EXEC_MEMBER.

A replaceable LOAD routine can choose to ignore an extended name and use the MEMBER name instead, or it can choose to load using the extended name, if present.

If the LOAD routine loads an exec specified by an extended execname, it should obtain storage to hold a copy of the extended name, and return an INSTBLK with INSTBLK_EXTNAME_PTR pointing to the copy of the extended name and INSTBLK_EXTNAME_LEN set to its length. If an extended name is not returned (as with the TSO/E default LOAD routine), INSTBLK_EXTNAME_PTR and INSTBLK_EXTNAME_LEN should be set to 0.

When the load routine is called for a FREE request, it is responsible for freeing any INSTBLK and extended name storage it had obtained during the LOAD request.

Format of the in-storage control block

The in-storage control block defines the structure of an exec in storage. It contains pointers to each record in the exec and the length of each record.

The in-storage control block consists of a header and the records in the exec, which are arranged as a vector of address/length pairs. Table 77 on page 400 shows the format of the in-storage control block header. Table 78 on page 402 shows the format of the vector of records. TSO/E provides a mapping macro, IRXINSTB, for the in-storage control block. The mapping macro is in SYS1.MACLIB.

Table 77. Format of the in-storage control block header

Offset (decimal)	Number of bytes	Field name	Description
0	8	ACRONYM	An eight-character field that identifies the control block. The field must contain the characters ‘IRXINSTB’.
8	4	HDRLEN	Specifies the length of the in-storage control block header only. The value must be 128 bytes.
12	4	—	Reserved.
16	4	ADDRESS	Specifies the address of the vector of records. See Table 78 on page 402 for the format of the address/length pairs. If this field is 0, the exec contains no statements.
20	4	USERLEN	Specifies the length of the address/length vector of records in bytes. This is not the number of records. The value is the number of records multiplied by 8. If this field is 0, the exec contains no statements.

Table 77. Format of the in-storage control block header (continued)

Offset (decimal)	Number of bytes	Field name	Description
24	8	MEMBER	Specifies the name of the exec. This is the name of the member in the partitioned data set from which the exec was loaded. If the exec was loaded from a sequential data set, this field is blank. If the exec was loaded using an extended execname specification (as pointed to by EXTNAME_PTR) this field can be left blank. (See the EXTNAME_PTR field below.) The PARSE SOURCE instruction returns the folded member name in token3 of the PARSE SOURCE string. If this field is blank or if an extended execname is specified then the name that PARSE SOURCE returns in token3 is either: • a question mark (?), if no extended name is specified, or • the extended execname pointed to by EXTNAME_PTR, if specified. An extended name is not folded to uppercase within the PARSE SOURCE string. Any blanks in the extended name are changed to null characters (x'00') when the extended name is placed in the PARSE SOURCE string. Note: If EXTNAME_PTR and MEMBER are both specified, EXTNAME_PTR is used to build the PARSE SOURCE string token3.
32	8	DDNAME	Specifies the ddname that represents the exec load DD from which the exec was loaded.
40	8	SUBCOM	Specifies the name of the initial host command environment when the exec starts running.
48	4	—	Reserved.
52	4	DSNLEN	Specifies the length of the data set name that is specified at offset +56. If a data set name is not specified, this field is 0.
56	54	DSNAME	A 54-byte field that contains the name of the data set, if known, from which the exec was loaded. The name can be up to 54 characters long (44 characters for the fully qualified data set name, 8 characters for the member name, and 2 characters for the left and right parentheses).
110	2	—	Reserved.

Table 77. Format of the in-storage control block header (continued)

Offset (decimal)	Number of bytes	Field name	Description
112	4	EXTNAME_PTR	Pointer to the extended execname. The extended execname can be used instead of the MEMBER field to return the exec name of the loaded exec if the name is longer than eight characters or is case sensitive. For example, this field can be used to return the *pathname/filename* specification of an exec loaded from a UNIX file system file. If specified, the PARSE SOURCE instruction returns the name pointed to by this field, without folding to uppercase, instead of the MEMBER name. (Any blanks within an extended name are changed to null characters (x'00') when moved into the PARSE SOURCE string.) See the discussion of PARSE SOURCE under MEMBER field above. (Note: The extended execname is not currently used by the default TSO/E REXX load routine). (This field is valid only if PTF for OW28404 is applied.)
116	4	EXTNAME_LEN	Specifies the length of extended execname pointed to by EXTNAME_PTR, or 0 if no extended name is specified. The maximum length of an extended name is 4096 (x'1000'). If a length larger than the maximum value is specified, the extended name is ignored.
120	8	—	Reserved.

At offset +16 in the in-storage control block header, the field points to the vector of records that are in the exec. The records are arranged as a vector of address/length pairs. Table 78 on page 402 shows the format of the address/length pairs.

The addresses point to the text of the record to be processed. This can be one or more REXX clauses, parts of a clause that are continued with the REXX continuation character (the continuation character is a comma), or a combination of these. The address is the actual address of the record. The length is the length of the record in bytes.

Table 78. Vector of records for the in-storage control block

Offset (decimal)	Number of bytes	Field name	Description
0	4	STMT@	Address of record 1
4	4	STMTLEN	Length of record 1
8	4	STMT@	Address of record 2
12	4	STMTLEN	Length of record 2
16	4	STMT@	Address of record 3

Table 78. Vector of records for the in-storage control block (continued)

Offset (decimal)	Number of bytes	Field name	Description
20	4	STMTLEN	Length of record 3
x	4	STMT@	Address of record n
y	4	STMTLEN	Length of record n

Return specifications

For the exec load routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

Table 79 on page 403 shows the return codes for the exec load routine. The routine returns the return code in register 15. If you specify the return code parameter (parameter 5), the exec load routine also returns the return code in the parameter.

Table 79. Return codes for the exec load replaceable routine

Return code	Description
-3	The exec could not be located. The exec is not loaded.
0	Processing was successful. The requested function completed.
4	The specified exec is not currently loaded. A return code of 4 is used for the STATUS function only.
20	Processing was not successful. The requested function is not performed. A return code of 20 occurs if: • A ddname was not specified and was required (LOAD, STATUS, and CLOSEDD functions) • The TSOLOAD function was requested, but the current language processor environment is not integrated into TSO/E • The ddname was specified, but the DD has not been allocated • An error occurred during processing. The system also issues an error message that describes the error.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Input/Output routine

The input/output (I/O) replaceable routine is also called the read input/write output data routine. The system calls the I/O routine to:

- Read a record from a specified DD

- Write a record to a specified DD
- Open a DD

The DD must be allocated to either a sequential data set or a single member of a partitioned data set. The name of the system-supplied I/O routine is IRXINOUT.

Note:

1. The system supplied I/O routine does not support I/O on files allocated to data sets with spanned, track overflow, or undefined record formats.
2. To permit FORTRAN programs to call IRXINOUT, TSO/E provides an alternate entry point for the IRXINOUT routine. The alternate entry point name is IRXIO.

If a read is requested, the routine returns a pointer to the record that was read and the length of the record. If a write is requested, the caller provides a pointer to the record to be written and the length of the record. If an open is requested, the routine opens the file if the file is not yet open. The routine also returns a pointer to an area in storage containing information about the file. You can use the IRXDSIB mapping macro to map this area. The mapping macro is in SYS1.MACLIB.

You specify the name of the I/O routine in the IOROUT field in the module name table. “Module name table” on page 326 describes the format of the module name table. I/O processing is based on the QSAM access method.

The I/O routine is called for:

- Initialization. When IRXINIT initializes a language processor environment, the system calls the I/O replaceable routine to initialize I/O processing.
- Open, when:
  - You use the LINESIZE built-in function in an exec
  - Before the language processor does any output
- For input, when:
  - A PULL or a PARSE PULL instruction is processed, and the data stack is empty, and the language processor environment is not integrated into TSO/E (see “Types of environments - integrated and not integrated into TSO/E” on page 316).
  - A PARSE EXTERNAL instruction is processed in a language processor environment that is not integrated into TSO/E (see “Types of environments - integrated and not integrated into TSO/E” on page 316).
  - The EXECIO command is processed.
  - A program outside of REXX calls the I/O replaceable routine for input of a record.
- For output, when:
  - A SAY instruction is processed in a language processor environment that is not integrated into TSO/E (see “Types of environments - integrated and not integrated into TSO/E” on page 316).
  - Error messages must be written.
  - Trace (interactive debug facility) messages must be written.
  - A program outside of REXX calls the I/O replaceable routine for output of a record.
- Termination. When the system terminates a language processor environment, the I/O replaceable routine is called to cleanup I/O.

Entry specifications

For the I/O replaceable routine, the contents of the registers on entry are described below. The address of the environment block can be specified in either register 0 or in the environment block address parameter in the parameter list. For more information, see “Using the environment block address” on page 391.

Register 0
Address of the current environment block

Register 1
Address of the parameter list

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 80 on page 405 describes the parameters for the I/O routine.

Table 80. Input parameters for the I/O replaceable routine

Parameter	Number of bytes	Description
Parameter 1	8	The function to be performed. The function name is left justified, in uppercase, and padded to the right with blanks. The valid functions are: • INIT • OPENR • OPENW • OPENX • READ • READX • WRITE • TERM • CLOSE "Functions supported for the I/O routine" on page 406 describes the functions in more detail.
Parameter 2	4	Specifies the address of the record read, the record to be written, or the *data set information block*, which is an area in storage that contains information about the file (see "Data set information block" on page 409).
Parameter 3	4	Specifies the length of the data in the buffer pointed to by parameter 2. On output for an open request, parameter 3 contains the length of the data set information block. "Buffer and buffer length parameters" on page 408 describes the buffer and buffer length in more detail.

Table 80. Input parameters for the I/O replaceable routine (continued)

Parameter	Number of bytes	Description
Parameter 4	8	An eight character string that contains the name of a preallocated input or output DD. The DD must be either a sequential data set or a single member of a PDS. If a member of a PDS is to be used, the DD must be specifically allocated to the member of the PDS. If the input or output file is not sequential, the I/O routine returns a return code of 20.
Parameter 5	4	For a read operation, this parameter is used on output and specifies the absolute record number of the last logical record read. For a write to a DD that is opened for update, it can be used to provide a record number to verify the number of the record to be updated. Verification of the record number can be bypassed by specifying a 0. This parameter is not used for the INIT, OPENR, OPENW, OPENX, TERM, or CLOSE functions. See "Line number parameter" on page 409 for more information.
Parameter 6	4	The address of the environment block that represents the environment in which you want the I/O replaceable routine to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, the I/O routine uses the value you specify and ignores register 0. However, the routine does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Using the environment block address" on page 391.
Parameter 7	4	A 4-byte field that the I/O replaceable routine uses to return the return code. The return code parameter is optional. If you use this parameter, the I/O routine returns the return code in the parameter and also in register 15. Otherwise, the routine uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 411 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

Functions supported for the I/O routine

The function to be performed by the I/O routine is specified in parameter 1. The valid functions are described below.

INIT

The routine performs any initialization that is required. During the initialization of a language processor environment, the I/O routine is called to initialize I/O processing.

OPENR

The routine opens the specified DD for a read operation if the DD is not already open. The ddname is specified in parameter 4.

The I/O routine returns the address of the data set information block in parameter 2. It returns the length of the data set information block (DSIB) in parameter 3. "Data set information block" on page 409 describes the block in more detail.

The routine opens the specified DD for a write operation if the DD is not already open. The ddname is specified in parameter 4.

The I/O routine returns the address of the data set information block in parameter 2. It returns the length of the data set information block (DSIB) in parameter 3. "Data set information block" on page 409 describes the block in more detail.

OPENX

The routine opens the specified DD for an update operation if the DD is not already open. The ddname is specified in parameter 4.

The I/O routine returns the address of the data set information block in parameter 2. It returns the length of the data set information block (DSIB) in parameter 3. "Data set information block" on page 409 describes the block in more detail.

READ

The routine reads data from the DD specified in parameter 4. It returns the data in the buffer pointed to by the address in parameter 2. It also returns the number of the record that was read in the line number parameter (parameter 5).

The READ and READX functions are equivalent, except that the data set is opened differently. Subsequent read operations to the same data set can be done using either the READ or READX function because they do not reopen the data set.

If the data set to be read is closed, the routine opens it for input and then performs the read.

READX

The routine reads data from the DD specified in parameter 4. It returns the data in the buffer pointed to by the address in parameter 2. It also returns the number of the record that was read in the line number parameter (parameter 5).

If the data set to be read is closed, the routine opens it for update and then performs the read.

The READ and READX functions are equivalent, except that the data set is opened differently. Subsequent read operations to the same data set can be done using either the READ or READX function because they do not reopen the data set.

WRITE

The routine writes data from the specified buffer to the specified DD. The buffer is pointed to by the address in parameter 2 and the ddname is specified in parameter 4.

If the data set is closed, the routine first opens it for output and then writes the record. For sequential data sets, if the data set is allocated as OLD, the first record that is written after the data set is opened is written as record number 1. If a sequential data set is allocated as MOD, the record is added at the end of the data set.

Rule: MOD cannot be used to append data to a member of a PDS. You can use MOD only when appending information to a sequential data set. To append information to a member of a PDS, rewrite the member with the additional records added.

When a data set is opened for update, the WRITE function is used to rewrite the last record that was retrieved by the READ or READX function. You can optionally use the line number parameter (parameter 5) to ensure that the number of the record being updated agrees with the number of the last record that was read.

TERM

The routine performs cleanup and closes any opened data sets.

CLOSE

The routine closes the DD specified in parameter 4. The CLOSE function permits data sets to be freed and reallocated.

The CLOSE function is allowed only from the task under which the data set was opened. If CLOSE is requested from a different task, the request is ignored and a return code of 20 is returned.

Buffer and buffer length parameters

Parameter 2 specifies the address of a buffer and parameter 3 specifies the buffer length. These two parameters are not used for the INIT, TERM, and CLOSE functions.

On input for a WRITE function, the buffer address points to a buffer that contains the record to be written. The buffer length parameter specifies the length of the data to be written from the buffer. The caller must provide the buffer address and length.

For the WRITE function, if data is truncated during the write operation, the I/O routine returns the length of the data that was actually written in the buffer length parameter. A return code of 16 is also returned.

On output for a READ or READX function, the buffer address points to a buffer that contains the record that was read. The buffer length parameter specifies the length of the data being returned in the buffer.

For a READ or READX function, the I/O routine obtains the buffer needed to store the record. The caller must copy the data that is returned into its own storage before calling the I/O routine again for another request. The buffers are reused for subsequent I/O requests.

On output for an OPENR, OPENW, or OPENX function, the buffer address points to the *data set information block*, which is an area in storage that contains information about the file. The buffer length parameter returns the length of the data set information block (DSIB) whose address is being returned. "Data set information block" on page 409 describes the format of this area. TSO/E provides a mapping macro, IRXDSIB, that you can use to map the buffer area returned for an open request.

For an OPENR, OPENW, or OPENX function, all of the information in the data set information block does not have to be returned. The buffer length must be large enough for all of the information being returned about the file or unpredictable results can occur. The data set information block buffer must be large enough to contain the flags field and any fields that have been set, as indicated by the flags field (see "Data set information block" on page 409).

REXX does not check the content of the buffer for valid or printable characters. Any hexadecimal characters may be passed.

The buffers that the I/O routine returns are reserved for use by the environment block (ENVBLOCK) under which the original I/O request was made. The buffer should not be used again until:

- A subsequent I/O request is made for the same environment block, or
- The I/O routine is called to terminate the environment represented by the environment block (TERM function), in which case, the I/O buffers are freed and the storage is made available to the system.

Any replaceable I/O routine must conform to this procedure to ensure that the exec that is currently running accesses valid data.

If you provide your own replaceable I/O routines, your routine must support all of the functions that the system-supplied I/O routine performs. All open requests must open the specified file. However, for an open request, your replaceable I/O routine need only fill in the data set information block fields for the logical record length (LRECL) and its corresponding flag bit. These fields are DSIB_LRECL and DSIB_LRECL_FLAG. The language processor needs these two fields to determine the line length being used for its write operations. The language processor will format all of its output lines to the width that is specified by the LRECL field. Your routine can specify a LRECL (DSIB_LRECL field) of 0, which means that the language processor will format its output using a width of 80 characters, which is the default.

When the I/O routine is called with the TERM function, all buffers are freed.

Line number parameter

The line number parameter (parameter 5) is not used for the INIT, OPENR, OPENW, OPENX, TERM, or CLOSE functions. The parameter is used as an input parameter for the WRITE function and as an output parameter for the READ and READX functions.

If you are writing to a DD that is opened for update, you can use this parameter to verify the record being updated. The parameter must be either:

- A non-zero number that is checked against the record number of the last record that was read for update. This ensures that the correct record is updated. If the record numbers are identical, the record is updated. If not, the record is not written and a return code of 20 is returned.
- 0 - No record verification is done. The last record that was read is unconditionally updated.

If you are writing to a DD that is opened for output, the line number parameter is ignored.

On output for the READ or READX functions, the parameter returns the absolute record number of the last logical record that was read.

Data set information block

The data set information block is a control block that contains information about a file that the I/O replaceable routine opens. For an OPENR, OPENW, or OPENX function request, the I/O routine returns the address of the data set information block (DSIB) in parameter 2, and the length of the DSIB in parameter 3. TSO/E provides a mapping macro IRXDSIB you can use to map the block. The mapping macro is in SYS1.MACLIB.

Table 81 on page 409 shows the format of the control block.

Table 81. Format of the data set information block

Offset (decimal)	**Number of bytes**	**Field name**	**Description**
0	8	ID	An eight character string that identifies the information block. It contains the characters ‘IRXDSIB’.
8	2	LENGTH	The length of the data set information block.
10	2	—	Reserved.
12	8	DDNAME	An eight character string that specifies the ddname for which information is being returned. This is the DD that the I/O routine opened.
20	4	FLAGS	A fullword of bits that are used as flags. Only the first nine bits are used. The remaining bits are reserved. The flag bits indicate whether information is returned in the fields at offset +24 - offset +42. Each flag bit corresponds to one of the remaining fields in the control block. Information about how to use the flag bits and their corresponding fields is provided after the table.

Table 81. Format of the data set information block (continued)

Offset (decimal)	Number of bytes	Field name	Description
24	2	LRECL	The logical record length (LRECL) of the data set. This field is required. **Note:** The LRECL field and its corresponding flag bit (at offset +20) are the last required fields to be returned in the data set information block. The remaining fields are not required.
26	2	BLKSZ	The block size (BLKSIZE) of the data set.
28	2	DSORG	The data set organization (DSORG) of the data set. • ‘0200’ - Data set is partitioned. • ‘0300’ - Data set is partitioned and unmovable. • ‘4000’ - Data set is sequential. • ‘4100’ - Data set is sequential and unmovable.
30	2	RECFM	The record format (RECFM) of the data set. • ‘F ’ - Fixed • ‘FB’ - Fixed blocked • ‘V ’ - Variable • ‘VB’ - Variable blocked • 'VS' - Variable spanned • 'VX' - Variable blocked spanned • 'U' - Undefined.
32	4	GET_CNT	The total number of records read by the GET macro for this DCB.
36	4	PUT_CNT	The total number of records written by the PUT or PUTX macro for this DCB.
40	1	IO_MODE	The mode in which the DCB was opened. • ‘R’ - open for READ (uses GET macro) • ‘X’ - open for READX (update uses GET and PUTX macros) • ‘W’ - open for WRITE (uses PUT macro) • ‘L’ - open for exec load (uses READ macro)
41	1	CC	Carriage control information. • ‘A’ - ANSI carriage control • ‘M’ - machine carriage control • ‘ ’ - no carriage control
42	1	TRC	IBM 3800 Printing Subsystem character set control information. • ‘Y’ - character set control characters are present • ‘N’ - character set control characters are not present

Table 81. Format of the data set information block (continued)

Offset (decimal)	Number of bytes	Field name	Description
43	1	—	Reserved.
44	4	—	Reserved.

At offset +20 in the data set information block, there is a fullword of bits that are used as flags. Only the first nine bits are used. The remaining bits are reserved. The bits are used to indicate whether information is returned in each field in the control block starting at offset +24. A bit must be set on if its corresponding field is returning a value. If the bit is set off, its corresponding field is ignored.

The flag bits are:

- The LRECL flag. This bit must be on and the logical record length must be returned at offset +24. The logical record length is the only data set attribute that is required. The remaining eight attributes starting at offset +26 in the control block are optional.
- The BLKSIZE flag. This bit must be set on if you are returning the block size at offset +26.
- The DSORG flag. This bit must be set on if you are returning the data set organization at offset +28.
- The RECFM flag. This bit must be set on if you are returning the record format at offset +30.
- The GET flag. This bit must be set on if you are returning the total number of records read at offset +32.
- The PUT flag. This bit must be set on if you are returning the total number of records written at offset +36.
- The MODE flag. This bit must be set on if you are returning the mode in which the DCB was opened at offset +40.
- The CC flag. This bit must be set on if you are returning carriage control information at offset +41.
- The TRC flag. This bit must be set on if you are returning IBM 3800 Printing Subsystem character set control information at offset +42.

Return specifications

For the I/O routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

Table 82 on page 411 shows the return codes for the I/O routine. The routine returns the return code in register 15. If you specify the return code parameter (parameter 7), the I/O routine also returns the return code in the parameter.

Table 82. Return codes for the I/O replaceable routine

Return code	Description
0	Processing was successful. The requested function completed. For an OPENR, OPENW, or OPENX request, the DCB was successfully opened. The I/O routine returns the address of an area of storage that contains information about the file. The address is returned in the buffer address parameter (parameter 2). You can use the IRXDSIB mapping macro to map this area.

Table 82. Return codes for the I/O replaceable routine (continued)

Return code	Description
4	Processing was successful. For a READ, READX, or WRITE, the DCB was opened. For an OPENR, OPENW, or OPENX, the DCB was already open in the requested mode. The I/O routine returns the address of an area of storage that contains information about the file. The address is returned in the buffer address parameter (parameter 2). You can use the IRXDSIB mapping macro to map this area.
8	This return code is used only for a READ or READX function. Processing was successful. However, no record was read because the end-of-file (EOF) was reached.
12	An OPENR, OPENW, or OPENX request was issued and the DCB was already open, but not in the requested mode. The I/O routine returns the address of an area of storage that contains information about the file. The address is returned in the buffer address parameter (parameter 2). You can use the IRXDSIB mapping macro to map this area.
16	Output data was truncated for a write or update operation (WRITE function). The I/O routine returns the length of the data that was actually written in parameter 3.
20	Processing was not successful. The requested function is not performed. One possibility is that a ddname was not specified. An error message that describes the error is also issued.
24	Processing was not successful. During an OPENR, OPENX, READ, or READX function, an empty data set was found in a concatenation of data sets. The file was not successfully opened. The requested function is not performed.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Host command environment routine

The host command environment replaceable routine is called to process all *host commands* for a specific host command environment (see "Host commands and host command environments" on page 23 for the definition of "host commands"). A REXX exec may contain host commands to be processed. When the language processor processes an expression that it does not recognize as a keyword instruction or function, it evaluates the expression and then passes the string to the active host command environment. A specific environment is in effect when the command is processed. The host command environment table (SUBCOMTB table) is searched for the name of the active host command environment. The corresponding routine specified in the table is then called to process the string. For each valid host command environment, there is a corresponding routine that processes the command.

In an exec, you can use the ADDRESS instruction to route a command string to a specific host command environment and therefore to a specific host command environment replaceable routine.

The names of the routines that are called for each host command environment are specified in the ROUTINE field of the host command environment table. "Host command environment table" on page 331 describes the table.

You can provide your own replaceable routine for any one of the default environments provided. You can also define your own host command environment that handles certain types of "host commands" and provide a routine that processes the commands for that environment.

Entry specifications

For a host command environment routine, the contents of the registers on entry are described below. For more information about register 0, see "Using the environment block address" on page 391.

Register 0
Address of the current environment block

Register 1
Address of the parameter list

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. All parameters are passed on the call. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. Table 83 on page 413 describes the parameters for a host command environment replaceable routine.

Table 83. Parameters for a host command environment routine

Parameter	Number of bytes	Description
Parameter 1	8	The name of the host command environment that is to process the string. The name is left justified, in uppercase, and padded to the right with blanks.
Parameter 2	4	Specifies the address of the string to be processed. REXX does not check the contents of the string for valid or printable characters. Any characters can be passed to the routine. REXX obtains and frees the storage required to contain the string.
Parameter 3	4	Specifies the length of the string to be processed.
Parameter 4	4	Specifies the address of the user token. The user token is a 16-byte field in the SUBCOMTB table for the specific host command environment. "Host command environment table" on page 331 describes the user token field.

Table 83. Parameters for a host command environment routine (continued)

Parameter	Number of bytes	Description
Parameter 5	4	Contains the return code of the host command that was processed. This parameter is used only on output. The value is a signed binary number. After the host command environment replaceable routine returns the value, REXX converts it into a character representation of its equivalent decimal number. The result of this conversion is placed into the REXX special variable RC and is available to the exec that invoked the command. Positive binary numbers are represented as unsigned decimal numbers. Negative binary numbers are represented as signed decimal numbers. For example: • If the command's return code is X'FFFFFF3F', the special variable RC contains -193. • If the command's return code is X'0000000C', the special variable RC contains 12. If you provide your own host command environment routines, you should establish a standard for the return codes that your routine issues and the contents of this parameter. If a standard is used, execs that issue commands to a particular host command environment can check for errors in command processing using consistent REXX instructions. With the host command environments that TSO/E provides, a return code of -3 in the REXX special variable RC indicates the environment could not locate the host command. The -3 return code is a standard return code for host commands that could not be processed. If your routine processes an invalid command, it is suggested that you return X'FFFFFFFD' as the return code, which means the REXX special variable RC will contain a -3.

Error recovery

When the host command environment routine is called, an error recovery routine (ESTAE) is in effect. The one exception is if the language processor environment was initialized with the NOESTAE flag set on. In this case, an ESTAE is not in effect unless the host command environment replaceable routine establishes its own ESTAE.

Unless the replaceable routine establishes its own ESTAE, REXX traps all abends that occur. This includes abends that occur in any routines that are loaded by the host command environment replaceable routine to process the command to be executed. If an abend occurs and the host command environment routine has not established a new level of ESTAE, REXX:

- Issues message IRX0250E if a system abend occurred or message IRX0251E if a user abend occurred
- Issues message IRX0255E

The language processor is restarted with a FAILURE condition enabled. See Chapter 7, “Conditions and condition traps,” on page 181 for information about conditions and condition traps. The special variable RC will be set to the decimal equivalent of the abend code as described in Table 83 on page 413 for the return code parameter (parameter 5).

Return specifications

For a host command environment routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

Table 84 on page 415 shows the return codes for the host command environment routine. These are the return codes from the replaceable routine itself, not from the command that the routine processed. The command's return code is passed back in parameter 5. See Chapter 7, “Conditions and condition traps,” on page 181 for information about ERROR and FAILURE conditions and condition traps.

Table 84. Return codes for the host command environment routine

Return code	Description
Less than or equal to -13	If the value of the return code is -13 or less than -13, the routine requested that the HOSTFAIL flag be turned on. This is a TRACE NEGATIVE condition and a FAILURE condition is trapped in the exec.
-1 – -12	If the value of the return code is from -1 to -12 inclusive, the routine requested that the HOSTERR flag be turned on. This is a TRACE ERROR condition and an ERROR condition is trapped in the exec.
0	No error condition was indicated by the routine. No error conditions are trapped (for example, to indicate a TRACE condition).
1 – 12	If the value of the return code is 1 - 12 inclusive, the routine requested that the HOSTERR flag be turned on. This is a TRACE ERROR condition and an ERROR condition is trapped in the exec.
Greater than or equal to 13	If the value of the return code is 13 or greater than 13, the routine requested that the HOSTFAIL flag be turned on. This is a TRACE NEGATIVE condition and a FAILURE condition is trapped in the exec.

Data stack routine

The data stack routine is called to handle any requests for data stack services. The routine is called when an exec wants to perform a data stack operation or when a program needs to process data stack-related operations. The routine is called for the following:

- PUSH
- PULL
- QUEUE
- QUEUED()
- MAKEBUF
- DROPBUF
- NEWSTACK
- DELSTACK
- QSTACK
- QBUF
- QELEM
- MARKTERM
- DROPTERM

The name of the system-supplied data stack routine is IRXSTK. If you provide your own data stack routine, your routine can handle all of the data stack requests or your routine can perform pre-processing and then call the system routine, IRXSTK. If your routine handles the data stack requests without calling the system-supplied routine, your routine must manipulate its own data stack.

If your data stack routine performs pre-processing and then calls the system routine IRXSTK, your routine must pass the address of the environment block for the language processor environment to IRXSTK.

An application running in any address space can invoke IRXSTK to operate on the data stack. The only requirement is that a language processor environment has been initialized.

Parameter 1 indicates the type of function to be performed against the data stack. If the data stack routine is called to pull an element off the data stack (PULL function) and the data stack is empty, a return code of 4 indicates an empty data stack. However, you can use the PULLEXTR function to bypass the data stack and read from the input stream (for example, from the terminal in TSO/E foreground).

If the data stack routine is called and a data stack is not available, all services operate as if the data stack were empty. A PUSH or QUEUE will seem to work, but the pushed or queued data is lost. QSTACK returns a 0. NEWSTACK will seem to work, but a new data stack will not be created and any subsequent data stack functions will operate as if the data stack is permanently empty.

The maximum string that can be placed on the data stack is one byte less than 16 MB. REXX does not check the content of the string, so the string can contain any hexadecimal characters.

If multiple data stacks are associated with a single language processor environment, all data stack operations are performed on the last data stack that was created under the environment. If a language processor environment is initialized with the NOSTKFL flag off, a data stack is always available to execs that run in that environment. The language processor environment might not have its own data stack. The environment might share the data stack with its parent environment depending on the setting of the NEWSTKFL flag when the environment is initialized.

If the NEWSTKFL flag is on, a new data stack is initialized for the new environment. If the NEWSTKFL flag is off and a previous environment on the chain of environments was initialized with a data stack, the new environment shares the data stack with the previous environment on the chain. “Using the data stack in different environments” on page 366 describes how the data stack is shared between language processor environments.

The name of the data stack replaceable routine is specified in the STACKRT field in the module name table. “Module name table” on page 326 describes the format of the module name table.

Entry specifications

For the data stack replaceable routine, the contents of the registers on entry are described below. The address of the environment block can be specified in either register 0 or in the environment block address parameter in the parameter list. For more information, see “Using the environment block address” on page 391.

Register 0
Address of the current environment block

Register 1
Address of the parameter list

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 85 on page 417 describes the parameters for the data stack routine.

Table 85. Parameters for the data stack routine

Parameter	Number of bytes	Description

Parameter 1	8	The function to be performed. The function name is left justified, in uppercase, and padded to the right with blanks. The valid functions are:
		`PUSH          PULL`
		`QUEUE         PULLEXTR`
		`MAKEBUF       QUEUED`
		`NEWSTACK      DROPBUF`
		`QSTACK        DELSTACK`
		`QELEM         QBUF`
		`DROPTERM      MARKTERM`
		"Functions supported for the data stack routine" on page 418 describes the functions in more detail.

Parameter 2	4	The address of a fullword in storage that points to a data stack element, a parameter string, or a fullword of zeros. The use of this parameter depends on the function requested. If the function is DROPBUF, the parameter points to a character string containing the number of the data stack buffer from which to start deleting data stack elements. If the function is a function that places an element on the data stack (for example, PUSH), the address points to a string of bytes that the caller wants to place on the data stack. There are no restrictions on the string. The string can contain any combination of hexadecimal characters. For PULL and PULLEXTR, this parameter is not used on input. On output, it specifies the address of the string that was returned. For PULL, the string was pulled from the data stack. For PULLEXTR, the string was read from the input stream, for example, the terminal or the SYSTSIN file. It is suggested that you do not change the original string and that you copy the original string into your own dynamic storage. In addition, the original string will no longer be valid when another data stack operation is performed.
Parameter 3	4	The length of the string pointed to by the address in parameter 2.
Parameter 4	4	A fullword binary number into which the result from the call is stored. The value is the result of the function performed and is valid only when the return code from the routine is 0. For more information about the results that can be returned in parameter 4, see the descriptions of the supported functions below and the individual descriptions of the data stack commands in this book.

Table 85. Parameters for the data stack routine (continued)

Parameter	Number of bytes	Description
Parameter 5	4	The address of the environment block that represents the environment in which you want the data stack replaceable routine to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, the data stack routine uses the value you specify and ignores register 0. However, the routine does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Using the environment block address" on page 391.
Parameter 6	4	A 4-byte field that the data stack replaceable routine uses to return the return code. The return code parameter is optional. If you use this parameter, the data stack routine returns the return code in the parameter and also in register 15. Otherwise, the routine uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 420 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

Functions supported for the data stack routine

The function to be performed by the data stack routine is passed in parameter 1. The valid functions are described below. The functions operate on the currently active data stack. For more information about each of the functions, see the individual descriptions of the corresponding data stack commands in this book.

PUSH
Adds an element to the top of the data stack.

PULL
Retrieves an element off the top of the data stack.

PULLEXTR
Bypasses the data stack and reads a string from the input stream. In TSO/E foreground, PULLEXTR reads from the terminal. In TSO/E background, PULLEXTR reads from SYSTSIN. In non-TSO/E address spaces, the PULLEXTR function reads from the input stream as defined by the INDD field in the module name table. The default is SYSTSIN.

PULLEXTR is useful if the data stack is empty or you want to bypass the data stack entirely. For example, suppose you use the PULL function and the data stack routine returns with a return code of 4, which indicates that the data stack is empty. You can then use the PULLEXTR function to read a string from the input stream.

QUEUE
Adds an element at the logical bottom of the data stack. If there is a buffer on the data stack, the element is placed immediately above the buffer.

QUEUED
Returns the number of elements on the data stack, not including buffers.

MAKEBUF

Places a buffer on the top of the data stack. The return code from the data stack routine is the number of the new buffer. The data stack initially contains one buffer (buffer 0), but MAKEBUF can be used to create additional buffers on the data stack. The first time MAKEBUF is issued for a data stack, the value 1 is returned.

DROPBUF n

Removes all elements from the data stack starting from the “n”th buffer. All elements that are removed are lost. If *n* is not specified, the last buffer that was created and all subsequent elements that were added are deleted.

For example, if MAKEBUF was issued six times (that is, the last return code from the MAKEBUF function was 6), and the command

```
DROPBUF 2
```

is issued, five buffers are deleted. These are buffers 2, 3, 4, 5, and 6.

DROPBUF 0 removes everything from the currently active data stack.

NEWSTACK

Creates a new data stack. The previously active data stack can no longer be accessed until a DELSTACK is issued.

DELSTACK

Deletes the currently active data stack. All elements on the data stack are lost. If the active data stack is the primary data stack (that is, only one data stack exists and a NEWSTACK was not issued), all elements on the data stack are deleted, but the data stack is still operational.

QSTACK

Returns the number of data stacks that are available to the running REXX exec.

QBUF

Returns the number of buffers on the active data stack. If the data stack contains no buffers, a 0 is returned.

QELEM

Returns the number of elements from the top of the data stack to the next buffer. If QBUF = 0, then QELEM = 0.

MARKTERM

Marks the top of the active data stack with the equivalent of a TSO/E terminal element, which is an element for the TSO/E input stack. The data stack now functions as if it were just initialized. The previous data stack elements cannot be accessed until a DROPTERM is issued. If you issue a MARKTERM, you must issue a corresponding DROPTERM to delete the terminal element that MARKTERM created.

MARKTERM is available only to calling programs to put a terminal element on the data stack. It is not available to REXX execs.

DROPTERM

Removes all data stack elements that were added after a MARKTERM was issued, including the terminal element created by MARKTERM. The data stack status is restored to the same status before the MARKTERM.

DROPTERM is available only to calling programs to remove a terminal element from the data stack. It is not available to REXX execs.

Return specifications

For the data stack routine, the contents of the registers on return are:

Registers 0-14

Same as on entry

Register 15
Return code

Return codes

Table 86 on page 420 shows the return codes for the data stack routine. These are the return codes from the routine itself. They are not the return codes from any of the TSO/E REXX commands, such as NEWSTACK, DELSTACK, and QBUF that are issued. The command's return code is placed into the REXX special variable RC, which the exec can retrieve.

The data stack routine returns the return code in register 15. If you specify the return code parameter (parameter 6), the routine also returns the return code in the parameter.

Table 86. Return codes for the data stack replaceable routine

Return code	Description
0	Processing was successful. The requested function completed.
4	The data stack is empty. A return code of 4 is used only for the PULL function.
8	A terminal marker, created by the MARKTERM function, was not on the active data stack. A return code of 8 is used only for the DROPTERM function.
20	Processing was not successful. An error condition occurred. The requested function is not performed. An error message describing the error may be issued. If there is no error message, REXX may have been invoked authorized. You cannot invoke a REXX exec or REXX service as authorized in either TSO/E foreground or background.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Storage management routine

REXX storage routines handle storage and have pools of storage available to satisfy storage requests for REXX processing. If the pools of storage available to the REXX storage routines are depleted, the routines then call the storage management routine to request a storage pool. A storage pool is contiguous storage that can be used by the REXX storage routines to satisfy storage requests for REXX processing.

You can provide your own storage management routine that interfaces with the REXX storage routines. If you provide your own storage management routine, when the pools of storage are depleted, the REXX storage routines will call your storage management routine for a storage pool. If you do not provide your own storage management routine, GETMAIN and FREEMAIN are used to handle storage pool requests. Providing your own storage management routine gives you an alternative to the system using GETMAIN and FREEMAIN.

The storage management routine is called to obtain or free a storage pool for REXX processing. The routine supplies a storage pool that is then managed by the REXX storage routines.

The storage management routine is called when:

- REXX processing requests storage and a sufficient amount of storage is not available in the pools of storage the REXX storage routines use.
- A storage pool needs to be freed. A storage pool may need to be freed when a language processor environment is terminated or when the REXX storage routines determine that a particular pool of storage can be freed.

Specify the name of the storage management routine in the GETFREER field in the module name table. “Module name table” on page 326 describes the format of the module name table.

Entry specifications

For the storage management replaceable routine, the contents of the registers on entry are described below. For more information about register 0, see “Using the environment block address” on page 391.

Register 0
Address of the current environment block

Register 1
Address of the parameter list

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. All parameters are passed on the call. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. Table 87 on page 421 describes the parameters for the storage management routine.

Table 87. Parameters for the storage management replaceable routine

Parameter	Number of bytes	Description
Parameter 1	8	The function to be performed. The name is left justified, in uppercase, and padded to the right with blanks. The following functions are valid: **GET** Obtain a storage pool above 16 MB in virtual storage **GETLOW** Obtain a storage pool below 16 MB in virtual storage **FREE** Free a storage pool
Parameter 2	4	Specifies the address of a storage pool. This parameter is required as an input parameter for the FREE function. It specifies the address of the storage pool the routine should free. This parameter is used as an output parameter for the GET and GETLOW functions. The parameter specifies the address of the storage pool the routine obtained.

Table 87. Parameters for the storage management replaceable routine (continued)

Parameter	Number of bytes	Description
Parameter 3	4	Specifies the size of the storage pool, in bytes, to be freed or that was obtained. On input for the FREE function, this specifies the size of the storage pool to be freed. This is the size of the storage pool pointed to by parameter 2. All requests for the FREE function will be for a single storage pool that was previously obtained using either GET or GETLOW. On output for the GET and GETLOW functions, the parameter specifies the size of the storage pool the routine obtained. The size obtained must be at least the size that was requested in parameter 4. The TSO/E storage routines will use the size returned in parameter 3.
Parameter 4	4	Specifies the size of the storage pool, in bytes, to be obtained. This parameter is used as an input parameter for the GET and GETLOW functions. It specifies the size of the storage pool that is being requested. The size of the storage pool that is actually obtained is returned in parameter 3. This parameter is not used for the FREE function.
Parameter 5	4	Specifies the subpool number from which the storage pool should be obtained. This parameter is used as input for all functions.

Return specifications

For the storage management replaceable routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return codes

Table 88 on page 422 shows the return codes for the storage management routine.

Table 88. Return codes for the storage management replaceable routine

Return code	Description
0	Processing was successful. The requested function completed.
20	Processing was not successful. An error condition occurred. Storage was not obtained or freed.

User ID routine

The user ID routine returns the same value as the USERID built-in function. The system calls the user ID replaceable routine whenever the USERID built-in function is issued in a language processor environment that is not integrated into TSO/E. The routine then returns either the user ID, stepname, or jobname. The name of the system-supplied user ID routine is IRXUID.

The name of the user ID replaceable routine is specified in the IDROUT field in the module name table. “Module name table” on page 326 describes the format of the module name table.

Entry specifications

For the user ID replaceable routine, the contents of the registers on entry are described below. The address of the environment block can be specified in either register 0 or in the environment block address parameter in the parameter list. For more information, see "Using the environment block address" on page 391.

Register 0
Address of the current environment block

Register 1
Address of the parameter list

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

Register 1 contains the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 89 on page 423 describes the parameters for the user ID routine.

Table 89. Parameters for the user ID replaceable routine

Parameter	Number of bytes	Description
Parameter 1	8	The function to be performed. The function name is left justified, in uppercase, and padded to the right with blanks. The valid functions are USERID and TSOID. "Functions supported for the user ID routine" on page 424 describes the functions in detail.
Parameter 2	4	An address of storage into which the routine places the user ID. On output, the area that this address points to contains a character representation of the user ID.
Parameter 3	4	The length of storage pointed to by the address in parameter 2. On input, this value is the maximum length of the area that is available to contain the ID. The length supplied is 160 bytes. The routine must change this parameter and return the actual length of the character string it returns. If the routine returns a 0, the USERID built-in function returns a null value. If the routine copies more characters into the storage area than the storage provided, REXX may abend and any results will be unpredictable.

Table 89. Parameters for the user ID replaceable routine (continued)

Parameter	Number of bytes	Description
Parameter 4	4	The address of the environment block that represents the environment in which you want the user ID replaceable routine to run. This parameter is optional. If you specify a non-zero value for the environment block address parameter, the user ID routine uses the value you specify and ignores register 0. However, the routine does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. For more information, see "Using the environment block address" on page 391.
Parameter 5	4	A 4-byte field that the user ID replaceable routine uses to return the return code. The return code parameter is optional. If you use this parameter, the user ID routine returns the return code in the parameter and also in register 15. Otherwise, the routine uses register 15 only. If the parameter list is invalid, the return code is returned in register 15 only. "Return codes" on page 425 describes the return codes. If you do not want to use the return code parameter, you can end the parameter list at a preceding parameter. Set the high-order bit on in the preceding parameter's address. For more information about parameter lists, see "Parameter lists for TSO/E REXX routines" on page 242.

Functions supported for the user ID routine

The function to be performed by the user ID routine is specified in parameter 1. The valid functions are described below.

USERID

Returns the same value that the USERID built-in function would return in an environment that is not integrated into TSO/E. The value returned may be a user ID, a stepname, or a jobname. You can use the USERID function only in environments that are not integrated into TSO/E.

TSOID

Returns the same value that the USERID built-in function would return in an environment that is integrated into TSO/E. The value returned is the TSO/E user ID. You can use the TSOID function only in a TSO/E address space in an environment that is integrated into TSO/E.

The TSOID function is intended for use if an application program calls the user ID routine, IRXUID, in a language processor environment that is integrated into TSO/E to obtain the user ID. You can also use the TSOID function if you write your own user ID routine and then call your routine from application programs running in environments that are integrated into TSO/E.

TSOID is intended only for language processor environments that are integrated into TSO/E. Because you can replace the user ID routine only in environments that are not integrated into TSO/E, your replaceable routine does not have to support the TSOID function.

Return specifications

For the user ID replaceable routine, the contents of the registers on return are:

Registers 0-14

Same as on entry

Register 15
Return code

Return codes

Table 90 on page 425 shows the return codes for the user ID routine. The routine returns the return code in register 15. If you specify the return code parameter (parameter 5), the user ID routine also returns the return code in the parameter.

Table 90. Return codes for the user ID replaceable routine

Return code	Description
0	Processing was successful. The user ID was returned or a null character string was returned.
20	Processing was not successful. Either parameter 1 (function) was not valid or parameter 3 (length) was less than or equal to 0. The user ID was not obtained.
28	Processing was not successful. The language processor environment could not be located.
32	Processing was not successful. The parameter list is not valid. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the parameter list is not set to 1 to indicate the end of the parameter list.

Message identifier routine

The message identifier replaceable routine is called to determine if the message identifier (message ID) is to be displayed with an error message. The name of the system-supplied message identifier routine is IRXMSGID.

Tip: To permit FORTRAN programs to call IRXMSGID, TSO/E provides an alternate entry point for the IRXMSGID routine. The alternate entry point name is IRXMID.

The routine is called whenever a message is to be written when a REXX exec or REXX routine (for example, IRXEXCOM or IRXIC) is running in:

- A non-TSO/E address space, or
- The TSO/E address space in a language processor environment that was not integrated into TSO/E (the TSOFL flag is off)

The name of the message identifier replaceable routine is specified in the MSGIDRT field in the module name table. “Module name table” on page 326 describes the format of the module name table.

Entry specifications

For the message identifier routine, the contents of the registers on entry are described below. For more information about register 0, see “Using the environment block address” on page 391.

Register 0
Address of the current environment block

Registers 1-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

There is no parameter list for the message identifier routine. Return codes are used to return information to the caller.

Return specifications

For the message identifier replaceable routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return Codes

Table 91 on page 426 shows the return codes for the message identifier routine.

Table 91. Return codes for the message identifier replaceable routine

Return code	Description
0	Display the message identifier (message ID) with the message.
Non-zero	Do not display the message identifier (message ID) with the message.

REXX exit routines

There are many exit routines you can use to customize REXX processing. The exits differ from other exit routines that TSO/E provides, such as exits for TSO/E command processors. Some of the REXX exits have fixed names while others you name yourself. Several exits receive parameters on entry and others receive no parameters.

Generally, you use exit routines to customize a particular command or function on a system-wide basis. You use the REXX exits to customize different aspects of REXX processing on a language processor environment basis. The following highlights the exits you can use for REXX. *z/OS TSO/E Customization* describes the exits in more detail. However, many of the exits receive the parameters that a caller passed on a call to a REXX routine, such as IRXINIT and IRXEXEC. Therefore, you will need to use *z/OS TSO/E Customization* and this book for complete information.

Some of the REXX exits do not have fixed names. You supply the name yourself and then define the name in the appropriate fields in the module name table. In the module name table, you also define the names of replaceable routines you provide. However, unlike the replaceable routines, which you can provide only in language processor environments that are not integrated into TSO/E, you can use the REXX exits in any type of environment (integrated and not integrated into TSO/E). One exception is the attention handling exit, which is available only in TSO/E (in an environment that is integrated into TSO/E).

Exits for Language Processor Environment initialization and termination

There are four exits you can use to customize the initialization and termination of language processor environments in any address space. The names of the four exits are fixed. If you provide one or more of these exits, the exit is invoked whenever the IRXINIT and IRXTERM routines are called. The exits are invoked whenever a user explicitly calls IRXINIT and IRXTERM or when the system automatically calls the routines to initialize and terminate a language processor environment. The exits are briefly described below. *z/OS TSO/E Customization* provides more information about each exit. Chapter 15, "Initialization and termination routines," on page 371 describes the IRXINIT and IRXTERM routines and their parameters.

IRXINITX

This is the pre-environment initialization exit routine. The exit is invoked whenever the initialization routine IRXINIT is called to initialize a new language processor environment. The exit receives control before IRXINIT evaluates any parameters to use to initialize the environment. The exit routine receives the same parameters that IRXINIT receives.

You can provide a pre-environment initialization exit in any type of language processor environment (integrated and not integrated into TSO/E).

IRXITTS or IRXITMV

There are two post-environment initialization exit routines:

- IRXITTS for environments that are integrated into TSO/E (the TSOFL flag is on)
- IRXITMV for environments that are not integrated into TSO/E (the TSOFL flag is off)

The IRXITTS exit is invoked whenever IRXINIT is called to initialize a new environment and the environment is to be integrated into TSO/E. The IRXITMV exit is invoked whenever IRXINIT is called to initialize a new environment and the environment is not to be integrated into TSO/E. The exits receive control after IRXINIT has initialized the language processor environment and has created the control blocks for the environment, such as the environment block and the parameter block. The exits do not receive any parameters.

IRXTERMX

This is the environment termination exit routine. The exit is invoked whenever the termination routine IRXTERM is called to terminate a language processor environment. The exit receives control before IRXTERM terminates the environment. The exit does not receive any parameters.

You can provide an environment termination exit in any type of language processor environment (integrated and not integrated into TSO/E).

Exec initialization and termination exits

You can provide exits for exec initialization and termination. The exec initialization exit is invoked after the variable pool for a REXX exec has been initialized, but before the language processor processes the first instruction in the exec. The exec termination exit is invoked after a REXX exec has completed, but before the variable pool for the exec has been terminated.

The exec initialization and termination exits do not have fixed names. You name the exits yourself and define the names in the following fields in the module name table:

- EXECINIT - for the exec initialization exit
- EXECTERM - for the exec termination exit

The two exits are used on a language processor environment basis. You can provide an exec initialization and exec termination exit in any type of environment (integrated and not integrated into TSO/E). You define the exit names in the module name table by:

- Providing your own parameters module that replaces the default module, or
- Calling IRXINIT to initialize a language processor environment and passing the module name table on the call.

"Changing the default values for initializing an environment" on page 344 describes how to provide your own parameters module. Chapter 15, "Initialization and termination routines," on page 371 describes the IRXINIT routine.

Exec processing (IRXEXEC) exit routine

You can provide an exec processing exit that is invoked whenever the IRXEXEC routine is called to invoke a REXX exec. The IRXEXEC routine can be explicitly called by a user or called by the system to invoke an exec. IRXEXEC is always called by the system to handle exec processing. For example, if you run a REXX exec in TSO/E using the EXEC command, the IRXEXEC routine is called to invoke the exec. If you provide an exit routine for IRXEXEC, the exit is invoked.

The exit for the IRXEXEC routine does not have a fixed name. You name the exit yourself and define the name in the IRXEXECX field in the module name table.

The exit is used on a language processor environment basis. You can provide an exec processing exit in any type of environment (integrated and not integrated into TSO/E). You define the exit name in the module name table by:

- Providing your own parameters module that replaces the default module, or
- Calling IRXINIT to initialize a language processor environment and passing the module name table on the call.

"Changing the default values for initializing an environment" on page 344 describes how to provide your own parameters module. Chapter 15, "Initialization and termination routines," on page 371 describes the IRXINIT routine.

The exit is invoked before the IRXEXEC routine loads the exec, if the exec is not preloaded, and before IRXEXEC evaluates any parameters passed on the call.

Attention handling exit routine

You can provide an attention handling exit routine that is invoked whenever an exec is running in the TSO/E address space (in a language processor environment that is integrated into TSO/E) and an attention interruption occurs. The exit does not have a fixed name. You name the exit yourself and define the name in the ATTNROUT field in the module name table.

The exit is used on a language processor environment basis. You can provide an attention handling exit in the TSO/E address space only, in an environment that is integrated into TSO/E (the TSOFL flag is on). You define the exit name in the module name table by:

- Providing your own parameters module that replaces the default IRXTSPRM or IRXISPRM module, or
- Calling IRXINIT to initialize a language processor environment and passing the module name table on the call.

"Changing the default values for initializing an environment" on page 344 describes how to provide your own parameters module. Chapter 15, "Initialization and termination routines," on page 371 describes the IRXINIT routine.

The exit is invoked when a REXX exec is running and the user presses the attention interrupt key (usually the PA1 key). The exit gets control before REXX attention processing issues the prompting message, IRX0920I, that asks the user to enter a null line to continue exec processing or one of the immediate commands. The exit is useful if your installation users are unfamiliar with TSO/E READY mode.

You can write an exit to:

- Halt the interpretation of the exec using either the EXECUTIL HI command or the IRXIC routine
- Request that REXX attention processing not display the attention prompting message
- Prohibit the use of the HE immediate command during REXX attention processing.

For information about how the attention handling exit can communicate with REXX attention processing, see *z/OS TSO/E Customization*.

If you provide an attention handling exit routine, the exit should not invoke any authorized commands or programs. Additionally, any unauthorized commands or programs that the exit invokes should be invoked from an unauthorized TSO service facility environment. Otherwise, unpredictable results may occur.

To invoke an unauthorized command or program from an unauthorized TSO service facility environment, you can request the TSO service facility to set up an unauthorized TSO service facility environment for the command or program invocations. For information about using the TSO service facility, see *z/OS TSO/E Programming Services*.

Appendix A. Double-byte character set (DBCS) support

A Double-Byte Character Set supports languages that have more characters than can be represented by 8 bits (such as Korean Hangeul and Japanese kanji). REXX has a full range of DBCS functions and handling techniques.

These include:

- String handling capabilities with DBCS characters
- OPTIONS modes that handle DBCS characters in literal strings, symbols (for example, variable names and labels), comments, and data operations
- A number of functions that specifically support the processing of DBCS character strings
- Defined DBCS enhancements to current instructions and functions

Note: The use of DBCS does not affect the meaning of the built-in functions as described in Chapter 4, "Functions," on page 73. This explains how the characters in a result are obtained from the characters of the arguments by such actions as selecting, concatenating, and padding. The appendix describes how the resulting characters are represented as bytes. This internal representation is not usually seen if the results are printed. It may be seen if the results are displayed on certain terminals.

General description

The following characteristics help define the rules used by DBCS to represent extended characters:

- Each DBCS character consists of 2 bytes.
- Each SBCS character consists of 1 byte.
- There are no DBCS control characters.
- The codes are within the ranges defined in the table, which shows the valid DBCS code for the DBCS blank. You cannot have a DBCS blank in a simple symbol, in the stem of a compound variable, or in a label.

  Table 92. DBCS ranges

Byte	EBCDIC
1st	X'41' to X'FE'
2nd	X'41' to X'FE'
DBCS blank	X'4040'

- DBCS alphanumeric and special symbols

  A DBCS contains double-byte representation of alphanumeric and special symbols corresponding to those of the Single-Byte Character Set (SBCS). In EBCDIC, the first byte of a double-byte alphanumeric or special symbol is X'42' and the second is the same hex code as the corresponding EBCDIC code.

  Here are some examples:

  `X'42C1'` is an EBCDIC double-byte A
  `X'4281'` is an EBCDIC double-byte a
  `X'427D'` is an EBCDIC double-byte quote

- No case translation

  In general, there is no concept of lowercase and uppercase in DBCS.

- Notational conventions

This appendix uses the following notational conventions:

```
DBCS character              ->    .A .B .C .D
SBCS character              ->    a b c d e
DBCS blank                  ->    '. '
EBCDIC shift-out (X'0E')    ->    <
EBCDIC shift-in  (X'0F')    ->    >
```

Note: In EBCDIC, the shift-out (SO) and shift-in (SI) characters distinguish DBCS characters from SBCS characters.

Enabling DBCS data operations and symbol use

The OPTIONS instruction controls how REXX regards DBCS data. To enable DBCS operations, use the EXMODE option. To enable DBCS symbols, use the ETMODE option on the OPTIONS instruction; this must be the first instruction in the program. (See "OPTIONS" on page 56 for more information.)

If OPTIONS ETMODE is in effect, the language processor does validation to ensure that SO and SI are paired in comments. Otherwise, the contents of the comment are not checked. The comment delimiters (/* and */) must be SBCS characters.

Symbols and strings

In DBCS, there are DBCS-only symbols and strings and mixed symbols and strings.

DBCS-Only Symbols and Mixed SBCS/DBCS Symbols

A DBCS-only symbol consists of only non-blank DBCS codes as indicated in Table 92 on page 429.

A mixed DBCS symbol is formed by a concatenation of SBCS symbols, DBCS-only symbols, and other mixed DBCS symbols. In EBCDIC, the SO and SI bracket the DBCS symbols and distinguish them from the SBCS symbols.

The default value of a DBCS symbol is the symbol itself, with SBCS characters translated to uppercase.

A *constant symbol* must begin with an SBCS digit (0–9) or an SBCS period. The delimiter (period) in a compound symbol must be an SBCS character.

DBCS-Only Strings and Mixed SBCS/DBCS Strings

A DBCS-only string consists of only DBCS characters. A mixed SBCS/DBCS string is formed by a combination of SBCS and DBCS characters. In EBCDIC, the SO and SI bracket the DBCS data and distinguish it from the SBCS data. Because the SO and SI are needed only in the mixed strings, they are not associated with the DBCS-only strings.

In EBCDIC:

```
DBCS-only string      ->      .A.B.C
Mixed string          ->      ab<.A.B>
Mixed string          ->      <.A.B>
Mixed string          ->      ab<.C.D>ef
```

Validation

The user must follow certain rules and conditions when using DBCS.

DBCS Symbol Validation

DBCS symbols are valid only if you comply with the following rules:

- The DBCS portion of the symbol must be an even number of bytes in length

- DBCS alphanumeric and special symbols are regarded as different to their corresponding SBCS characters. Only the SBCS characters are recognized by REXX in numbers, instruction keywords, or operators
- DBCS characters cannot be used as special characters in REXX
- SO and SI cannot be contiguous
- Nesting of SO or SI is not permitted
- SO and SI must be paired
- No part of a symbol consisting of DBCS characters may contain a DBCS blank.
- Each part of a symbol consisting of DBCS characters must be bracketed with SO and SI.

Note: When you use DBCS symbols as variable names or labels, the maximum length of a DBCS variable name is the same as the maximum length of an SBCS variable name, 250 bytes, including any SO, SI, DBCS, and SBCS characters. Each DBCS character is counted as 2 bytes and each SO or SI is counted as 1 byte.

These examples show some possible misuses:

```
<.A.BC>        ->  Incorrect because of odd byte length
<.A.B><.C>     ->  Incorrect contiguous SO/SI
<>             ->  Incorrect contiguous SO/SI (null DBCS symbol)
<.A<.B>.C>     ->  Incorrectly nested SO/SI
<.A.B.C        ->  Incorrect because SO/SI not paired
<.A. .B>       ->  Incorrect because contains blank
'. A<.B><.C>   ->  Incorrect symbol
```

Mixed String Validation

The validation of mixed strings depends on the instruction, operator, or function. If you use a mixed string with an instruction, operator, or function that does not allow mixed strings, this causes a **syntax error**.

The following rules must be followed for mixed string validation:

- DBCS strings must be an even number of bytes in length, unless you have SO and SI.

EBCDIC only:

- SO and SI must be paired in a string.
- Nesting of SO or SI is not permitted.

These examples show some possible misuses:

```
'ab<cd'       ->    INCORRECT - not paired
'<.A<.B>.C>   ->    INCORRECT - nested
'<.A.BC>'     ->    INCORRECT - odd byte length
```

The end of a comment delimiter is not found within DBCS character sequences. For example, when the program contains `/* < */`, then the `*/` is not recognized as ending the comment because the scanning is looking for the > (SI) to go with the < (SO) and not looking for `*/`.

When a variable is created, modified, or referred to in a REXX program under OPTIONS EXMODE, it is validated whether it contains a correct mixed string or not. When a referred variable contains a mixed string that is not valid, it depends on the instruction, function, or operator whether it causes a syntax error.

The ARG, PARSE, PULL, PUSH, QUEUE, SAY, TRACE, and UPPER instructions all require valid mixed strings with OPTIONS EXMODE in effect.

Using DBCS symbols as variable names or labels

To enable the use of DBCS characters in variable names and labels, use the ETMODE option of the OPTIONS instruction. For more information, see “OPTIONS” on page 56.

The following are some ways that DBCS names can be used:

- as variables or labels within your program
- as constant symbols
- as a STEM name on EXECIO or as a trapping variable for the OUTTRAP function
- to pass parameters on the LINKPGM, ATTCHPGM, LINKMVS, LU62, CPICOMM, and APPCMVS host command environments
- with functions like SYMBOL and DATATYPE
- in arguments of functions (like LENGTH)
- with the variable access routines IKJCT441 and IRXEXCOM.

The following example shows a program using a DBCS variable name and a DBCS subroutine label:

```
/* REXX */
OPTIONS 'ETMODE'            /* ETMODE to enable DBCS variable names */
MI<.X.E.D>=1                /* Set mixed DBCS variable name         */
<.S.Y.M.D> = 10             /* Variable with DBCS characters between
                               Shift-out (<) and Shift-in (>)       */
call <.D.B.C.S.R.T.N>       /* Call subroutine, with DBCS name      */
 ⋮
<.D.B.C.S.R.T.N>:           /* Subroutine with DBCS name            */
do i = 1 to 10
  if x.i = <.S.Y.M.D> then  /* Does x.i match the DBCS variable's
                               value?                               */
    say 'Value of the DBCS variable is: '<.S.Y.M.D>
 end
 exit 0
```

Instruction examples

Here are some examples that illustrate how instructions work with DBCS.

PARSE

In EBCDIC:

```
x1 = '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 w1
           w1   ->   '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 1 w1
           w1   ->   '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 w1 .
           w1   ->   '<.A.B>'
```

The leading and trailing SO and SI are unnecessary for word parsing and, thus, they are stripped off. However, one pair is still needed for a valid mixed DBCS string to be returned.

```
PARSE VAR x1 . w2
           w2   ->   '<. ><.E><.F><>'
```

Here the first blank delimited the word and the SO is added to the string to ensure the DBCS blank and the valid mixed string.

```
PARSE VAR x1 w1 w2
           w1   ->   '<.A.B>'
           w2   ->   '<. ><.E><.F><>'

PARSE VAR x1 w1 w2 .
           w1   ->   '<.A.B>'
           w2   ->   '<.E><.F>'
```

The word delimiting allows for unnecessary SO and SI to be dropped.

```
x2 = 'abc<>def <.A.B><><.C.D>'

PARSE VAR x2 w1 '' w2
```

```
            w1   ->   'abc<>def <.A.B><><.C.D>'
            w2   ->   ''

PARSE VAR x2 w1 '<>' w2
            w1   ->   'abc<>def <.A.B><><.C.D>'
            w2   ->   ''

PARSE VAR x2 w1 '<><>' w2
            w1   ->   'abc<>def <.A.B><><.C.D>'
            w2   ->   ''
```

Note that for the last three examples '', <>, and <><> are each a null string (a string of length 0). When parsing, the null string matches the end of string. For this reason, w1 is assigned the value of the entire string and w2 is assigned the null string.

PUSH and QUEUE

The PUSH and QUEUE instructions add entries to the data stack. Because an element on the data stack can be up to 1 byte less than 16 megabytes, truncation will probably never occur. However, if truncation splits a DBCS string, REXX ensures that the integrity of the SO-SI pairing is kept under OPTIONS EXMODE.

SAY and TRACE

The SAY and TRACE instructions write information to either the user's terminal or the output stream (the default is SYSTSPRT). Similar to the PUSH and QUEUE instructions, REXX ensures the SO-SI pairs are kept for any data that is separated to meet the requirements of the terminal line size or the OUTDD file.

When the data is split up in shorter lengths, again the DBCS data integrity is kept under OPTIONS EXMODE. In EBCDIC, if the terminal line size is less than 4, the string is treated as SBCS data, because 4 is the minimum for mixed string data.

UPPER

Under OPTIONS EXMODE, the UPPER instruction translates only SBCS characters in contents of one or more variables to uppercase, but it never translates DBCS characters. If the content of a variable is not valid mixed string data, no uppercasing occurs.

DBCS function handling

Some built-in functions can handle DBCS. The functions that deal with word delimiting and length determining conform with the following rules under OPTIONS EXMODE:

1. **Counting characters—**Logical character lengths are used when counting the length of a string (that is, 1 byte for one SBCS logical character, 2 bytes for one DBCS logical character). In EBCDIC, SO and SI are considered to be transparent, and are not counted, for every string operation.
2. **Character extraction from a string—**Characters are extracted from a string on a logical character basis. In EBCDIC, leading SO and trailing SI are not considered as part of one DBCS character. For instance, .A and .B are extracted from <.A.B>, and SO and SI are added to each DBCS character when they are finally preserved as completed DBCS characters. When multiple characters are consecutively extracted from a string, SO and SI that are between characters are also extracted. For example, .A><.B is extracted from <.A><.B>, and when the string is finally used as a completed string, the SO prefixes it and the SI suffixes it to give <.A><.B>.

   Here are some EBCDIC examples:

   ```
   S1 = 'abc<>def'

   SUBSTR(S1,3,1)      ->    'c'
   SUBSTR(S1,4,1)      ->    'd'
   SUBSTR(S1,3,2)      ->    'c<>d'

   S2 = '<><.A.B><>'

   SUBSTR(S2,1,1)      ->    '<.A>'
   SUBSTR(S2,2,1)      ->    '<.B>'
   ```

```
SUBSTR(S2,1,2)      ->    '<.A.B>'
SUBSTR(S2,1,3,'x')  ->    '<.A.B><>x'

S3 = 'abc<><.A.B>'

SUBSTR(S3,3,1)      ->    'c'
SUBSTR(S3,4,1)      ->    '<.A>'
SUBSTR(S3,3,2)      ->    'c<><.A>'
DELSTR(S3,3,1)      ->    'ab<><.A.B>'
DELSTR(S3,4,1)      ->    'abc<><.B>'
DELSTR(S3,3,2)      ->    'ab<.B>'
```

3. **Character concatenation**—String concatenation can only be done with valid mixed strings. In EBCDIC, adjacent SI and SO (or SO and SI) that are a result of string concatenation are removed. Even during implicit concatenation as in the DELSTR function, unnecessary SO and SI are removed.
4. **Character comparison**—Valid mixed strings are used when comparing strings on a character basis. A DBCS character is always considered greater than an SBCS one if they are compared. In all but the strict comparisons, SBCS blanks, DBCS blanks, and leading and trailing contiguous SO and SI (or SI and SO) in EBCDIC are removed. SBCS blanks may be added if the lengths are not identical.

   In EBCDIC, contiguous SO and SI (or SI and SO) between nonblank characters are also removed for comparison.

   Note: The strict comparison operators do not cause syntax errors even if you specify mixed strings that are not valid.

   In EBCDIC:

```
      '<.A>' = '<.A. >'     ->    1     /* true  */
  '<><><.A>' = '<.A><><>'   ->    1     /* true  */
  '<>  <.A>' = '<.A>'       ->    1     /* true  */
'<.A><><.B>' = '<.A.B>'     ->    1     /* true  */
       'abc' < 'ab<. >'     ->    0     /* false */
```

5. **Word extraction from a string**—“Word” means that characters in a string are delimited by an SBCS or a DBCS blank.

   In EBCDIC, leading and trailing contiguous SO and SI (or SI and SO) are also removed when *word*s are separated in a string, but contiguous SO and SI (or SI and SO) in a word are not removed or separated for word operations. Leading and trailing contiguous SO and SI (or SI and SO) of a word are not removed if they are among words that are extracted at the same time.

   In EBCDIC:

```
W1 = '<><. .A. . .B><.C. .D><>'

SUBWORD(W1,1,1)      ->    '<.A>'
SUBWORD(W1,1,2)      ->    '<.A. . .B><.C>'
SUBWORD(W1,3,1)      ->    '<.D>'
SUBWORD(W1,3)        ->    '<.D>'

W2 = '<.A. .B><.C><> <.D>'

SUBWORD(W2,2,1)      ->    '<.B><.C>'
SUBWORD(W2,2,2)      ->    '<.B><.C><> <.D>'
```

Built-in Function Examples

Examples for built-in functions, those that support DBCS and follow the rules defined, are given in this section. For full function descriptions and the syntax diagrams, refer to Chapter 4, “Functions,” on page 73.

ABBREV

In EBCDIC:

```
ABBREV('<.A.B.C>','<.A.B>')     ->    1
ABBREV('<.A.B.C>','<.A.C>')     ->    0
ABBREV('<.A><.B.C>','<.A.B>')   ->    1
ABBREV('aa<>bbccdd','aabbcc')   ->    1
```

Applying the character comparison and character extraction from a string rules.

COMPARE

In EBCDIC:

```
COMPARE('<.A.B.C>','<.A.B><.C>')     ->   0
COMPARE('<.A.B.C>','<.A.B.D>')       ->   3
COMPARE('ab<>cde','abcdx')           ->   5
COMPARE('<.A><>','<.A>','<. >')      ->   0
```

Applying the character concatenation for padding, character extraction from a string, and character comparison rules.

COPIES

In EBCDIC:

```
COPIES('<.A.B>',2)       ->   '<.A.B.A.B>'
COPIES('<.A><.B>',2)     ->   '<.A><.B.A><.B>'
COPIES('<.A.B><>',2)     ->   '<.A.B><.A.B><>'
```

Applying the character concatenation rule.

DATATYPE

```
DATATYPE('<.A.B>')       ->   'CHAR'
DATATYPE('<.A.B>','D')   ->   1
DATATYPE('<.A.B>','C')   ->   1
DATATYPE('a<.A.B>b','D') ->   0
DATATYPE('a<.A.B>b','C') ->   1
DATATYPE('abcde','C')    ->   0
DATATYPE('<.A.B','C')    ->   0
```

Restriction: If *string* is not a valid mixed string and C or D is specified as *type*, 0 is returned.

FIND

```
FIND('<.A. .B.C> abc','<.B.C> abc')    ->   2
FIND('<.A. .B><.C> abc','<.B.C> abc')  ->   2
FIND('<.A. . .B> abc','<.A> <.B>')     ->   1
```

Applying the word extraction from a string and character comparison rules.

INDEX, POS, and LASTPOS

```
INDEX('<.A><.B><><.C.D.E>','<.D.E>')    ->   4
POS('<.A>','<.A><.B><><.A.D.E>')        ->   1
LASTPOS('<.A>','<.A><.B><><.A.D.E>')    ->   3
```

Applying the character extraction from a string and character comparison rules.

INSERT and OVERLAY

In EBCDIC:

```
INSERT('a','b<><.A.B>',1)             -> 'ba<><.A.B>'
INSERT('<.A.B>','<.C.D><>',2)         -> '<.C.D.A.B><>'
INSERT('<.A.B>','<.C.D><><.E>',2)     -> '<.C.D.A.B><><.E>'
INSERT('<.A.B>','<.C.D><>',3,,'<.E>') -> '<.C.D><.E.A.B>'

OVERLAY('<.A.B>','<.C.D><>',2)          -> '<.C.A.B>'
OVERLAY('<.A.B>','<.C.D><><.E>',2)      -> '<.C.A.B>'
OVERLAY('<.A.B>','<.C.D><><.E>',3)      -> '<.C.D><><.A.B>'
OVERLAY('<.A.B>','<.C.D><>',4,,'<.E>')  -> '<.C.D><.E.A.B>'
OVERLAY('<.A>','<.C.D><.E>',2)          -> '<.C.A><.E>'
```

Applying the character extraction from a string and character comparison rules.

JUSTIFY

```
JUSTIFY('<><. .A. . .B><.C. .D>',10,'p')
                   ->   '<.A>ppp<.B><.C>ppp<.D>'
JUSTIFY('<><. .A. . .B><.C. .D>',11,'p')
                   ->   '<.A>pppp<.B><.C>ppp<.D>'
JUSTIFY('<><. .A. . .B><.C. .D>',10,'<.P>')
                   ->   '<.A.P.P.P.B><.C.P.P.P.D>'
JUSTIFY('<><.X. .A. . .B><.C. .D>',11,'<.P>')
                   ->   '<.X.P.P.A.P.P.B><.C.P.P.D>'
```

Applying the character concatenation for padding and character extraction from a string rules.

LENGTH

In EBCDIC:

```
LENGTH('<.A.B><.C.D><>')      ->   4
```

Applying the counting characters rule.

SPACE

In EBCDIC:

```
SPACE('a<.A.B. .C.D>',1)        ->  'a<.A.B> <.C.D>'
SPACE('a<.A><><. .C.D>',1,'x')  ->  'a<.A>x<.C.D>'
SPACE('a<.A><. .C.D>',1,'<.E>') ->  'a<.A.E.C.D>'
```

Applying the word extraction from a string and character concatenation rules.

STRIP

In EBCDIC:

```
STRIP('<><.A><.B><.A><>',,'<.A>')   ->  '<.B>'
```

Applying the character extraction from a string and character concatenation rules.

SUBSTR and DELSTR

In EBCDIC:

```
SUBSTR('<><.A><><.B><.C.D>',1,2)  ->  '<.A><><.B>'
DELSTR('<><.A><><.B><.C.D>',1,2)  ->  '<><.C.D>'
SUBSTR('<.A><><.B><.C.D>',2,2)    ->  '<.B><.C>'
DELSTR('<.A><><.B><.C.D>',2,2)    ->  '<.A><><.D>'
SUBSTR('<.A.B><>',1,2)            ->  '<.A.B>'
SUBSTR('<.A.B><>',1)              ->  '<.A.B><>'
```

Applying the character extraction from a string and character concatenation rules.

SUBWORD and DELWORD

In EBCDIC:

```
SUBWORD('<><. .A. . .B><.C. .D>',1,2) ->  '<.A. . .B><.C>'
DELWORD('<><. .A. . .B><.C. .D>',1,2) ->  '<><. .D>'
SUBWORD('<><.A. . .B><.C. .D>',1,2)   ->  '<.A. . .B><.C>'
DELWORD('<><.A. . .B><.C. .D>',1,2)   ->  '<><.D>'
```

```
SUBWORD('<.A. .B><.C><> <.D>',1,2)    ->  '<.A. .B><.C>'
DELWORD('<.A. .B><.C><> <.D>',1,2)    ->  '<.D>'
```

Applying the word extraction from a string and character concatenation rules.

TRANSLATE

In EBCDIC:

```
TRANSLATE('abcd','<.A.B.C>','abc')       ->  '<.A.B.C>d'
TRANSLATE('abcd','<><.A.B.C>','abc')     ->  '<.A.B.C>d'
TRANSLATE('abcd','<><.A.B.C>','ab<>c')   ->  '<.A.B.C>d'
TRANSLATE('a<>bcd','<><.A.B.C>','ab<>c') ->  '<.A.B.C>d'
TRANSLATE('a<>xcd','<><.A.B.C>','ab<>c') ->  '<.A>x<.C>d'
```

Applying the character extraction from a string, character comparison, and character concatenation rules.

VERIFY

In EBCDIC:

```
VERIFY('<><><.A.B><><.X>','<.B.A.C.D.E>')   ->  3
```

Applying the character extraction from a string and character comparison rules.

WORD, WORDINDEX, and WORDLENGTH

In EBCDIC:

```
W = '<><. .A. . .B><.C. .D>'

WORD(W,1)           ->    '<.A>'
WORDINDEX(W,1)      ->    2
WORDLENGTH(W,1)     ->    1

Y = '<><.A. . .B><.C. .D>'

WORD(Y,1)           ->    '<.A>'
WORDINDEX(Y,1)      ->    1
WORDLENGTH(Y,1)     ->    1

Z = '<.A  .B><.C> <.D>'

WORD(Z,2)           ->    '<.B><.C>'
WORDINDEX(Z,2)      ->    3
WORDLENGTH(Z,2)     ->    2
```

Applying the word extraction from a string and (for WORDINDEX and WORDLENGTH) counting characters rules.

WORDS

In EBCDIC:

```
W = '<><. .A. . .B><.C. .D>'

WORDS(W)           ->    3
```

Applying the word extraction from a string rule.

WORDPOS

In EBCDIC:

```
WORDPOS('<.B.C> abc','<.A. .B.C> abc')            ->   2
WORDPOS('<.A.B>','<.A.B. .A.B><. .B.C. .A.B>',3)  ->   4
```

Applying the word extraction from a string and character comparison rules.

DBCS processing functions

This section describes the functions that support DBCS mixed strings. These functions handle mixed strings regardless of the OPTIONS mode.

Restriction: When used with DBCS functions, *length* is always measured in bytes (as opposed to `LENGTH(`*`string`*`)`, which is measured in characters).

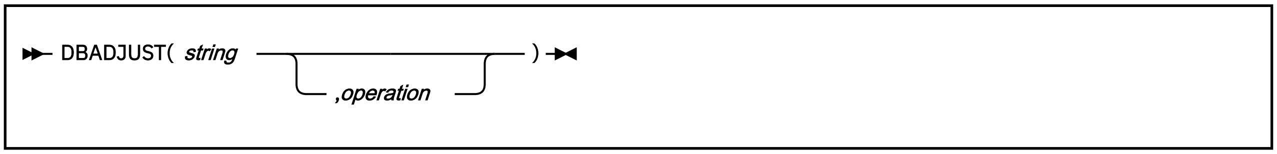
Counting Option

In EBCDIC, when specified in the functions, the counting option can control whether the SO and SI are considered present when determining the length. **Y** specifies counting SO and SI within mixed strings. **N** specifies *not* to count the SO and SI, and is the default.

Function descriptions

The following are the DBCS functions and their descriptions.

DBADJUST



In EBCDIC, adjusts all contiguous SI and SO (or SO and SI) characters in *string* based on the *operation* specified. The following are valid *operation*s. Only the capitalized and highlighted letter is needed; all characters following it are ignored.

Blank
changes contiguous characters to blanks (X'4040').

Remove
removes contiguous characters, and is the default.

Here are some EBCDIC examples:

```
DBADJUST('<.A><.B>a<>b','B')    ->    '<.A.  .B>a  b'
DBADJUST('<.A><.B>a<>b','R')    ->    '<.A.B>ab'
DBADJUST('<><.A.B>','B')        ->    '<.  .A.B>'
```

DBBRACKET

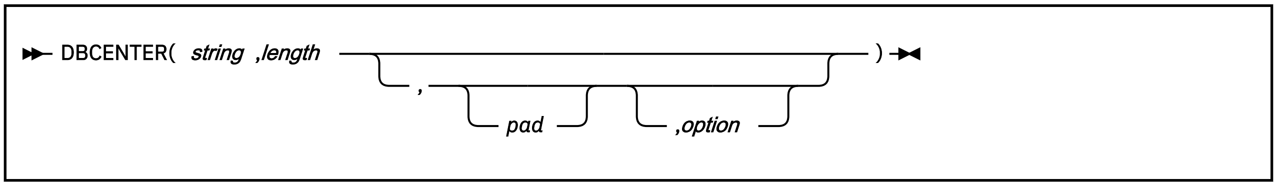
DBBRACKET(
string
)

In EBCDIC, adds SO and SI brackets to a DBCS-only string. If *string* is not a DBCS-only string, a SYNTAX error results. That is, the input string must be an even number of bytes in length and each byte must be a valid DBCS value.

Here are some EBCDIC examples:

```
DBBRACKET('.A.B')      ->    '<.A.B>'
DBBRACKET('abc')       ->    SYNTAX error
DBBRACKET('<.A.B>')    ->    SYNTAX error
```

DBCENTER



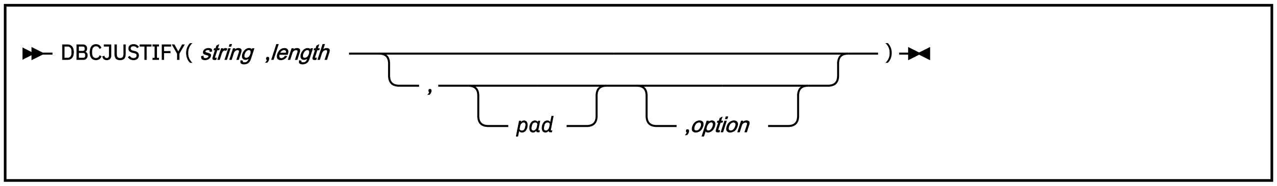
returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up length. The default *pad* character is a blank. If *string* is longer than *length*, it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand end loses or gains one more character than the left-hand end.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBCENTER('<.A.B.C>',4)             ->    ' <.B> '
DBCENTER('<.A.B.C>',3)             ->    ' <.B>'
DBCENTER('<.A.B.C>',10,'x')        ->    'xx<.A.B.C>xx'
DBCENTER('<.A.B.C>',10,'x','Y')    ->    'x<.A.B.C>x'
DBCENTER('<.A.B.C>',4,'x','Y')     ->    '<.B>'
DBCENTER('<.A.B.C>',5,'x','Y')     ->    'x<.B>'
DBCENTER('<.A.B.C>',8,'<.P>')      ->    ' <.A.B.C> '
DBCENTER('<.A.B.C>',9,'<.P>')      ->    ' <.A.B.C.P>'
DBCENTER('<.A.B.C>',10,'<.P>')     ->    '<.P.A.B.C.P>'
DBCENTER('<.A.B.C>',12,'<.P>','Y') ->    '<.P.A.B.C.P>'
```

DBCJUSTIFY



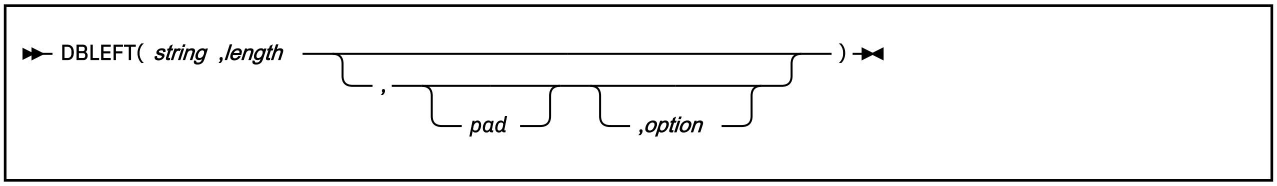
formats *string* by adding *pad* characters between nonblank characters to justify to both margins and length of bytes *length* (*length* must be nonnegative). Rules for adjustments are the same as for the JUSTIFY function. The default *pad* character is a blank.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some examples:

```
DBCJUSTIFY('<><AA  BB><CC>',20,,'Y')
                                  ->   '<AA>    <BB>    <CC>'

DBCJUSTIFY('<><  AA    BB><  CC>',20,'<XX>','Y')
                                  ->   '<AAXXXXXXBBXXXXXXCC>'

DBCJUSTIFY('<><  AA    BB><  CC>',21,'<XX>','Y')
                                  ->   '<AAXXXXXXBBXXXXXXCC> '

DBCJUSTIFY('<><  AA    BB><  CC>',11,'<XX>','Y')
                                  ->   '<AAXXXXBB> '

DBCJUSTIFY('<><  AA    BB><  CC>',11,'<XX>','N')
                                  ->   '<AAXXBBXXCC> '
```

DBLEFT



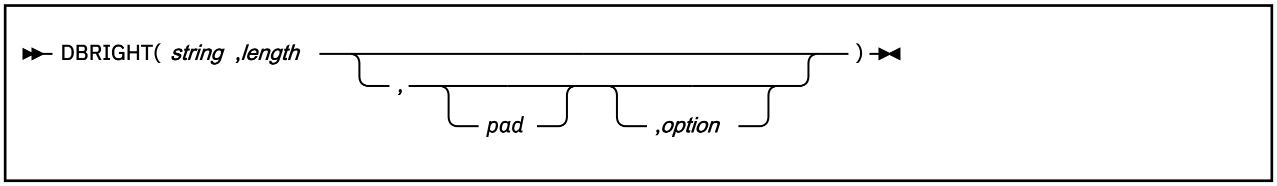
returns a string of length *length* containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBLEFT('ab<.A.B>',4)             ->    'ab<.A>'
DBLEFT('ab<.A.B>',3)             ->    'ab '
DBLEFT('ab<.A.B>',4,'x','Y')     ->    'abxx'
DBLEFT('ab<.A.B>',3,'x','Y')     ->    'abx'
DBLEFT('ab<.A.B>',8,'<.P>')      ->    'ab<.A.B.P>'
DBLEFT('ab<.A.B>',9,'<.P>')      ->    'ab<.A.B.P> '
DBLEFT('ab<.A.B>',8,'<.P>','Y')  ->    'ab<.A.B>'
DBLEFT('ab<.A.B>',9,'<.P>','Y')  ->    'ab<.A.B> '
```

DBRIGHT



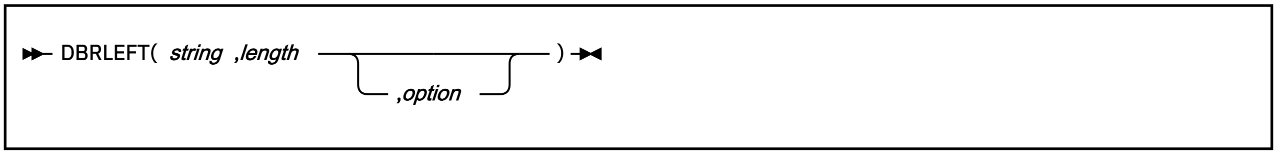
returns a string of length *length* containing the rightmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBRIGHT('ab<.A.B>',4)             ->    '<.A.B>'
DBRIGHT('ab<.A.B>',3)             ->    ' <.B>'
DBRIGHT('ab<.A.B>',5,'x','Y')     ->    'x<.B>'
DBRIGHT('ab<.A.B>',10,'x','Y')    ->    'xxab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>')      ->    '<.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',9,'<.P>')      ->    ' <.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>','Y')  ->    'ab<.A.B>'
DBRIGHT('ab<.A.B>',11,'<.P>','Y') ->    '   ab<.A.B>'
DBRIGHT('ab<.A.B>',12,'<.P>','Y') ->    '<.P>ab<.A.B>'
```

DBRLEFT



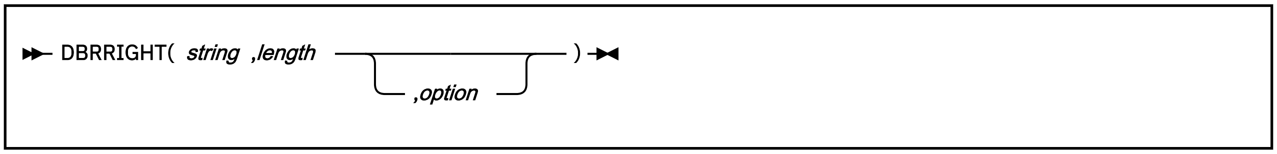
returns the remainder from the DBLEFT function of *string*. If *length* is greater than the length of *string*, returns a null string.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBRLEFT('ab<.A.B>',4)              ->    '<.B>'
DBRLEFT('ab<.A.B>',3)              ->    '<.A.B>'
DBRLEFT('ab<.A.B>',4,'Y')          ->    '<.A.B>'
DBRLEFT('ab<.A.B>',3,'Y')          ->    '<.A.B>'
DBRLEFT('ab<.A.B>',8)              ->    ''
DBRLEFT('ab<.A.B>',9,'Y')          ->    ''
```

DBRRIGHT



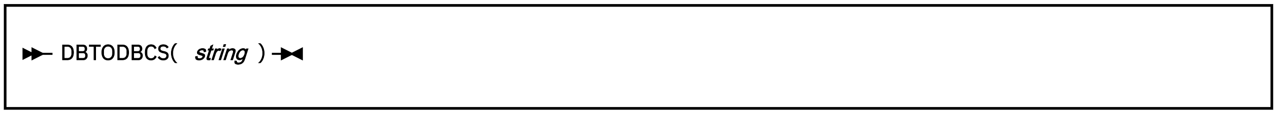
returns the remainder from the DBRIGHT function of *string*. If *length* is greater than the length of *string*, returns a null string.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBRRIGHT('ab<.A.B>',4)             ->    'ab'
DBRRIGHT('ab<.A.B>',3)             ->    'ab<.A>'
DBRRIGHT('ab<.A.B>',5)             ->    'a'
DBRRIGHT('ab<.A.B>',4,'Y')         ->    'ab<.A>'
DBRRIGHT('ab<.A.B>',5,'Y')         ->    'ab<.A>'
DBRRIGHT('ab<.A.B>',8)             ->    ''
DBRRIGHT('ab<.A.B>',8,'Y')         ->    ''
```

DBTODBCS



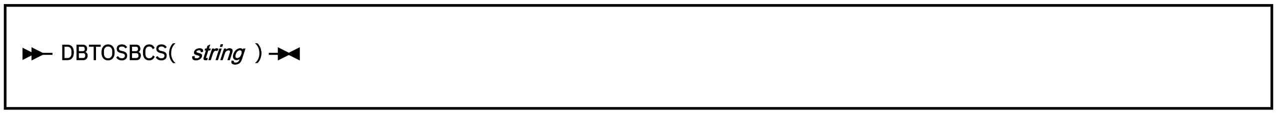
converts all passed, valid SBCS characters (including the SBCS blank) within *string* to the corresponding DBCS equivalents. Other single-byte codes and all DBCS characters are not changed. In EBCDIC, SO and SI brackets are added and removed where appropriate.

Here are some EBCDIC examples:

```
DBTODBCS('Rexx 1988')      ->    '<.R.e.x.x. .1.9.8.8>'
DBTODBCS('<.A> <.B>')      ->    '<.A. .B>'
```

In these examples, the `.x` is the DBCS character corresponding to an SBCS x.

DBTOSBCS



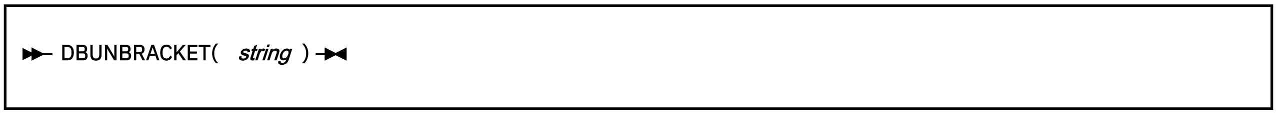
converts all passed, valid DBCS characters (including the DBCS blank) within *string* to the corresponding SBCS equivalents. Other DBCS characters and all SBCS characters are not changed. In EBCDIC, SO and SI brackets are removed where appropriate.

Here are some EBCDIC examples:

```
DBTOSBCS('<.S.d>/<.2.-.1>')     ->    'Sd/2-1'
DBTOSBCS('<.X. .Y>')            ->    '<.X> <.Y>'
```

In these examples, the `.d` is the DBCS character corresponding to an SBCS d. But the `.X` and `.Y` do not have corresponding SBCS characters and are not converted.

DBUNBRACKET

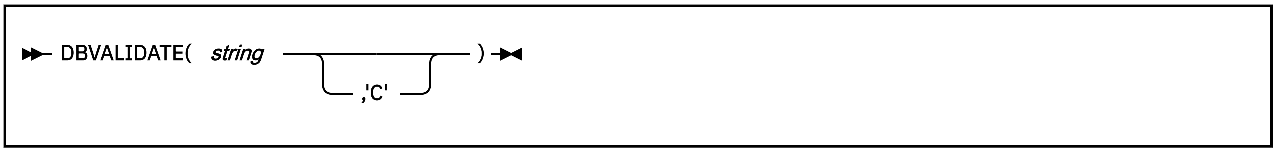


In EBCDIC, removes the SO and SI brackets from a DBCS-only *string* enclosed by SO and SI brackets. If the *string* is not bracketed, a SYNTAX error results.

Here are some EBCDIC examples:

```
DBUNBRACKET('<.A.B>')      ->    '.A.B'
DBUNBRACKET('ab<.A>')      ->    SYNTAX error
```

DBVALIDATE



returns `1` if the *string* is a valid mixed string or SBCS string. Otherwise, returns `0`. Mixed string validation rules are:

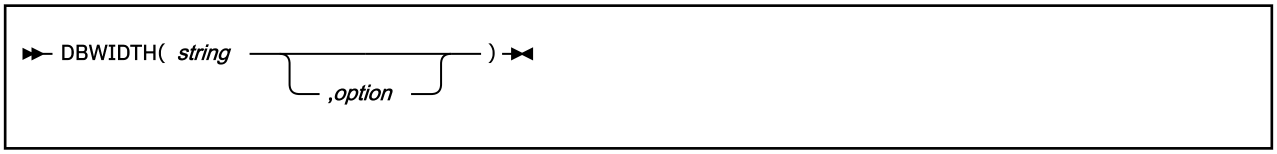
1. Only valid DBCS character codes
2. DBCS string is an even number of bytes in length
3. EBCDIC only — Proper SO and SI pairing

In EBCDIC, if **C** is omitted, only the leftmost byte of each DBCS character is checked to see that it falls in the valid range for the implementation it is being run on (that is, in EBCDIC, the leftmost byte range is from X'41' to X'FE').

Here are some EBCDIC examples:

```
z='abc<de'

DBVALIDATE('ab<.A.B>')          ->    1
DBVALIDATE(z)                   ->    0

y='C1C20E111213140F'X

DBVALIDATE(y)                   ->    1
DBVALIDATE(y,'C')               ->    0
```

DBWIDTH



returns the length of *string* in bytes.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBWIDTH('ab<.A.B>','Y')    ->    8
DBWIDTH('ab<.A.B>','N')    ->    6
```

Appendix B. IRXTERMA routine

The IRXTERMA routine terminates a language processor environment. IRXTERMA differs from the IRXTERM termination routine. IRXTERM terminates a language processor environment only if no active REXX execs are currently running in the environment. IRXTERMA terminates all active REXX execs under a language processor environment, and optionally terminates the environment. If you customize REXX processing and initialize a language processor environment using the IRXINIT initialization routine, when you terminate the environment, use the IRXTERM termination routine. For IRXTERM, see "Termination routine - IRXTERM" on page 386.

Tip: To permit FORTRAN programs to call IRXTERMA, TSO/E provides an alternate entry point for the IRXTERMA routine. The alternate entry point name is IRXTMA.

On the call to IRXTERMA, you specify whether IRXTERMA should terminate the environment in addition to terminating all active execs that are currently running in the environment. You can optionally pass the address of the environment block that represents the environment in which you want IRXTERMA to run. You can pass the address either in parameter 2 or in register 0. If you do not pass an environment block address, IRXTERMA locates the current non-reentrant environment that was created at the same task level and runs in that environment.

IRXTERMA does not terminate an environment if:

- the environment was not initialized under the current task
- the environment was the first environment initialized under the task and other environments are still initialized under the task

However, IRXTERMA does terminate all active execs running in the environment.

IRXTERMA invokes the exec load routine to free each exec in the environment. The exec load routine is the routine identified by the EXROUT field in the module name table, which is one of the parameters for the initialization routine, IRXINIT. All execs in the environment are freed regardless of whether they were pre-loaded before the IRXEXEC routine was called. IRXTERMA also frees the storage for each exec in the environment.

IRXTERMA sets the ENVBLOCK_TERMA_CLEANUP flag to indicate that IRXTERMA is cleaning up the environment. IRXTERMA frees all active execs and optionally terminates the environment itself. This ENVBLOCK_TERMA_CLEANUP flag may be used by the replaceable routines to allow special processing during abnormal termination. If IRXTERMA does not terminate the environment, the flag is cleared upon exit from IRXTERMA.

Entry specifications

For the IRXTERMA termination routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list passed by the caller

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters

In register 1, you pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. The high-order bit of the last address in the parameter list must be set to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter lists for TSO/E REXX routines" on page 242.

Table 93 on page 446 shows the parameters for IRXTERMA.

Table 93. Parameters for IRXTERMA

Parameter	Number of bytes	Description
Parameter 1	4	A fullword field in which you specify whether you want to terminate the environment in addition to terminating all active execs running in the environment. Specify one of the following: • 0 — terminates all execs and the environment • X'80000000' — terminates all execs, but does not terminate the environment.
Parameter 2	4	The address of the environment block that represents the environment you want IRXTERMA to terminate. This parameter is optional. If you specify an environment block address, IRXTERMA uses the value you specify and ignores register 0. However, IRXTERMA does not check whether the address is valid. Therefore, you must ensure the address you specify is correct or unpredictable results can occur. If you do not want to use this parameter, you cannot simply specify an address of 0. If you specify 0, IRXTERMA tries to use 0 as a valid address and fails with a return code of 28. To not use this parameter, end the parameter list at parameter 1 by setting the high-order bit on in the address that points to parameter 1. You can also use register 0 to specify the address of an environment block. If you use register 0, IRXTERMA checks whether the address is valid. If the address is valid, IRXTERMA terminates that environment. Otherwise, IRXTERMA locates the current non-reentrant environment that was created at the same task level and terminates that environment.

Return specifications

For the IRXTERMA termination routine, the contents of the registers on return are:

Register 0

If you passed the address of an environment block in register 0, IRXTERMA returns the address of the environment block for the previous environment. If you did not pass an address in register 0, the register contains the same value as on entry.

If IRXTERMA returns with return code 100 or 104, register 0 contains the abend and reason code. "Return codes" on page 447 describes the return codes and how IRXTERMA returns the abend and reason codes for return codes 100 and 104.

Registers 1-14

Same as on entry

Register 15
Return code

Return codes

Table 94 on page 447 shows the return codes for the IRXTERMA routine.

Table 94. Return codes for IRXTERMA

Return code	Description
0	Processing was successful. If IRXTERMA also terminated the environment, the environment was not the last environment on the task.
4	Processing was successful. If IRXTERMA also terminated the environment, the environment was the last environment on the task.
20	Processing was not successful. IRXTERMA could not terminate the environment.
28	Processing was not successful. The environment could not be found.
100	Processing was not successful. A system abend occurred while IRXTERMA was terminating the environment. IRXTERMA tries to terminate the environment again. If termination is still unsuccessful, the environment cannot be used. The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXTERMA returns the abend code in the two low-order bytes of register 0. IRXTERMA returns the abend reason code in the high-order two bytes of register 0. If the abend reason code is greater than two bytes, IRXTERMA returns only the two low-order bytes of the abend reason code. See *z/OS MVS System Codes* for information about the abend codes and reason codes.
104	Processing was not successful. A user abend occurred while IRXTERMA was terminating the environment. IRXTERMA tries to terminate the environment again. If termination is still unsuccessful, the environment cannot be used. The system may issue one or more messages that describe the abend. In addition, register 0 contains the abend code and the abend reason code. IRXTERMA returns the abend code in the two low-order bytes of register 0. IRXTERMA returns the abend reason code in the high-order two bytes of register 0. If the abend reason code is greater than two bytes, IRXTERMA returns only the two low-order bytes of the abend reason code. See *z/OS MVS System Codes* for information about the abend codes and reason codes.

Appendix C. Writing REXX Execs to perform MVS operator activities

From TSO/E, you can establish an extended MCS console session using the TSO/E CONSOLE command. After you activate a console session, you can issue MVS system and subsystem commands and obtain command responses. This appendix describes the different commands and functions you can use in REXX execs to set up and use a console session.

Activating a console session and issuing MVS commands

TSO/E provides the CONSOLE command that lets you perform MVS operator activities from your TSO/E session. You use the CONSOLE command to activate an extended MCS console session. After you activate a console session, you can then issue MVS system and subsystem commands and obtain command responses. The MVS system and subsystem commands you can use during a console session depend on the MVS command authority defined for the user console. For more information, see *z/OS MVS Planning: Operations*.

To activate a console session, use the TSO/E CONSOLE command with the ACTIVATE keyword, for example:

```
CONSOLE ACTIVATE
```

After you activate a console session, you can use the CONSOLE command with the SYSCMD keyword to issue MVS system and subsystem commands from a REXX exec. For example:

```
"CONSOLE SYSCMD(system_command)"
```

You need not activate the console session from within the REXX exec. You could use the CONSOLE command from TSO/E READY mode to activate a console session and then invoke an exec that issues MVS system and subsystem commands.

To deactivate a console session, use the CONSOLE command with the DEACTIVATE keyword, for example:

```
CONSOLE DEACTIVATE
```

To use the TSO/E CONSOLE command, you must have CONSOLE command authority. For more information, see *z/OS TSO/E System Programming Command Reference*.

Using the CONSOLE Host command environment

TSO/E provides the CONSOLE host command environment that lets you issue MVS system and subsystem commands from a REXX exec. Using the CONSOLE environment eliminates the need for you to repeatedly use the TSO/E CONSOLE command with the SYSCMD keyword to issue MVS commands. With ADDRESS CONSOLE, you need only enter the name of the command.

You can use ADDRESS CONSOLE to issue a single MVS system or subsystem command, for example:

```
ADDRESS CONSOLE "system_command"
```

You can also use ADDRESS CONSOLE and then issue several MVS system or subsystem commands from the CONSOLE host command environment, for example:

```
/*  REXX program ...    */
⋮
"CONSOLE ACTIVATE"
⋮
ADDRESS CONSOLE
```

```
"mvs_cmd1"
⋮
"mvs_cmd2"
⋮
"mvs_cmd3"
⋮
EXIT
```

If you have established CONSOLE as the host command environment and you want to enter TSO/E commands, use the ADDRESS TSO instruction to change the host command environment to TSO. The following example shows how to use the ADDRESS instruction to change between the TSO and CONSOLE host command environments.

```
/*  REXX program  ... */
⋮
"tso_cmd"            /* initial environment is TSO                       */
"CONSOLE ACTIVATE"
⋮
ADDRESS CONSOLE      /* change environment to CONSOLE for all commands   */
"mvs_cmd"
⋮
"mvs_cmd"
ADDRESS TSO tso_cmd      /* change environment to TSO for one command     */
⋮
"mvs_cmd"
⋮
ADDRESS TSO              /* change environment to TSO for all commands    */
"tso_cmd"
⋮
ADDRESS CONSOLE mvs_cmd /* change environment to CONSOLE for one command */
⋮
"tso_cmd"
⋮
"CONSOLE DEACTIVATE"
⋮
EXIT
```

For more information about using the ADDRESS keyword instruction, see "ADDRESS" on page 41.

To use the CONSOLE host command environment, you must have CONSOLE command authority. You must also activate a console session before using ADDRESS CONSOLE. If you use ADDRESS CONSOLE and issue an MVS command before you activate a console session, the CONSOLE environment will not be able to locate the command you issued. In this case, the REXX special variable RC is set to -3 and the FAILURE condition occurs. The -3 return code indicates that the host command environment could not locate the command. In this case, the environment could not locate the command because a console session is not active.

The MVS system and subsystem commands you can use during a console session depend on the MVS command authority defined for the user console. For more information, see *z/OS MVS Planning: Operations*.

Processing messages during a console session

You can use the TSO/E CONSPROF command to control the processing of messages during a console session. Like the CONSOLE command, you must have CONSOLE command authority to use the CONSPROF command.

Usually, you issue the CONSPROF command to tailor a console profile before activating a console session. However, you can also use CONSPROF during a console session to change the profile settings.

There are two types of messages that are routed to the user's console:

- Solicited messages, which are messages that are responses to MVS system and subsystem commands that were issued during the console session.
- Unsolicited messages, which are any messages that are not direct responses to MVS system or subsystem commands. For example, an unsolicited message can be a message that another user sends you or a broadcast message.

You can use the CONSPROF command with the SOLDISPLAY and UNSOLDISPLAY keywords to specify whether solicited messages and unsolicited messages should be displayed at the terminal or saved for later retrieval. See *z/OS TSO/E System Programming Command Reference* for more information about the CONSPROF command.

If messages are not displayed at the terminal during a console session, you can use the TSO/E external function GETMSG to retrieve messages. Using GETMSG, you can retrieve either solicited or unsolicited messages.

The TSO/E external function SYSVAR has the SOLDISP and UNSDISP arguments that relate to the SOLDISPLAY and UNSOLDISPLAY keywords on the CONSPROF command. You can use these SYSVAR arguments to determine whether solicited and unsolicited messages are being displayed. For more information, see "SYSVAR" on page 148.

If messages are not displayed at the terminal, the system stores the messages in message tables. The system stores solicited messages in the solicited message table and unsolicited messages in the unsolicited message table. You can use the SOLNUM and UNSNUM arguments of the TSO/E external function SYSVAR (see "#unique_236/unique_236_Connect_42_tsosvar" on page 148) to determine the current size of the message tables. You can also use the CONSPROF command to change the current size of each table. The size you specify cannot exceed the maximum size set by your installation in SYS1.PARMLIB (member IKJTSOxx). If you do not specify the table size, TSO/E uses the default that your installation defines in SYS1.PARMLIB (member IKJTSOxx).

If you write execs that retrieve messages using GETMSG rather than displaying the messages at the terminal, note the following.

- If a message table exceeds 100% capacity, any new messages are not routed to the user's console until you resolve the message capacity situation.
- TSO/E provides two exits for the CONSOLE command that your installation can use to handle the capacities of the message tables. An exit is invoked when a message table reaches 80% capacity. Another exit is invoked when a table reaches 100% capacity. If your installation provides CONSOLE exits, an exit may be invoked during processing of the exec if the message tables reach 80% or 100% capacity. Exit processing depends on the exits that your installation provides. *z/OS TSO/E Customization* describes the exits for the CONSOLE command and how to set up the sizes for the message tables.

If you retrieve messages using the GETMSG function and then want to display the message to the user, you can use the SYSVAR external function to obtain information related to displaying the message. The MFTIME, MFOSNM, MFJOB, and MFSNMJBX arguments of the SYSVAR function indicate whether the user requested that certain types of information should be displayed with the message, such as the time stamp or the originating job name. For more information about the arguments, see "SYSVAR" on page 148. To obtain information, such as the time stamp or originating job name, you can use the additional MDB variables that the GETMSG function sets. For more information, see Appendix D, "Additional variables that GETMSG sets," on page 455.

Using the CART to associate commands and their responses

The *command and response token* (CART) is a keyword on the TSO/E CONSOLE command and an argument on the GETMSG external function. You can use the CART to associate MVS system and subsystem commands your exec issues with the corresponding responses from the commands that are routed to the user's console. To use a CART to associate commands and their responses, solicited messages that are routed to the user's console should not be displayed at the terminal. You must store solicited messages and then retrieve the messages using the GETMSG function.

When you issue an MVS system or subsystem command with a CART, the CART is associated with any messages (responses) that the command issues. When you use GETMSG to retrieve responses from the MVS command, use the same CART on the GETMSG function.

If you issue MVS commands during a console session and have never specified a CART, the default CART value is '0000000000000000'X. When you specify a CART, the CART remains in effect for all subsequent MVS commands you issue until you specify a different CART.

You can use a CART in different ways depending on how you issue MVS system and subsystem commands in the exec. If you use the CONSOLE command with the SYSCMD keyword to issue an MVS command, you can use the CART keyword on the CONSOLE command to specify a CART. For example:

```
"CONSOLE SYSCMD(system_command) CART(AP090032)"
```

In the example, the CART value AP090032 is used for all subsequent MVS commands until you use another CART.

If you use the CONSOLE host command environment, you can specify the CART in several ways. If you use ADDRESS CONSOLE to issue a single MVS system or subsystem command and you want to use a CART, first use ADDRESS CONSOLE and specify the word CART followed by the CART value. You must separate the word CART and the CART value with a blank. Then use ADDRESS CONSOLE and specify the command. For example:

```
ADDRESS CONSOLE "CART AP120349"
ADDRESS CONSOLE "system_command"
```

Again, the CART is used for all subsequent MVS system and subsystem commands until you use another CART value.

You can also use ADDRESS CONSOLE to change the host command environment for all subsequent commands. If you want to use a CART for specific commands, enter the word CART followed by a blank, followed by the CART value. The CART remains in effect until you use another CART value.

For example, suppose you use ADDRESS CONSOLE to issue a series of MVS commands. For the first command, you want to use a CART of APP50000. For the second command, you want to use a CART of APP50001. For the third, fourth, and fifth commands, you want to use a CART of APP522. For the remaining commands, you want to use a CART of APP5100. You could specify the CART values as follows:

```
/*  REXX program ....   */
⋮
ADDRESS CONSOLE
"CART APP50000"
"mvs_cmd1"
⋮
"CART APP50001"
"mvs_cmd2"
⋮
"CART APP522"
"mvs_cmd3"
"mvs_cmd4"
"mvs_cmd5"
⋮
"CART APP5100"
"mvs_cmd6"
⋮
EXIT
```

Considerations for Multiple Applications

If you have two or more programs that issue MVS system and subsystem commands during a console session and the programs will run simultaneously in a user's TSO/E address space, the programs must use CART values to ensure they retrieve messages intended only for their program. If two programs that use the CONSOLE command's services coexist in one TSO/E address space, you should be aware of the following:

- You should issue all MVS system and subsystem commands with a CART.
- Use the first 4 bytes of the CART as an application identifier. Installations should establish standards so that each program uses an identifier that identifies the program. Whenever the program uses a CART, the CART should begin with the four byte identifier.
- You should not display solicited messages at the terminal. Each application should use GETMSG to explicitly retrieve solicited messages intended for that application.
- You cannot selectively retrieve unsolicited messages. You can have unsolicited messages displayed or you can have one application retrieve all unsolicited messages using GETMSG.

- When you use GETMSG to retrieve a solicited message, you can use the `mask` argument with the `cart` argument as follows. Use a MASK of ‘FFFFFFFF00000000’X. The CART should contain the application identifier as the first four bytes. For more information about using a MASK, see “GETMSG” on page 110.

You may also want to use CART values if you have an exec that calls a second exec and both execs issue MVS commands during a console session. You could establish a four byte application identifier for each exec and then use the CART and MASK on the GETMSG function to retrieve solicited messages intended for that exec. You could also simply use unique CART values.

Example of Determining Results From Commands in One Exec

You can use CART values in one exec to determine the results from particular commands. For example, if you issue MVS commands and want to perform different processing based on each command, use a unique CART value for each command invocation. When you use GETMSG to retrieve solicited messages from a specific command, specify the same CART that you used when you invoked the command.

The following illustrates the use of the CART for determining the results of two specific commands. From TSO/E READY mode, activate a console session and then start two system printers (PRT1 and PRT2). Specify a unique CART for each START command. After you start the printers, call the CHKPRT exec and pass the value of the CART as an argument. For example:

```
READY

CONSPROF SOLDISP(NO) SOLNUM(400)
CONSOLE ACTIVATE
CONSOLE SYSCMD($S PRT1) CART('PRT10001')
CONSOLE SYSCMD($S PRT2) CART('PRT20002')
EXEC MY.EXEC(CHKPRT) 'PRT10001' EXEC
EXEC MY.EXEC(CHKPRT) 'PRT20002' EXEC
```

The exec you invoke (CHKPRT) checks whether the printers were started successfully. The exec uses the arguments you pass on the invocation (CART values) as the CART on the GETMSG function. Figure 30 on page 453 shows the example exec.

```
/* REXX exec to check start of printers */
ARG CARTVAL
GETCODE = GETMSG('PRTMSG.','SOL',CARTVAL,,60)
IF GETCODE = 0 THEN
  DO
    IF POS('$HASP000',PRTMSG.1) ¬= 0 THEN
      SAY "Printer started successfully."
    ELSE
      DO INDXNUM = 1 TO PRTMSG.0
        SAY PRTMSG.INDXNUM
      END
  END
ELSE
  SAY "GETMSG error retrieving message.  Return code is" GETCODE
EXIT
```

Figure 30. Example exec (CHKPRT) to check start of printers

For more information about the GETMSG function, see “GETMSG” on page 110. For more information about the TSO/E CONSOLE command, see *z/OS TSO/E System Programming Command Reference*.

Appendix D. Additional variables that GETMSG sets

The TSO/E external function GETMSG retrieves a message that has been issued during a console session and stores the message in variables. On the call to GETMSG, you specify the msgstem argument. GETMSG places each line of the message text it retrieves into successive variables identified by the msgstem you specify. For more information about GETMSG, see "GETMSG" on page 110.

In addition to the variables into which GETMSG places the retrieved message (as specified by the *msgstem* argument), GETMSG sets other variables that contain additional information about the message that was retrieved. One set of variables relates to the entire message itself (that is, to all lines of message text that GETMSG retrieves, regardless of how many lines of text the message has). "Variables GETMSG sets for the entire message" on page 455 describes these variables.

The second set of variables is an array of variables that GETMSG sets for each line of message text that GETMSG retrieves. "Variables GETMSG sets for each line of message text" on page 460 describes these variables.

Variables GETMSG sets for the entire message

GETMSG sets specific variables that relate to the entire message that it retrieves. GETMSG sets these variables, regardless of how many lines of text the retrieved message contains.

The names of the variables that GETMSG sets correspond to the field names in the message data block (MDB) in MVS/ESA System Product. The variable names consist of the msgstem you specified on the call to GETMSG followed by the name of the field in the MDB. That is, TSO/E uses the name of the field in the MDB as the suffix for the variable name and concatenates the MDB field name to the msgstem. For example, one field in the MDB is MDBLEN, which is the length of the MDB. If you specify msgstem as "CONSMSG." (with a period), REXX returns the length of the MDB in the variable:

```
CONSMSG.MDBLEN
```

If you specify msgstem as "CMSG" (without a period), the variable name would be CMSGMDBLEN.

Table 95 on page 455 describes the variables GETMSG sets for a message that it retrieves. For any variable that needs a minimum MVS release to have a proper value returned, this minimum prerequisite release is listed in the second column.

For detailed information about the MDB and each field in the MDB, see *z/OS MVS Data Areas* in the z/OS Internet library (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosInternetLibrary).

Table 95. Variables GETMSG sets for an entire message

Variable suffix name	Prerequisite release	Description
MDBLEN		Length of the MDB, in decimal.
MDBTYPE		MDB type, in decimal.
MDBMID		Four character MDB identifier, which is 'MDB '.
MDBVER		Version of the MDB; 4-byte hexadecimal value.
MDBGLEN		General object length of the MDB, in decimal.
MDBGTYPE		General object type of the MDB, in decimal.
MDBGMID		Four byte message identifier, in hexadecimal.
MDBGSYID		One byte system ID, in hexadecimal. The value is the same as the first byte of the MDBGMID variable (message identifier).

Table 95. Variables GETMSG sets for an entire message (continued)

Variable suffix name	Prerequisite release	Description
MDBGSEQ		Three byte sequence number, in hexadecimal. The value is the same as the last three bytes of the MDBGMID variable (message identifier).
MDBGTIMH		Time stamp in the format: `hh.mm.ss` where hh is hours, mm is minutes, and ss is seconds.
MDBGTIMT		Time stamp in the format: `.th` where th is tenths of seconds, .36, for example.
MDBGDSTP		Date stamp in the format yyyyddd, where yyyy is the year and ddd is the number of days, including the current day, so far in the year.
MDBGDOM		General DOM indicator. Contains the value YES or NO that indicates whether messages that match the message ID are to be deleted.
MDBGALRM		YES or NO to indicate whether the processor alarm is sounded.
MDBGHOLD		Hold indicator: YES or NO to indicate whether the message should be held until DOMed or deleted by other external means.
MDBGFCON		Foreground control presentation attribute, in decimal.
MDBGFCOL		Foreground color presentation attribute, in decimal.
MDBGFHIL		Foreground highlighting presentation attribute, in decimal.
MDBGFINT		Foreground intensity presentation attribute, in decimal.
MDBGBCON		Background control presentation attribute, in decimal.
MDBGBCOL		Background color presentation attribute, in decimal.
MDBGBHIL		Background highlighting presentation attribute, in decimal.
MDBGBINT		Background intensity presentation attribute, in decimal.
MDBGOSNM		Eight character originating system name.
MDBGJBNM		Eight character job name.
MDBCLEN		Control object length of the MDB, in decimal.
MDBCTYPE		Control object type of the MDB, in decimal.
MDBCPROD		Sixteen character originating system identifier.
MDBCVER		MVS CP object version level; 4-byte hexadecimal value.
MDBCPNAM		Four character control program name.
MDBCFMID		Eight character FMID of the originating system.
MDBCERC		Routing codes; 16-byte hexadecimal value.
MDBCDESC		Descriptor codes; 2-byte hexadecimal value.

Table 95. Variables GETMSG sets for an entire message (continued)

Variable suffix name	Prerequisite release	Description
MDBDESCA		YES or NO to indicate whether the message pertains to a system failure.
MDBDESCB		YES or NO to indicate whether the message requires an immediate action.
MDBDESCC		YES or NO to indicate whether the message requires an eventual action.
MDBDESCD		YES or NO to indicate whether the message pertains to system status.
MDBDESCE		YES or NO to indicate whether the message is an immediate command response.
MDBDESCF		YES or NO to indicate whether the message pertains to job status.
MDBDESCG		YES or NO to indicate whether or not the message was issued by an application program or application processor.
MDBDESCH		YES or NO to indicate whether the message is directed to an out-of-line area.
MDBDESCI		YES or NO to indicate whether the message pertains to an operator request.
MDBDESCJ		YES or NO to indicate whether the message is a track command response.
MDBDESCK		YES or NO to indicate whether the message requires a critical eventual action.
MDBDESCL		YES or NO to indicate whether the message is an important informational message.
MDBDESCM	4.3.0	YES or NO to indicate: previously automated.
MDBCMLVL		Message level; 2-byte hexadecimal value.
MDBMLR		YES or NO to indicate whether the message is a WTOR.
MDBMLIA		YES or NO to indicate whether the message requires an immediate action.
MDBMLCE		YES or NO to indicate whether the message requires a critical eventual action.
MDBMLE		YES or NO to indicate whether the message requires an eventual action.
MDBMLI		YES or NO to indicate whether the message is an informational message.
MDBMLBC		YES or NO to indicate whether the message is a broadcast message.
MDBCSUPP	5.1.0	YES or NO to indicate whether the message is suppressed.
MDBCMCSC		YES or NO to indicate whether the message is a command response.

Table 95. Variables GETMSG sets for an entire message (continued)

Variable suffix name	Prerequisite release	Description
MDBCAUTH		YES or NO to indicate whether the message was issued by an authorized program.
MDBCRETN		YES or NO to indicate whether the message is retained by AMRF.
MDBCSPVD	5.2.0	YES or NO to indicate: WQE backlog message.
MDBCASID		ASID of the issuer; 2-byte hexadecimal value.
MDBCTCB		TCB of the job step; 4-byte hexadecimal value.
MDBCTOKN		Token that the issuer of the message used, in decimal.
MDBCSYID		System ID, in decimal.
MDBDMSGI		YES or NO to indicate whether operator messages with the specific message ID (as specified by the MDBGMID variable) should be deleted.
MDBDSYSI		YES or NO to indicate whether operator messages with the specific system ID (as specified by the MDBGSYID variable) should be deleted.
MDBDASID		YES or NO to indicate whether operator messages with the specific ASID (as specified by the MDBCASID variable) should be deleted.
MDBDJTCB		YES or NO to indicate whether operator messages with the specific job step TCB (as specified by the MDBCTCB variable) should be deleted.
MDBDTOKN		YES or NO to indicate whether operator messages with the specific token (as specified by the MDBCTOKN variable) should be deleted.
MDBCAUT	4.3.0	YES or NO to indicate: QUEUE by using automation.
MDBCHC	5.1.0	YES or NO to indicate: QUEUE via hardcopy.
MDBCOCMD	5.2.0	YES or NO to indicate: Echo operator command.
MDBCICMD	5.2.0	YES or NO to indicate: Echo internal command.
MDBCWTL	5.2.0	YES or NO to indicate: result of WTL macro.
MDBCOJID		Eight character originating job ID.
MDBCKEY		Eight character retrieval key.
MDBCAUTO		Eight character automation token.
MDBCCART		Eight character command and response token (CART).
MDBCCNID		Console ID; 4-byte hexadecimal value.
MDBMSGTA		YES or NO to indicate whether the message was issued because job names were being monitored.
MDBMSGTB		YES or NO to indicate whether the message was issued because status was being monitored.
MDBMSGTC		YES or NO to indicate whether monitor is active.
MDBMSGTD		YES or NO to indicate whether the QID field exists in the WPL (AOS/1).

Table 95. Variables GETMSG sets for an entire message (continued)

Variable suffix name	Prerequisite release	Description
MDBMSGTF		YES or NO to indicate whether the message was issued because sessions were being monitored.
MDBCRPYL		Length of the reply ID, in decimal. The reply ID is returned in the variable MDBCRPYI, which is described below.
MDBCRPYI		EBCDIC representation of the reply ID.
MDBCTOFF		The offset in the message text field to the beginning of the message, in decimal.
MDBCRPYB		Reply ID, in decimal.
MDBCAREA		One character area ID.
MDBCLCNT		Number of lines of message text in the message, in decimal.
MDBCOJBN		Eight character originating job name.
MDBCSPLX	5.1.0	8-character SYSPLEX name.
MDBCRCMT	5.2.0	YES or NO to indicate whether the message text was changed.
MDBCRCRC	5.2.0	YES or NO to indicate whether routing code(s) were changed.
MDBCRCDC	5.2.0	YES or NO to indicate whether descriptor code(s) were changed.
MDBCRQPC	5.2.0	YES or NO to indicate: queued to a particular active console.
MDBCRQRC	5.2.0	YES or NO to indicate: queued by routing codes only.
MDBCRPML	5.2.0	YES or NO to indicate whether minor lines were processed.
MDBCRDTM	5.2.0	YES or NO to indicate whether the message was deleted.
MDBCROMS	5.2.0	YES or NO to indicate: MPF suppression was overridden.
MDBCRFHC	5.2.0	YES or NO to indicate: hardcopy forced.
MDBCRNHC	5.2.0	YES or NO to indicate: no hardcopy was forced.
MDBCRHCO	5.2.0	YES or NO to indicate: only hardcopy forced.
MDBCRBCA	5.2.0	YES or NO to indicate: broadcast message to active consoles.
MDBCRBCN	5.2.0	YES or NO to indicate: did not broadcast message to active consoles.
MDBCRNRT	5.2.0	YES or NO to indicate: AMRF is not to retain this message.
MDBCRRET	5.2.0	YES or NO to indicate: AMRF is to retain this message.
MDBCRCKY	5.2.0	YES or NO to indicate: changed the retrieval key.
MDBCRCFC	5.2.0	YES or NO to indicate: changed the 4-byte console ID.
MDBCRCMF	5.2.0	YES or NO to indicate: changed the message type flags.
MDBCRANO	5.2.0	YES or NO to indicate: automation was not required.
MDBCRAYS	5.2.0	YES or NO to indicate: automation was required and/or automation token updated.
MDBCQHCO	5.2.0	YES or NO to indicate: message issued as hardcopy only.

Table 95. Variables GETMSG sets for an entire message (continued)

Variable suffix name	Prerequisite release	Description
MDBCSNSV	5.2.0	YES or NO to indicate: not serviced by any WTO user exit routine.
MDBCSEER	5.2.0	YES or NO to indicate: A WTO user exit abended while processing this message.
MDBCSNSI	5.2.0	YES or NO to indicate: not serviced due to an incompatible request.
MDBCSAUT	5.2.0	YES or NO to indicate: automation specified.
MDBCSSSI	5.2.0	YES or NO to indicate: suppressed by a subsystem.
MDBCSWTO	5.2.0	YES or NO to indicate: suppressed by a WTO user exit routine.
MDBCSMPF	5.2.0	YES or NO to indicate: suppressed by MPF.
MDBCCNNM	5.2.0	8-character console name.
MDBMCSB	5.2.0	YES or NO to indicate: MCSFLAG=REG0 was specified.
MDBMCSI	5.2.0	YES or NO to indicate: MCSFLAG=NOTIME was specified.

Variables GETMSG sets for each line of message text

GETMSG also sets an array of variables for the message it retrieves. The variables are set for each line of message text for the retrieved message.

The variable names are compound symbols. The stem of each variable name is the same for all lines of message text. The value following the period (.) in the variable name is the line number of the line of message text.

The names of the variables correspond to the field names in the message data block (MDB) in MVS/ESA System Product. The variable names consist of the `msgstem` you specified on the call to GETMSG, followed by the name of the field in the MDB, followed by a period (.), which is then followed by the line number of the message text. For example, one field in the message data block is MDBTTYPE, which is the text object type of the MDB. If you specify `msgstem` as "CMSG." (with a period), and GETMSG retrieves a message that has three lines of message text, GETMSG sets the following MDBTTYPE variables:

CMSG.MDBTTYPE.1 (corresponding to the first line of message text)
CMSG.MDBTTYPE.2 (corresponding to the second line of message text)
CMSG.MDBTTYPE.3 (corresponding to the third line of message text)

If you specified the `msgstem` as "CMSG" (without a period), GETMSG sets the three variables as CMSGMDBTTYPE.1, CMSGMDBTTYPE.2, and CMSGMDBTTYPE.3.

Table 96 on page 460 describes the array of variables that GETMSG sets for each line of message text.

Table 96. Variables GETMSG sets for each line of message text

Variable suffix name	Description
MDBTLEN.n	Text object length of the MDB, in decimal.
MDBTTYPE.n	Text object type of the MDB, in decimal.
MDBTCONT.n	YES or NO to indicate whether the line of message text consists of control text.
MDBTLABT.n	YES or NO to indicate whether the line of message text consists of label text.

Table 96. Variables GETMSG sets for each line of message text (continued)

Variable suffix name	Description
MDBTDATT.n	YES or NO to indicate whether the line of message text consists of data text.
MDBTENDT.n	YES or NO to indicate whether the line of message text consists of end text.
MDBTPROT.n	YES or NO to indicate whether the line of message text consists of prompt text.
MDBTFPAF.n	YES or NO to indicate whether the text object presentation attribute field overrides the general object presentation attribute field.
MDBTPCON.n	Presentation control attribute, in decimal.
MDBTPCOL.n	Presentation color attribute, in decimal.
MDBTPHIL.n	Presentation highlighting attribute, in decimal.
MDBTPINT.n	Presentation intensity attribute, in decimal.

Appendix E. REXX symbol and hexadecimal code cross-reference

The tables below show the REXX symbols and their corresponding hexadecimal codes as found in U.S Code Page (037). Table 97 on page 463 shows these symbols in the order of their hexadecimal values. Table 98 on page 466 shows these symbols in a logical sequence.

Table 97. REXX symbol and hexadecimal code - in hexadecimal sequence

Hex code	Symbol	Description
0E		DBCS shift out character (SO)
0F		DBCS shift in character (SI)
40		blank
4A	¢	cent sign
4B	.	period
4C	<	less than
4D	(	left parenthesis
4E	+	plus sign
4F	|	or bar
50	&	ampersand
5A	!	exclamation point
5B	$	dollar sign
5C	*	asterisk
5D	)	right parenthesis
5E	;	semicolon
5F	¬	logical not
60	–	dash
61	/	forward slash
6B	,	comma
6C	%	percent sign
6D	_	underscore
6E	>	greater than
6F	?	question mark
7A	:	colon
7B	#	pound sign
7C	@	at sign

Table 97. REXX symbol and hexadecimal code - in hexadecimal sequence (continued)

Hex code	Symbol	Description
7D	'	single quote
7E	=	equal sign
7F	"	double quote
81	a	lowercase a
82	b	lowercase b
83	c	lowercase c
84	d	lowercase d
85	e	lowercase e
86	f	lowercase f
87	g	lowercase g
88	h	lowercase h
89	i	lowercase i
91	j	lowercase j
92	k	lowercase k
93	l	lowercase l
94	m	lowercase m
95	n	lowercase n
96	o	lowercase o
97	p	lowercase p
98	q	lowercase q
99	r	lowercase r
A2	s	lowercase s
A3	t	lowercase t
A4	u	lowercase u
A5	v	lowercase v
A6	w	lowercase w
A7	x	lowercase x
A8	y	lowercase y
A9	z	lowercase z
C1	A	uppercase A
C2	B	uppercase B
C3	C	uppercase C
C4	D	uppercase D

Table 97. REXX symbol and hexadecimal code - in hexadecimal sequence (continued)

Hex code	Symbol	Description
C5	E	uppercase E
C6	F	uppercase F
C7	G	uppercase G
C8	H	uppercase H
C9	I	uppercase I
D1	J	uppercase J
D2	K	uppercase K
D3	L	uppercase L
D4	M	uppercase M
D5	N	uppercase N
D6	O	uppercase O
D7	P	uppercase P
D8	Q	uppercase Q
D9	R	uppercase R
E0	\	backslash
E2	S	uppercase S
E3	T	uppercase T
E4	U	uppercase U
E5	V	uppercase V
E6	W	uppercase W
E7	X	uppercase X
E8	Y	uppercase Y
E9	Z	uppercase Z
F0	0	numeral 0
F1	1	numeral 1
F2	2	numeral 2
F3	3	numeral 3
F4	4	numeral 4
F5	5	numeral 5
F6	6	numeral 6
F7	7	numeral 7
F8	8	numeral 8
F9	9	numeral 9

Table 98. REXX symbol and hexadecimal code - in logical sequence

Symbol	Hex code	Description
Uppercase Letters		
A	C1	uppercase A
B	C2	uppercase B
C	C3	uppercase C
D	C4	uppercase D
E	C5	uppercase E
F	C6	uppercase F
G	C7	uppercase G
H	C8	uppercase H
I	C9	uppercase I
J	D1	uppercase J
K	D2	uppercase K
L	D3	uppercase L
M	D4	uppercase M
N	D5	uppercase N
O	D6	uppercase O
P	D7	uppercase P
Q	D8	uppercase Q
R	D9	uppercase R
S	E2	uppercase S
T	E3	uppercase T
U	E4	uppercase U
V	E5	uppercase V
W	E6	uppercase W
X	E7	uppercase X
Y	E8	uppercase Y
Z	E9	uppercase Z
Lowercase Letters		
a	81	lowercase a
b	82	lowercase b
c	83	lowercase c
d	84	lowercase d
e	85	lowercase e

Table 98. REXX symbol and hexadecimal code - in logical sequence (continued)

Symbol	Hex code	Description
f	86	lowercase f
g	87	lowercase g
h	88	lowercase h
i	89	lowercase i
j	91	lowercase j
k	92	lowercase k
l	93	lowercase l
m	94	lowercase m
n	95	lowercase n
o	96	lowercase o
p	97	lowercase p
q	98	lowercase q
r	99	lowercase r
s	A2	lowercase s
t	A3	lowercase t
u	A4	lowercase u
v	A5	lowercase v
w	A6	lowercase w
x	A7	lowercase x
y	A8	lowercase y
z	A9	lowercase z
Numbers		
0	F0	numeral 0
1	F1	numeral 1
2	F2	numeral 2
3	F3	numeral 3
4	F4	numeral 4
5	F5	numeral 5
6	F6	numeral 6
7	F7	numeral 7
8	F8	numeral 8
9	F9	numeral 9
Special Characters		

Table 98. REXX symbol and hexadecimal code - in logical sequence (continued)

Symbol	Hex code	Description
¢	4A	cent sign
<	4C	less than
+	4E	plus sign
|	4F	or bar
>	6E	greater than
=	7E	equal sign
.	4B	period
(	4D	left parenthesis
&	50	ampersand
!	5A	exclamation point
$	5B	dollar sign
*	5C	asterisk
)	5D	right parenthesis
;	5E	semicolon
¬	5F	logical not
—	60	dash
/	61	forward slash
\	E0	backslash
,	6B	comma
%	6C	percent sign
_	6D	underscore
?	6F	question mark
:	7A	colon
#	7B	pound sign
@	7C	at sign
'	7D	single quote
"	7F	double quote
Other Codes		
	0E	DBCS shift out character (SO)
	0F	DBCS shift in character (SI)
	40	blank

Appendix F. Accessibility

Accessible publications for this product are offered through IBM Documentation (www.ibm.com/docs/en/zos).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the Contact the z/OS team web page (www.ibm.com/systems/campaignmail/z/zos/contact_z) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- *z/OS TSO/E Primer*
- *z/OS TSO/E User's Guide*
- *z/OS ISPF User's Guide Vol I*

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Documentation with 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 are considered 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 the 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, 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 is placed 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 to provide 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, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element

The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark 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 the 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.

! indicates a default syntax element

The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate 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 dotted decimal number can specify the ! 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 the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the 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 applies only 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.

*** indicates an optional syntax element that is repeatable**

The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero 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, 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

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 can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the 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 repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Notices

This information was developed for products and services that are offered in the USA or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for IBM Documentation. Use of hyperlinks in other output formats of this information is at your own risk.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Site Counsel
2455 South Road

Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user’s name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at ibm.com®/privacy and IBM’s Online Privacy Statement at ibm.com/privacy/details in the section entitled “Cookies, Web Beacons and Other Technologies,” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMSdfp, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming interface information

This document describes intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS TSO/E REXX language processor.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and Trademark information (www.ibm.com/legal/copytrade.shtml).

Index

Product Number: 5650-ZOS

SA32-0972-50