IBM VSE/Enterprise Systems Architecture
VSE Central Functions
6.7

VSE/REXX Reference

SC33-6642-11

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xxiii.

Twelfth Edition (March 2004)

This edition applies to Version 6 Release 7 of IBM REXX/VSE, which is part of VSE/Central Functions, Program Number 5686-066, and to all subsequent releases and modifications until otherwise indicated in new editions.

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

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to:

IBM Deutschland Entwicklung GmbH
Department 3248
Schoenaicher Strasse 220
D-71032 Boeblingen
Federal Republic of Germany

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

Contents

Figures

Tables

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, USA.

For online versions of this book, we authorize you to:

- Copy, modify, and print the documentation contained on the media, for use within your enterprise, provided you reproduce the copyright notice, all warning statements, and other required statements on each copy or partial copy.
- Transfer the original unaltered copy of the documentation when you transfer the related IBM product (which may be either machines you own, or programs, if the program's license terms permit a transfer). You must, at the same time, destroy all other copies of the documentation.

You are responsible for payment of any taxes, including personal property taxes, resulting from this authorization.

There are no warranties, express or implied, including the warranties of merchantability and fitness for a particular purpose.

Some juristictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

Your failure to comply with the terms above terminates this authorization. Upon termination, you must destroy your machine readable documentation.

Programming Interface Information

This book is intended to help the customer write programs in the REXX programming language and customize services that REXX/VSE 6.7 provides for REXX processing. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by REXX/VSE 6.7.

General-use programming interfaces allow the customer to write programs that obtain the services of REXX/VSE 6.7.

However, this book also documents Product-sensitive Programming Interface and Associated Guidance Information provided by REXX/VSE 6.7.

Product-sensitive programming interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of REXX/VSE 6.7. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive programming interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs, by an introductory statement.

The programming interfaces include data areas and parameter lists. Unless otherwise stated, all fields in data areas/parameter lists are part of the programming interface. However, all "Reserved ..." fields are not part of the programming interface.

See Appendix A, “List of the Names of Macros Intended for Customers' Use,” on page 527 for a list of macros intended as programming interfaces.

Trademarks and Service Marks

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

IBM	BookManager
Library Reader	VSE/ESA
Systems Application Architecture	SAA
MVS/ESA	AS/400
PS/2	OS/2

Summary of Changes

The twelfth edition of this manual (March 2004) contains several updates and editorial changes.

Chapter 1. Introduction

This introductory section:

- Identifies the book's purpose and audience
- Explains how to use the book.

Who Should Read This Book

This book describes the REstructured eXtended eXecutor (REXX) language. The REXX language is implemented through:

- The REXX/VSE interpreter
- The Library for REXX/370 in REXX/VSE.

The interpreter is also called the language processor. The Library for REXX/370 in REXX/VSE is also called a compiler's runtime processor. This book is intended for experienced programmers, particularly those who have used a block-structured, high-level language (for example, PL/I, Algol, or Pascal).

REXX/VSE is a partial implementation of Level 2 Systems Application Architecture (SAA) REXX on the VSE/ESA system. The purpose of SAA REXX is to define a consistent set of language elements that can be used on several operating systems. If you plan to run REXX programs on other environments, however, some restrictions may apply and you should review the publication *SAA Common Programming Interface REXX Level 2 Reference*, SC24-5549.

REXX programs can do many tasks, including the automation of VSE/Operations. For example, if you use the JCL EXEC command to call a REXX program, you can leave JCL statements on the stack for VSE/ESA to process. This enables you to insert JCL statements or data into the current job stream.

Descriptions include the use and syntax of the language and how the language processor "interprets" the language while a program is running under the REXX/VSE interpreter. The book also describes:

- REXX/VSE external functions and REXX/VSE commands you can use in a REXX program
- Programming services that let you interface with REXX and the language processor
- 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.

The Compiler and the Library for REXX/370

See Chapter 24, “Support for the Library for REXX/370 in REXX/VSE,” on page 499 for information about the Compiler and the Library for REXX/370.

How to Use This Book

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

1. Chapter 1, “Introduction,” on page 1
2. Chapter 2, “REXX General Concepts,” on page 7
3. Chapter 3, “Keyword Instructions,” on page 27 (in alphabetic order)
4. Chapter 4, “Functions,” on page 59 (in alphabetic order)
5. Chapter 5, “Parsing,” on page 105 (a method of dividing character strings, such as commands)
6. Chapter 6, “Numbers and Arithmetic,” on page 119
7. Chapter 7, “Conditions and Condition Traps,” on page 129

8. Chapter 8, “Using REXX,” on page 135
9. Chapter 9, “Reserved Keywords, Special Variables, and Command Names,” on page 141
10. Chapter 10, “REXX/VSE Commands,” on page 143
11. Chapter 11, “ADDRESS POWER Commands,” on page 181
12. Chapter 12, “JCL Command Environment,” on page 201
13. Chapter 13, “Host Command Environments for Loading and Calling Programs,” on page 205
14. Chapter 14, “REXX/VSE Console Automation,” on page 217
15. Chapter 15, “REXX Sockets Application Program Interface,” on page 275
16. Chapter 16, “Debug Aids,” on page 319
17. Chapter 17, “Programming Services,” on page 323
18. Chapter 18, “Customizing Services,” on page 381
19. Chapter 19, “Language Processor Environments,” on page 387
20. Chapter 20, “Initialization and Termination Routines,” on page 427
21. Chapter 21, “Replaceable Routines and Exits,” on page 439
22. Chapter 22, “Double-Byte Character Set (DBCS) Support,” on page 479
23. Chapter 23, “ARXTERMA Routine,” on page 495
24. Chapter 24, “Support for the Library for REXX/370 in REXX/VSE,” on page 499

Appendixes cover:

- Appendix A, “List of the Names of Macros Intended for Customers' Use,” on page 527
- Appendix B, “Servicing REXX/VSE,” on page 529
- Appendix C, “REXX Supplied Link Books,” on page 531

This introduction and Chapter 2, “REXX General Concepts,” on page 7 provide general information about the REXX programming language. The two chapters provide an introduction to REXX/VSE 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.

Chapter 3, “Keyword Instructions,” on page 27 describes the keyword instructions. Chapter 4, “Functions,” on page 59 describes the SAA built-in functions, additional built-in functions, and external functions that REXX/VSE provides.

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 105
- Chapter 6, “Numbers and Arithmetic,” on page 119
- Chapter 7, “Conditions and Condition Traps,” on page 129
- Chapter 9, “Reserved Keywords, Special Variables, and Command Names,” on page 141
- Chapter 16, “Debug Aids,” on page 319.

REXX/VSE provides several REXX/VSE commands you can use for REXX processing. Chapter 10, “REXX/VSE Commands,” on page 143 describes the syntax of these commands.

Chapter 12, “JCL Command Environment,” on page 201 describes these environments introduced in Chapter 2, “REXX General Concepts,” on page 7 in greater detail.

Chapter 14, “REXX/VSE Console Automation,” on page 217 describes a special REXX/VSE facility that is centered around a VSE/ESA programmable console. This facility enables you to automate and make more productive the operation of your VSE/ESA console.

Besides REXX language support, REXX/VSE provides:

- Programming services you can use to interface with REXX and the language processor

- Customizing services that let you customize REXX processing and how the language processor accesses and uses system services, such as I/O and storage.

Chapter 17, “Programming Services,” on page 323 describes programming services. Chapter 18, “Customizing Services,” on page 381 introduces customizing services, which the following chapters describe in greater detail:

- Chapter 19, “Language Processor Environments,” on page 387
- Chapter 20, “Initialization and Termination Routines,” on page 427
- Chapter 21, “Replaceable Routines and Exits,” on page 439.

Note: REXX/VSE is interactive only from the operator's console. This reservation applies to any terms in this book suggesting interactive input and output. For example, *displaying* output refers to presenting it through the current output stream; *entering* information refers to providing it through the current input stream.

The REXX/VSE messages are included in the z/VSE Messages and Codes manual and therefore available in all VSE/ESA supported languages.

See the z/VSE Planning, and z/VSE System Upgrade and Service, if you plan to use the Fast Service Upgrade (FSU) function to migrate to VSE/ESA 2.1. Appendix B, “Servicing REXX/VSE,” on page 529 provides information to help you reload phases after service into the SVA.

How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ▶▶— symbol indicates the beginning of a statement.

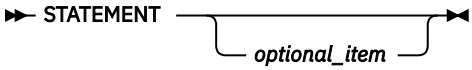
  The —▶ symbol indicates that the statement syntax is continued on the next line.

  The ▶— symbol indicates that a statement is continued from the previous line.

  The —▶◀ symbol indicates the end of a statement.

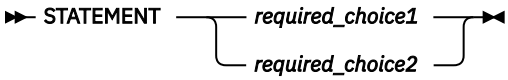
  Diagrams of syntactical units other than complete statements start with the ▶— symbol and end with the —▶ symbol.
- Required items appear on the horizontal line (the main path).

- Optional items appear below the main path.

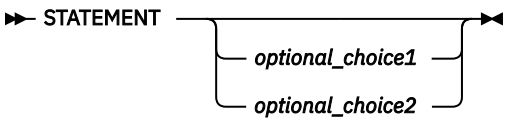


- If you can choose from two or more items, they appear vertically, in a stack.

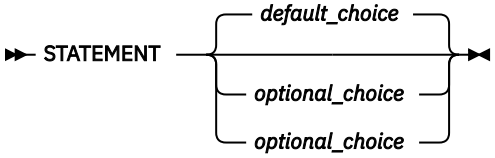
  If you *must* choose one of the items, one item of the stack appears on the main path.



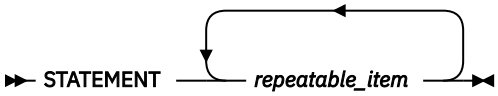
  If choosing one of the items is optional, the entire stack appears below the main path.



- If one of the items is the default, it appears above the main path and the remaining choices are shown below.

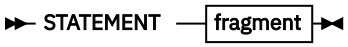


- An arrow returning to the left above the main line indicates an item that can be repeated.



  A repeat arrow above a stack indicates that you can repeat the items in the stack.

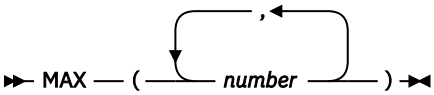
- A set of vertical bars around an item indicates that the item is a *fragment*, a part of the syntax diagram that appears in greater detail below the main diagram.



fragment

- Keywords appear in uppercase (for example, PARM1). They must be spelled exactly as shown but can be specified in any case. Variables appear in all lowercase letters (for example, *parmx*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

The following example shows how the syntax is described:



For Further REXX Information

The following lists publications that are useful for programming in REXX:

- The *SAA Common Programming Interface REXX Level 2 Reference*, SC24-5549, may be useful to more experienced REXX users who may wish to code portable programs. This book defines SAA REXX. Descriptions include the use and syntax of the language as well as explanations on how the language processor interprets the language as a program is running.
- The OS/390 TSO/E REXX Reference, is a comprehensive reference for use on TSO/E.
- The OS/390 TSO/E REXX User's Guide, introduces the instructions and functions the REXX language provides and explains how to write a REXX program. It describes how to run a REXX program in TSO/E foreground and background, in MVS batch using JCL, or in any address space. This book also highlights the major differences between the TSO/E CLIST language and the REXX language.
- The *VM/ESA: REXX/VM Primer*, SC24-5598, is an excellent introduction to REXX and can help you get started. If you have little or no experience in computer programming or programming in REXX, it is worthwhile reading.
- The VM/ESA REXX/VM Reference, is a comprehensive reference for use on VM.
- The REXX/VM User's Guide, is suitable for beginners and programmers who have not used a structured language before.
- The VSE/ESA REXX/VSE User's Guide, provides a general introduction to REXX programming for beginners. It introduces REXX instructions and built-in functions and explains how to write a REXX program. It includes many examples of REXX applications.

- The VSE/ESA REXX/VSE Diagnosis Reference, provides information to help with diagnosing problems, developing search arguments for searching problem reporting data bases, and collecting data for reporting problems to IBM.
- The VSE/ESA Messages and Codes, contains REXX error numbers and messages.

See page "Compiler Publications" on page 500 for a list of books for the IBM Compiler and Library for REXX/370.

Chapter 2. REXX General Concepts

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

- Command procedures
- Application front ends
- 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.

The limit on the length of the value of variables is the amount of storage available in a single request.

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.

A host command is a command for the surrounding system to act upon. Issuing host commands from within a REXX program is an integral part of the REXX language.

You can use REXX/VSE commands (for example, MAKEBUF, DROPBUF, and NEWSTACK) and ADDRESS POWER commands in a REXX program. You can also link to programs and issue JCL commands. “Host Commands and Host Command Environments.” on page 24 describes the different environments for using host services.

The location for all parts of REXX/VSE is the PRD1.BASE sublibrary. All descriptions and examples in this book refer to this sublibrary.

REXX programs must reside in a member of a sublibrary in the active PROC chain. For more information, see REXX/VSE User's Guide.

You can call a program from batch using the JCL EXEC command. See Figure 19 on page 329 for an example. Or you can call the ARXEXEC or ARXJCL interface from any program. See “Calling REXX with ARXEXEC or ARXJCL” on page 331 for more information. Programs are loaded from the active PROC chain.

A language processor runs REXX programs. If a program is interpreted, it is processed line-by-line and word-by-word. It is not first translated to another form (compiled).

When a program is loaded into storage, the load routine checks for sequence numbers in the REXX program. 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 442.

Where to Find More Information

You can find useful information in the REXX/VSE User's Guide. For any program written in the REXX language, you can use the REXX TRACE instruction to get information on how the language processor interprets the program or a particular instruction.

See page “Compiler Publications” on page 500 for a list of books for the IBM Compiler and Library for REXX/370.

Structure and General Syntax

REXX programs are recommended to start with a comment. REXX/VSE does not require this. However, for portability reasons, you are recommended to start each REXX program with a comment that begins on the first line and includes the word REXX. The example in Figure 1 on page 8 illustrates this. The program starts with a comment and the characters "REXX" are in the first line (line 1).

```
/* REXX program to check ...
   ...  The program then ... */
   ...
   ...
   ...
EXIT
```

Figure 1. Example of Using the REXX Program Identifier

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 9)
- Zero or more blanks (again ignored)
- A semicolon (;) delimiter that may 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 "Special Characters: " on page 12) 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 encodings 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 encodings for each character in a set. 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 encodings in different code pages (for example, !, the exclamation point).

For information about Double-Byte Character Set characters, see Chapter 22, "Double-Byte Character Set (DBCS) Support," on page 479.

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 Chapter 22, “Double-Byte Character Set (DBCS) Support,” on page 479 and the OPTIONS instruction “OPTIONS” on page 43.

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 may 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
```

Note: 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 may 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 may 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. See Chapter 22, “Double-Byte Character Set (DBCS) Support,” on page 479 for more information.

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

[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.

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 may 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:

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

Implementation maximum: A symbol can consist of up to 250 characters. (But note that, if it is a variable, the only limit on its value is the amount of storage obtainable in a single request.)

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 Chapter 6, “Numbers and Arithmetic,” on page 119-“Errors” on page 126 for a full definition of numbers.

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

You can specify numbers with or without quotation marks around them. Note that the sequence `-17.9` (without quotation marks) in an expression is not simply a number. It is a minus operator (which may be prefix minus if no term is to the left of it) followed by a positive number. The result of the operation is a number.

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 “Operators” on page 13). 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 may not be available in all character sets, and, if this is the case, appropriate translations may 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 may 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.

Note: 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

One way to continue a clause onto the next line is to use 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.
```

Expressions and Operators

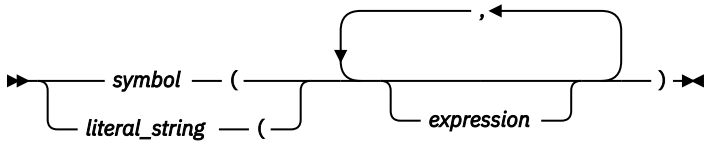
Expressions in REXX are a general mechanism for combining one or more pieces of data in various ways to produce a result, usually different from the original data.

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 59), 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 16). 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) may 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 pages describe how each operator (except for the prefix operators) acts on two terms, which may 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 may 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 9) 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 119 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 Chapter 6, "Numbers and Arithmetic," on page 119) 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

Note: 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).

Note: 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 may 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.

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 19.

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 may contain *any* characters.

You can assign a new value to a variable with the ARG, PARSE, or PULL instructions, the VALUE built-in function, the VALUE built-in function, or the Variable Access Routine (IRXEXCOM), or the variable pool access interface (ARXEXCOM) 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. If you omit `expression`, the variable is set to the null string. However, it is recommended that you explicitly set a variable to the null string: `symbol=''`.

Variable names can contain DBCS characters. For information about DBCS characters, see Chapter 22, "Double-Byte Character Set (DBCS) Support," on page 479.

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 20), 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
```

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
```

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.

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 to the individual variable.

For example:

```
hole.  = "empty"
hole.9 = "full"

say  hole.1  hole.mouse  hole.9

/* says "empty empty full" */
```

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
```

Note: 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 “DROP” on page 37), and `PROCEDURE EXPOSE FRED.` exposes *all possible* variables with that stem (see “PROCEDURE” on page 46).

Note:

1. When the ARG, PARSE, or PULL instruction or the VALUE built-in function or the variable pool access interface (ARXEXCOM), 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 may consist purely of an expression (see “Commands to External Environments” on page 23), 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 page “SYMBOL” on page 82) 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 “NOVALUE ” on page 129).

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. The environment for processing host commands. is known as the host command environment. 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.

REXX/VSE provides six host command environments: VSE, POWER, JCL, LINK, LINKPGM, and CONSOLE. The default environment for processing commands is VSE.

"Host Commands and Host Command Environments." on page 24 explains the different types of host commands you can use in a REXX program and the different host command environments 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 in the host command environment table. "Host Command Environment Table" on page 401 describes the table in more detail. "Changing the Default Values for Initializing an Environment" on page 412 describes how to change the defaults to define your own host command environments. You can also use the ARXSUBCM routine to maintain entries in the host command environment table (see page "Maintain Entries in the Host Command Environment Table – ARXSUBCM" on page 357).

Commands

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

```
expression;
```

The expression is evaluated, resulting in a character string (which may 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 may 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 132.

In addition to setting a return code, the underlying system may 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. (An example of an error could be an EXECIO command that tries to write a record that is truncated.) 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 129). They may 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 53.

Here is an example of submitting a command.

```
"ADDRESS VSE EXEC" myprog
```

The host command environment is VSE. MYPROG is a member in a sublibrary in the active PROC chain. The command results in running MYPROG.

Note: Whenever you enter a host command from a REXX program, enclose in quotation marks any part of the expression that is not to be evaluated. This can be the entire command or parts of the expression.

Whenever a host command is processed, the return code from the command is placed in the REXX special variable RC.

Host Commands and Host Command Environments.

A host command is a command for the surrounding environment to act upon. You can issue host commands from a REXX program. When the language processor processes a clause that it does not recognize as an assignment or other REXX instruction, the language processor treats the clause as a *host command* and routes the command to the 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 REXX/VSE command (such as NEWSTACK or QBUF)
- An ADDRESS POWER command (such as PUTQE, GETQE, QUERYMSG, or any of the POWER commands that you can issue through a CTL service request. See "The POWER Host Command Environment" on page 25 and Chapter 11, "ADDRESS POWER Commands," on page 181 for details.)
- The name of a REXX procedure in the active PROC search chain.
- A JCL command.
- The name of a program invoked by ADDRESS LINK or ADDRESS LINKPGM.
- An ADDRESS CONSOLE command (such as ACTIVATE, CART, CONSTATE, CONSWITCH, and DEACTIVATE).

If a REXX program contains

```
FRED var1 var2
```

the language processor considers the clause to be a *host 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 can be the return code from a VSE/ESA command processor.) A return code of **-3** is always returned if you use a host command in a program and the host command environment cannot locate the command (REXX/VSE command, REXX program, or phase).

If a system abend occurs during a host command, no return code is set and no recovery is available. 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 129 for more information.

If you issue a host command in a REXX program, it is recommended that you enclose the entire command (or as much of it as possible) in quotation marks, for example:

```
"routine-name p1 p2"
```

REXX/VSE provides six host command environments:

- VSE

- POWER
- JCL
- LINK
- LINKPGM
- CONSOLE

The VSE Host Command Environment

The default host command environment is VSE.

You can use the VSE host command environment to invoke REXX/VSE commands (such as MAKEBUF and NEWSTACK) and services. (See Chapter 10, “REXX/VSE Commands,” on page 143).

You can also call another REXX program using the EXEC command. In the VSE environment, you can use all REXX/VSE commands but you cannot use POWER, JCL, or Console commands. You can use one of the following instructions to call a REXX program. The instructions in the following example assume the current host command environment is **not** VSE.

```
ADDRESS VSE "EXEC programname p1 p2 …"
ADDRESS VSE "EX programname p1 p2 …"
ADDRESS VSE "programname p1 p2 …"  /* Implicit EXEC command */
```

If you use the ADDRESS VSE EXEC command to invoke another REXX program, the system searches the active PROC chain for the partition. If the program is not found, the search for the program 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 VSE environment is a signed 31 bit number in the range -2,147,483,648 to +2,147,483,647.

To load and call a phase from the active PHASE search chain, use one of the host command environments that Chapter 13, “Host Command Environments for Loading and Calling Programs,” on page 205 describes.

The POWER Host Command Environment

The POWER host command environment is for VSE/POWER spool-access services requests, GET, PUT, and CTL. (For details about the VSE/POWER spool-access services interface, see VSE/POWER Application Programming. In the POWER host command environment, you can use both REXX/VSE and POWER commands. The POWER host command environment lets you:

- Use the PUTQE command to put elements on a POWER queue and the GETQE command to retrieve POWER queue elements
- Send a CTL service request to POWER. See “CTL” on page 196 for a list of the POWER commands that you can send through a CTL service request. See VSE/POWER Administration and Operation, for the syntax of these commands.
- Use the QUERYMSG command to return job completion messages into the stem specified by OUTTRAP.
- Execute REXX/VSE commands

When the language processor encounters a command for the POWER host command environment, it:

1. Checks if it is GETQE, PUTQE or QUERYMSG. If so, the language processor executes the command.
2. Checks if it is a valid command for the ADDRESS VSE environment. If so, the language processor executes the command.
3. Sends the command to POWER through the VSE/POWER spool-access services interface.

The JCL Host Command Environment

You can use the JCL host command environment to issue a JCL command in a much simpler way than with the conditional job control language. This host command environment is invoked via the command ADDRESS JCL.

See Chapter 12, “JCL Command Environment,” on page 201 for detailed information.

The LINK and LINKPGM Host Command Environments

Loading and calling a program is called **linking**. REXX/VSE provides the LINK and LINKPGM host command environments to let you load and call non-REXX programs. These programs must be phases from the active PHASE search chain. LINK and LINKPGM offer different ways to provide parameters.

See Chapter 13, “Host Command Environments for Loading and Calling Programs,” on page 205 for detailed information.

The CONSOLE Host Command Environment

The CONSOLE host command environment allows to activate and deactivate one or more VSE/ESA console sessions. Having activated a VSE/ESA console session, VSE/ESA console commands can be imbedded into a REXX program. A GETMSG function receives command responses and console messages.

See Chapter 14, “REXX/VSE Console Automation,” on page 217 for detailed information.

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.

As explained in "Keyword Instructions" on page 19, 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 (though this is generally not recommended).

Certain other keywords, known as subkeywords, are reserved within the clauses of individual instructions. 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 141.

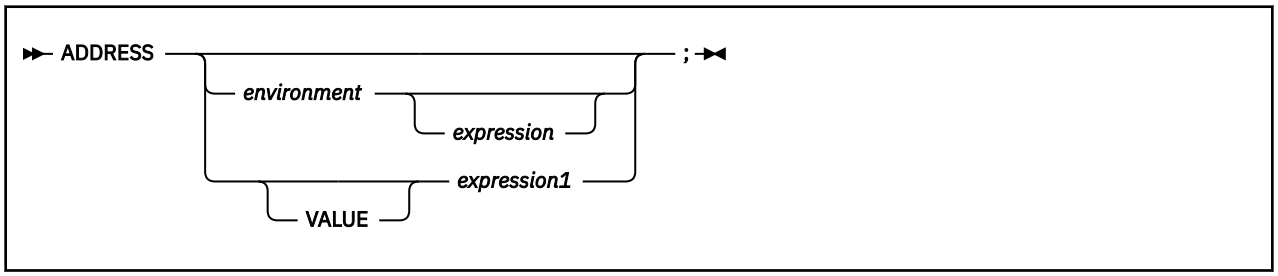
Blanks adjacent to keywords have no effect other than to separate the keyword from the subsequent token. One or more blanks following VALUE are required to separate the *expression* from the subkeyword in the example following:

```
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. A command is a clause that is not a REXX assignment or another REXX instruction. Commands are strings sent to an external environment. You can send commands by specifying clauses consisting of only an expression (see "Commands to External Environments" on page 23) or by using the ADDRESS instruction.

REXX/VSE provides the following host command environments:

- VSE (for REXX/VSE commands). **This is the default**. In this environment, you can use REXX/VSE commands but not POWER commands.
- POWER (for VSE/POWER spool-access services requests—GET, CTL, GCM, and PUT). In this environment, you can use both REXX/VSE and POWER commands.
- JCL. In this environment, you can issue a JCL command in a much simpler way than with the conditional VSE job control language. You can issue JCL commands via a REXX program.
- Environments for linking to a program
  - LINK (See "The LINK Host Command Environment" on page 206 for details.)
  - LINKPGM (See "The LINKPGM Host Command Environment" on page 208).
- CONSOLE. In this environment, you can manage VSE/ESA console sessions.

"Commands to External Environments" on page 23 describes how to enter commands to the host.

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 23.) Errors and failures in commands processed in this way are trapped or traced as usual.

Example:

```
ADDRESS LINK "routine p1 p2"
ADDRESS JCL "MAP"                  /*    VSE environment    */
```

If you specify only *environment*, a lasting change of destination occurs: all commands that follow are routed to the specified command environment, until the next ADDRESS instruction is processed. The previously selected environment is saved.

Example:

```
Address VSE
"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 the CALL instruction ("CALL" on page 30) 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 page "ADDRESS" on page 62).

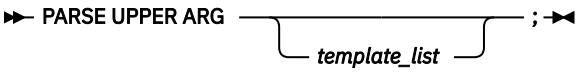
REXX/VSE provides host command environments that you can use with the ADDRESS instruction. After the environment processes the host command, a return code from the command is set in the REXX special variable RC. The return code may be a -3, which indicates that the environment could not

locate the command you specified. For more information about the environments you can use with the ADDRESS instruction and the return codes set in the special variable RC, see "The VSE Host Command Environment" on page 25.

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 401.

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 the section on parsing ("Parsing Rules" on page 105).

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:

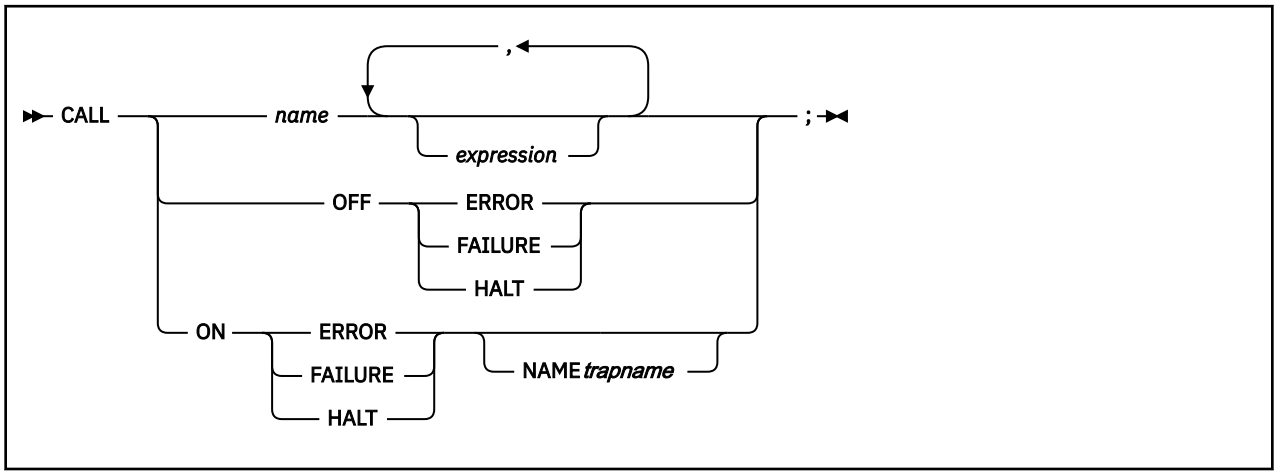
```
/* Function is called by  FRED('data X',1,5)    */

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 63.

2. The source of the data being processed is also made available on entry to the program. See the PARSE instruction (SOURCE option) “PARSE” on page 44 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 on condition traps is contained in Chapter 7, “Conditions and Condition Traps,” on page 129.

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:

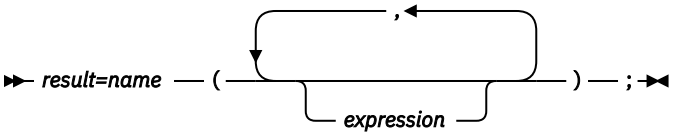
An internal routine
A function or subroutine that is in the same program as the CALL instruction or function call that calls it.

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 24), but RC is not set when you use the CALL instruction. See Chapter 9, “Reserved Keywords, Special Variables, and Command Names,” on page 141 for descriptions of the three REXX special variables RESULT, RC, and SIGL.

REXX/VSE 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 59.) The search order is in the section on functions (see “Search Order” on page 60) 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 51). 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 61.)

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 344. For information about the search order REXX/VSE uses to locate external routines, see “Search Order” on page 60. 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:

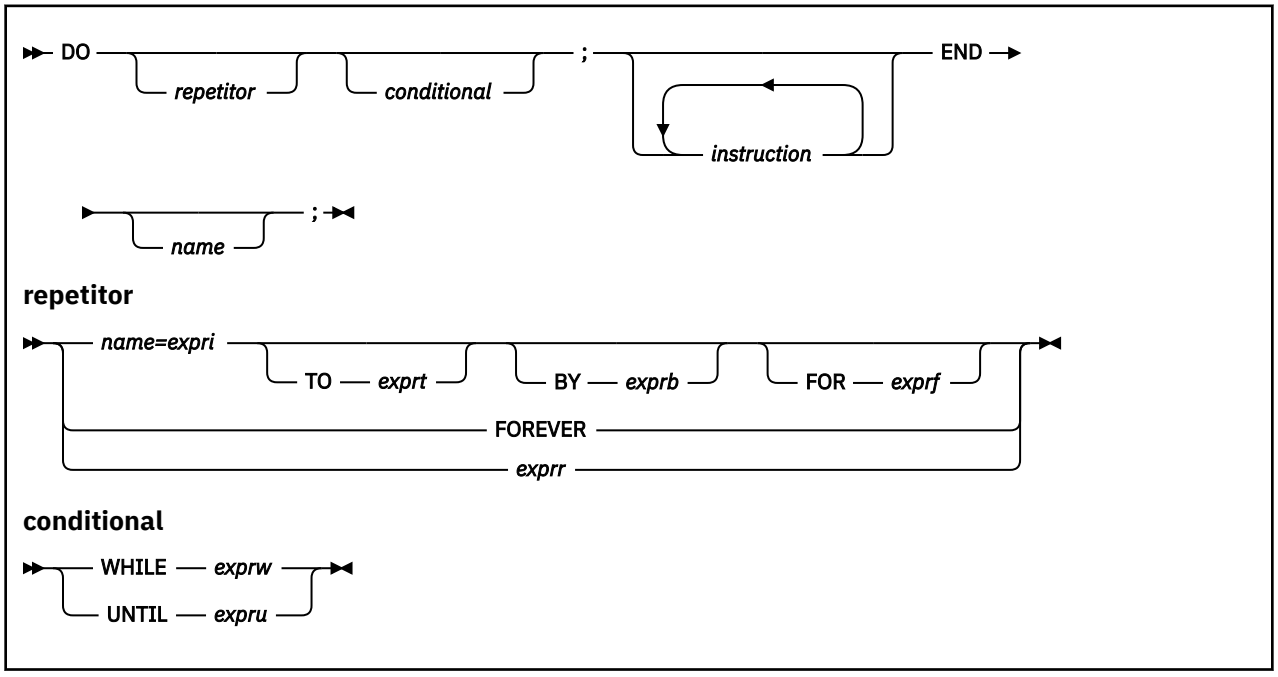
```
/* Recursive subroutine execution... */
arg z
call factorial z
say z'! =' result
exit

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 42) 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 27) 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 66.
- **Elapsed-time clocks**: A subroutine inherits the elapsed-time clock from its caller (see "TIME" on page 83), 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 43.

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 35).

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).

Note: For a discussion on conditional phrases, see “Conditional Phrases (WHILE and UNTIL)” on page 35.

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 */
```

Note: 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" */
```

Note: Using the LEAVE or ITERATE instructions can also modify the execution of repetitive loops.

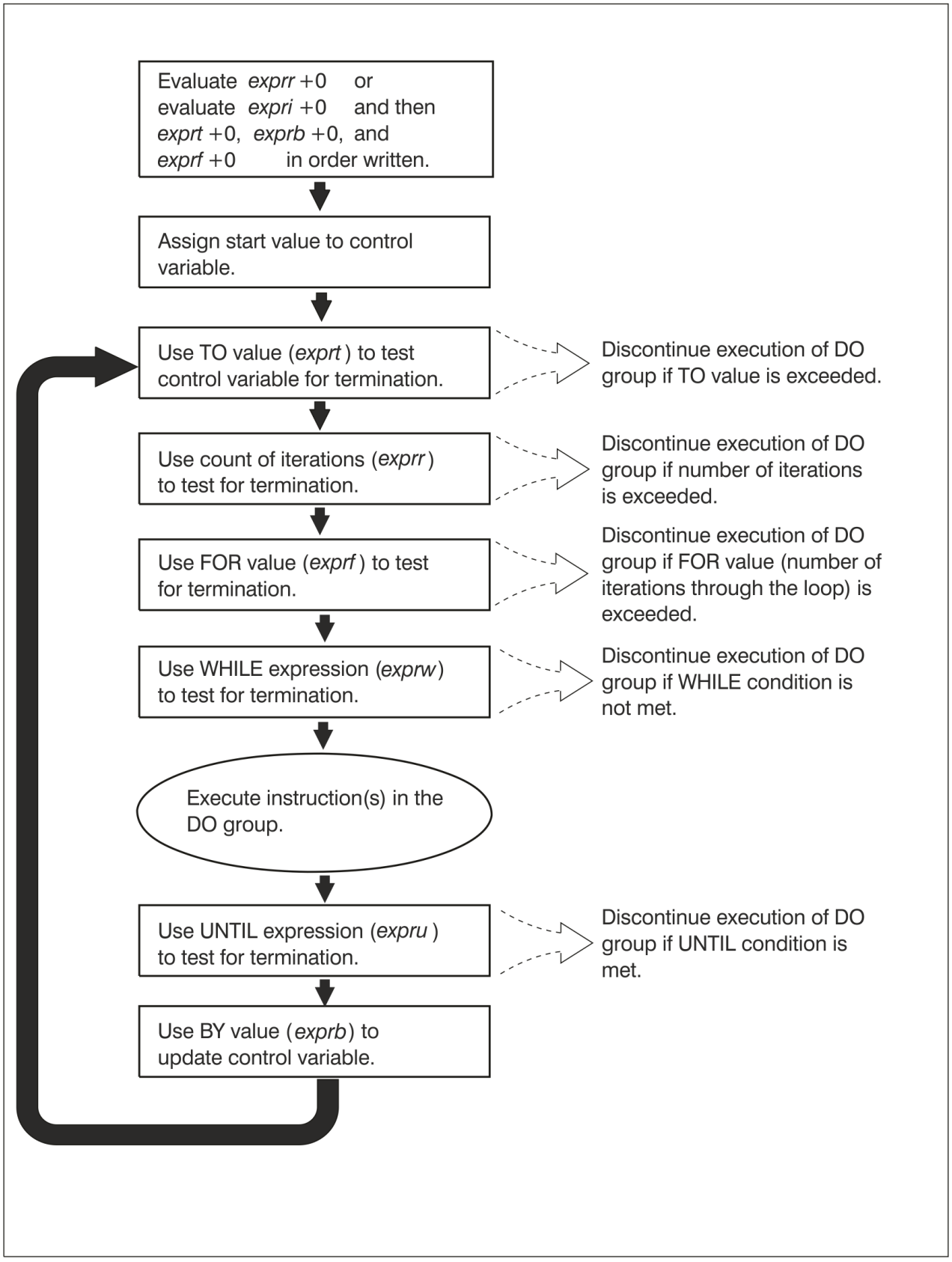
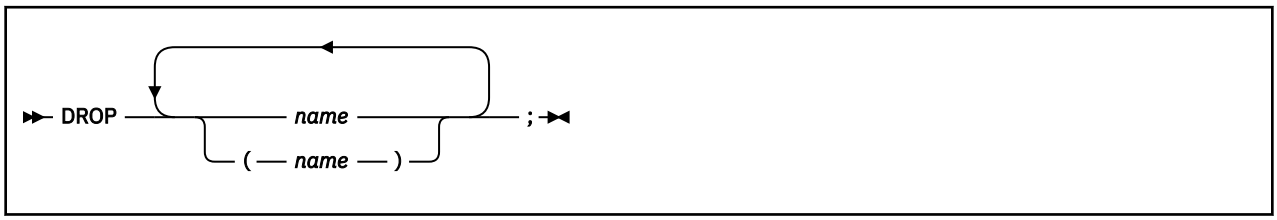


Figure 2. Concept of a DO Loop

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 46), 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 49) 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:

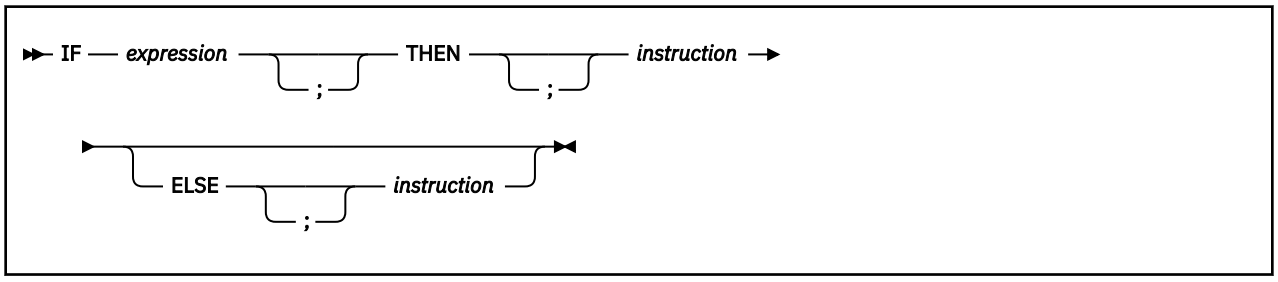
```
j=3
Exit j*4
/* Would exit with the string '12' */
```

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). Further processing of this value depends on the method of invocation of the REXX procedure (see “Calling REXX Directly with the JCL EXEC Command” on page 329).

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 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 86) 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

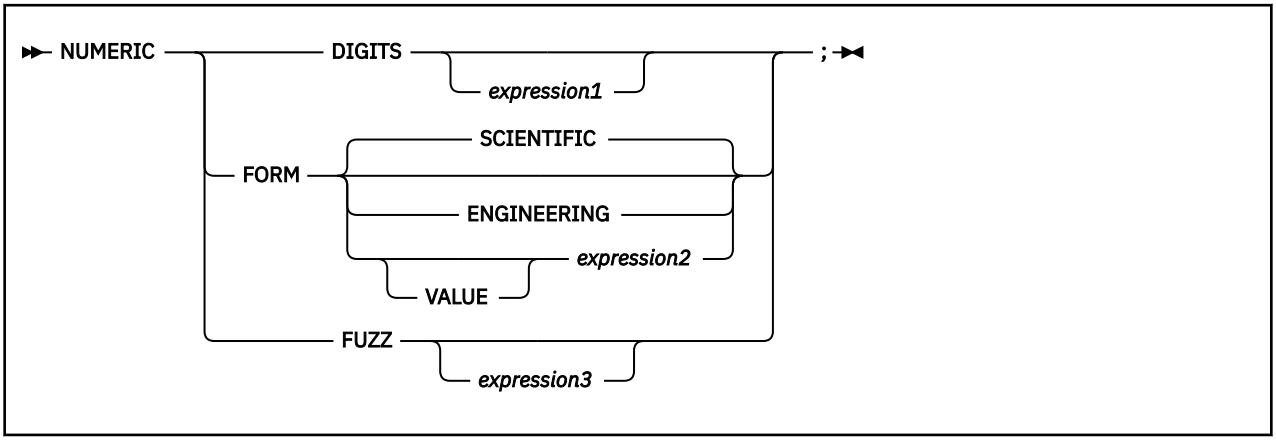
NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause:

Example:

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

Note: 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. The options of this instruction are described in detail on Chapter 6, "Numbers and Arithmetic," on page 119-"Errors" on page 126, 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 recommended 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 72.

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 74.

NUMERIC FUZZ
controls how many digits, at full precision, are ignored during a numeric comparison operation. (See "Numeric Comparisons" on page 124.) 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 75.

Note: The three numeric settings are automatically saved across internal and external subroutine and function calls. See the CALL instruction ("CALL" on page 30) for more details.

OPTIONS

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 recommended 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.

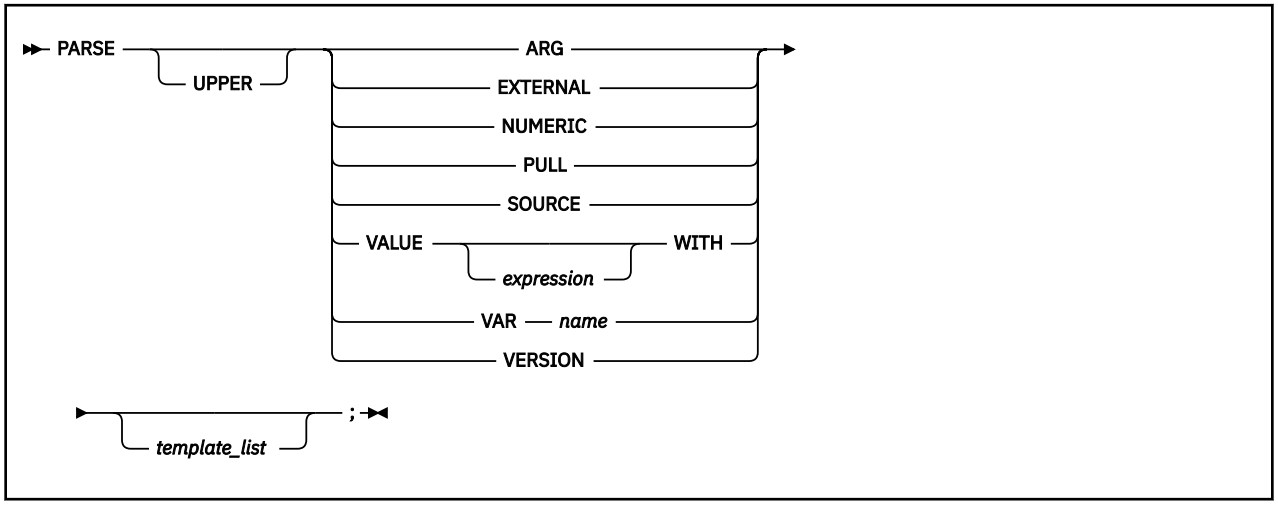
EXMODE
specifies that instructions, operators, and functions handle DBCS data in mixed strings on a logical character basis. DBCS data integrity is maintained.

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 ARX0033I. 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 105.)

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 page “Parsing Multiple Strings” on page 114 for information on 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 the ARG instruction on page “ARG” on page 29 for details and examples.)

  Note: You can also retrieve or check the argument strings to a REXX program or internal routine with the ARG built-in function.

PARSE EXTERNAL
: reads from the current input stream. ASSGN(STDIN) returns the name of the current input stream. PARSE EXTERNAL returns a field based on the record that is read. If the current input stream is SYSIPT, REXX/VSE reads SYSIPT data until encountering an end-of-data indicator, such as `/*`. If SYSIPT has no data, then PARSE EXTERNAL returns a null string. If the input stream is SYSLOG, then REXX/VSE solicits input from the operator's console. The operator receives a message containing the partition number and is asked to supply some input to the program. (If you are sending output to the console, code a pertinent SAY instruction before the PARSE EXTERNAL.)

PARSE NUMERIC
: The current numeric controls (as set by the NUMERIC instruction, see “NUMERIC” on page 42) are available. 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 42 and the built-in functions “DIGITS” on page 72, “FORM” on page 74, and “FUZZ” on page 75.

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 current input stream and the program pauses, if necessary, until a line is complete. ASSGN(STDIN) returns the name of the current input stream. If the current input stream is SYSLOG, the PULL instruction gets input from the operator's console. The operator receives the partition number and is asked to supply some input to the program. (If you are sending output to the console, code a pertinent SAY instruction before the PARSE PULL.) 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 page “QUEUED” on page 78.) The queue remains active as long as the language processor is active. for the life of the job. Other programs in the system can alter the queue and use it as a means of communication with programs written in REXX. See also the PULL instruction on page “PULL” on page 48.

PULL and PARSE PULL read from the data stack. In REXX/VSE, if the data stack is empty, PULL and PARSE PULL read from the current input stream. ASSGN(STDIN) returns the name of the current input stream. If the input stream is SYSIPT REXX/VSE reads SYSIPT data until encountering an end-of-data indicator, such as /*. If SYSIPT has no data, PULL and PARSE PULL 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 string parsed has the following general structure:

```
system_id  how_called  program_name  additional_tokens
```

system_id

This is VSE.

how_called

The string COMMAND, FUNCTION, or SUBROUTINE, depending on whether the program was called as a host command (for example as a host command from ADDRESS VSE), a function call in an expression, or through the CALL instruction.

program_name

The name of the program in uppercase. This is the member name only (no library or sublibrary name). If the name is not known, this token is a question mark (?).

additional_tokens

Note that for all of the additional tokens, if the information is not known, the token is a question mark.

- A string indicating the active chain from which the program was loaded, for example PROC.
- The name of the file from which the program was loaded. This is in the format: *library.sublibrary.membername.membertype*.
- Program name as called, not translated to uppercase. This is the name exactly as it was passed to the language processor.
- Initial (default) environment name in uppercase.
- The name of the address space in uppercase. This is from the ADDRSPN field in the parameters module.
- The token from the PARSETOK field in the parameters module (see page “PARSETOK ” on page 393).

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

```
VSE COMMAND PARSE PROC LIZH.PROC.PARSE.PROC PARSE VSE VSE ?
```

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

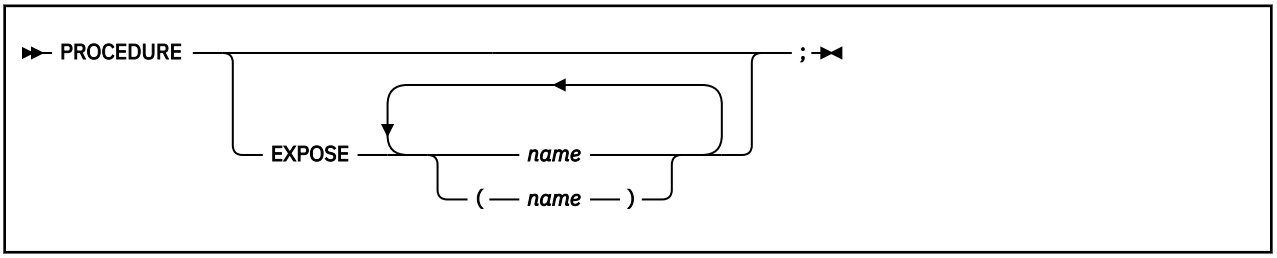
```
PARSE UPPER VAR string word1 string
```

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, `3.48`.
3. Three tokens describing the language processor release date, for example, "31 May 1993". `3 May 1993`. The date, month, and year are in the format `dd mon yyyy`, the same format as the default for the DATE function.

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 “Stems” on page 21 for information about stems.)

Example:

```
/* This is the main REXX program */
a.=11; i=13; j=15
i = i + 1
```

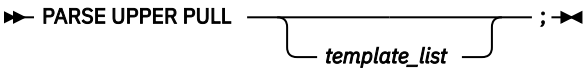
```
C.5 = 'FRED'
call lucky7
say a. a.1 i j c. c.5
say 'You should see 11 7 14 15 C. FRED'
exit
lucky7:Procedure Expose i j a. c.
/* This exposes I, J, and all variables whose     */
/* names start with A. or C.                      */
A.1='7'  /* This sets A.1 in the caller's         */
         /* 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 the CALL instruction and function descriptions on "CALL" on page 30 and Chapter 4, "Functions," on page 59 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 ("Parsing Rules" on page 105). Use the PARSE PULL instruction if you do not desire uppercase translation.

The REXX/VSE implementation of the external data queue is the data stack. REXX programs can use the data stack. In REXX/VSE, if the data stack is empty, PULL reads from the current input stream. ASSGN(STDIN) returns the name of the current input stream. If the current input stream is SYSIPT, REXX/VSE reads SYSIPT data until encountering an end-of-data indicator, such as /*. If SYSIPT has no data, the PULL instruction returns a null string. If the current input stream is SYSLOG, then REXX/VSE solicits input from the operator's console. The operator receives a message containing the partition number and is asked to supply some input to the program. (If you are sending output to the console, code a pertinent SAY instruction before the PULL.)

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 78) returns the number of lines currently in the

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 REXX/VSE 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 MAKEBUF.

Example:

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

The QUEUED built-in function (described on "QUEUED" on page 78) 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 REXX/VSE 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 MAKEBUF.

Example:

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

The QUEUED built-in function (described on "QUEUED" on page 78) 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 37.)

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 30.)

If a *function* is being processed, the action taken is identical, except that *expression* **must** be specified on the RETURN instruction. The result of *expression* is then used in the original expression at the point where the function was called. See the description of functions on Chapter 4, "Functions," on page 59 for more details.

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.

SAY

SAY writes a line to the current output stream. ASSGN(STDOUT) returns the name of the current output stream. If the output stream is SYSLOG, REXX/VSE sends the data to the operator's console. (NOMSGIO and NOMSGWTO in the PARMBLOCK FLAGS flag byte determine the REXX processing rules that affect sending this data. "Flags and Corresponding Masks" on page 393 describes the flags.) Along with the actual output, REXX/VSE sends the partition number of the job producing the output.

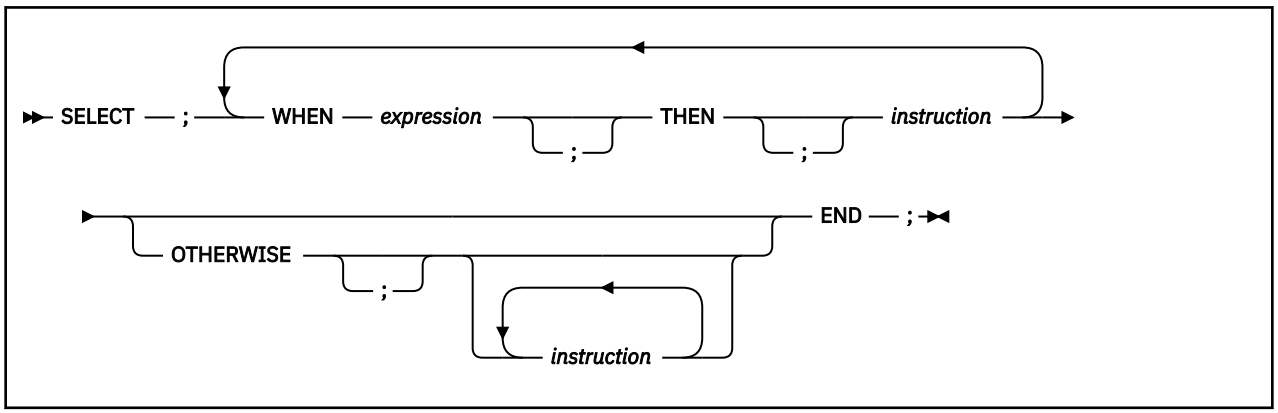
Note: VSE/ESA replaces any non-displayable character with a blank if SYSLOG is receiving the output.

The result of *expression* may be of any length. If you omit *expression*, the null string is written.

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).

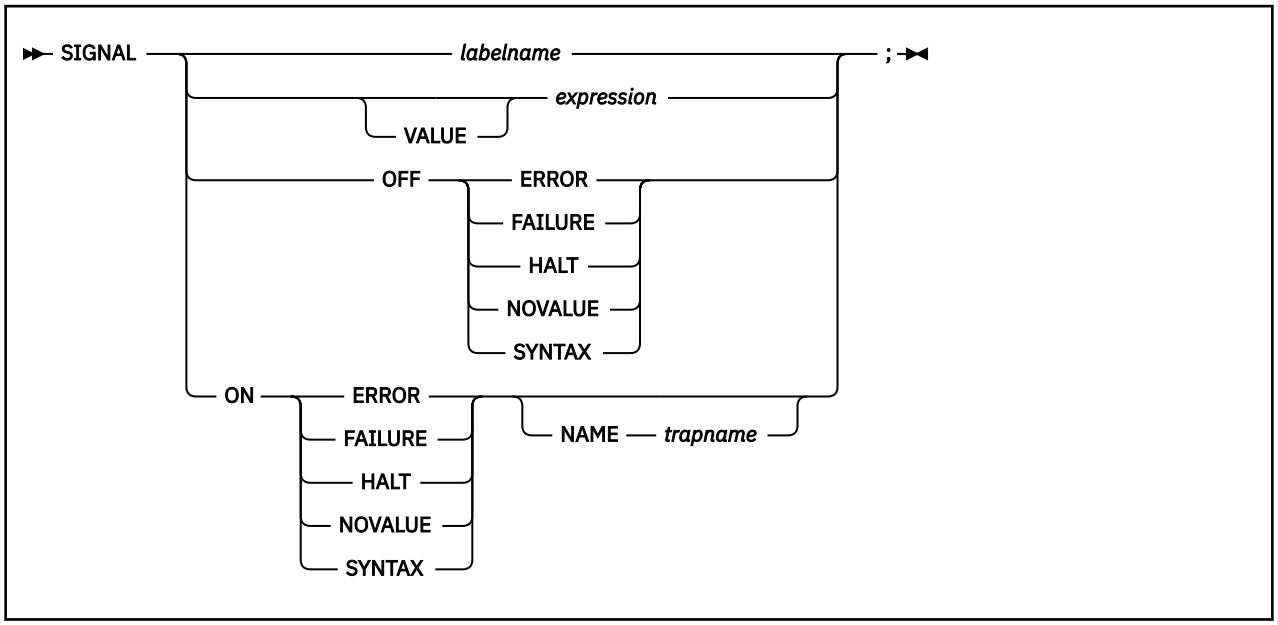
Example:

```
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 'Warning, 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 on condition traps is contained in Chapter 7, “Conditions and Condition Traps,” on page 129.

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.

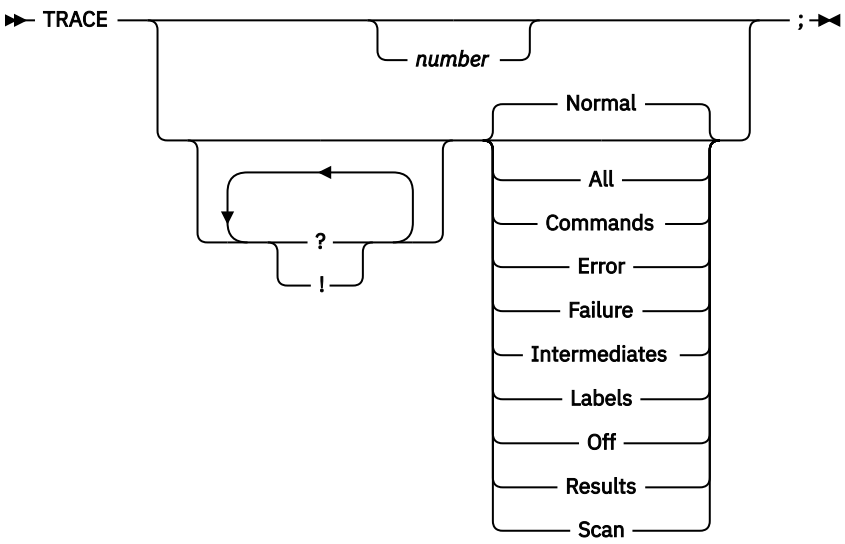
Using SIGNAL VALUE

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.

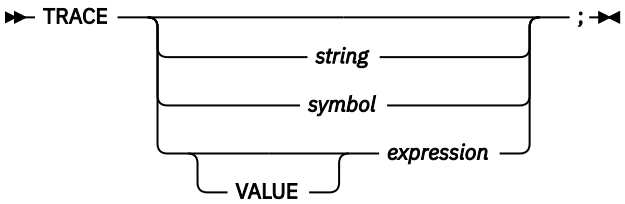
Example:

```
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 sent to the output stream) 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. (In a batch environment, the interaction is between the current input stream and the program. ASSGN(STDOUT) returns the name of the current output stream, and ASSGN(STDIN) returns the name of the current input stream. Tracing and interactive debug use the same input and output streams.)

TRACE writes to the current output stream. If the output stream is SYSLOG, REXX/VSE sends the data to the operator's console. Along with the actual output, REXX/VSE sends the partition number of the job producing the output.

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.

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 recommended 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 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:

[3] See "Commands" on page 23 for definitions of error and failure.

?

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 319 for full details of this facility.) While interactive debug is on, interpretation pauses after most clauses that are traced. (If you are working from the operator's console, these pauses occur. If you are using files for input and output, interactive debug reads the next line instead of pausing. The term *pause* is used generically in this description. It means the activity that is usual for the input stream you are using. Similarly, this description mentions information you *enter*; this means information you input using the method appropriate for your current input stream.)

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). If the current input stream provides a null string as input, processing continues to the next instructon. (If the current input stream is SYSIPT, the language processor strips trailing blanks.) The current input stream can also provide an instruction for execution.

When interactive debug is starting, a message indicating this is sent to the current output stream. While interactive debug is active, it reads from the current input stream. (If the current input stream is a file, it is read one line at a time, and a null string is returned when there are no more lines to read. If the current input stream is SYSLOG, interactive debug reads from the operator's console.)

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 0 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.)

Note: You can start interactive debug by using the TS immediate command in a REXX program or by specifying TS on a call to ARXIC from a non-REXX program. See Chapter 10, "REXX/VSE Commands," on page 143 for more information about immediate commands and "TS" on page 167 for more information about TS.

!

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 0 or TRACE with no options.

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 319, 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 84).
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 the CALL instruction ("CALL" on page 30) 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.     */
```

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 106).

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 20.)

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.

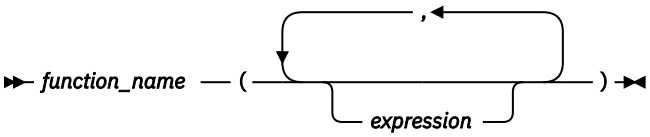
For more complete information, see the VM/ESA REXX/VM Reference.

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 REXX/VSE, 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 82) 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 the CALL instruction (“CALL” on page 30) 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 51).

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 61—"X2D (Hexadecimal to Decimal)" on page 89.)

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 344.

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 page "FUNCSOFL " on page 394) indicates whether to search the active PHASE chain or the PROC chain first.

   If the flag is off, check the active PHASE chain. If a match to the function name is not found, search the PROC chain.

   If the flag is on, search the PROC chain. If a match to the function name is not found, check the active PHASE chain.

   Note: By default, the FUNCSOFL flag is off, which indicates searching the active PHASE chain before searching for a REXX program.

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.

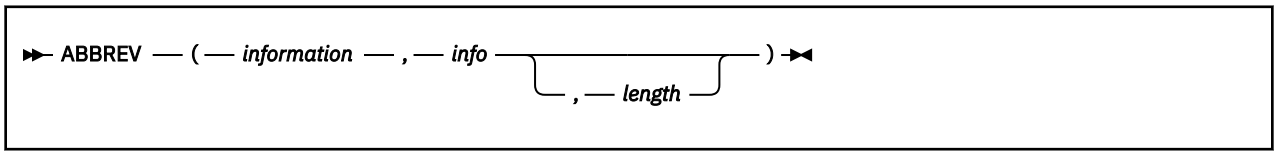
In addition to the functions SAA REXX provides, REXX/VSE has six additional built-in functions: EXTERNALS, FIND, INDEX, JUSTIFY, LINESIZE, and USERID. If you plan to write REXX programs that run on other SAA environments, note that these functions are not available to all the environments. In this section, these six built-in functions are identified as non-SAA functions.

In addition to the built-in functions, REXX/VSE also provides external functions that you can use to perform different tasks. "External Functions" on page 92 describes these functions. 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.
- 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 76.)

- 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 topic support DBCS. A complete list and descriptions of these functions are in Chapter 22, “Double-Byte Character Set (DBCS) Support,” on page 479.

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
```

Note: A null string always matches if a length of 0 (or the default) is used. This allows a default keyword to be selected automatically if desired; for example:

```
say 'Enter option:';   pull option .
select  /* keyword1 is to be the default */
  when abbrev('keyword1',option) then ...
  when abbrev('keyword2',option) then ...
  ...
  otherwise nop;
end;
```

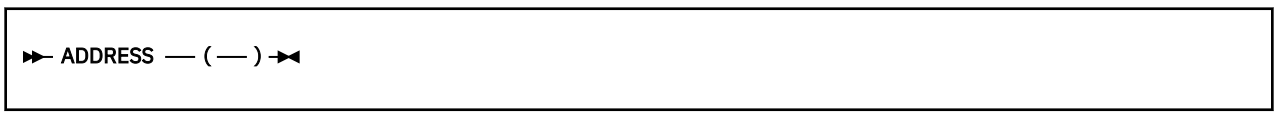
ABS (Absolute Value)

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
```

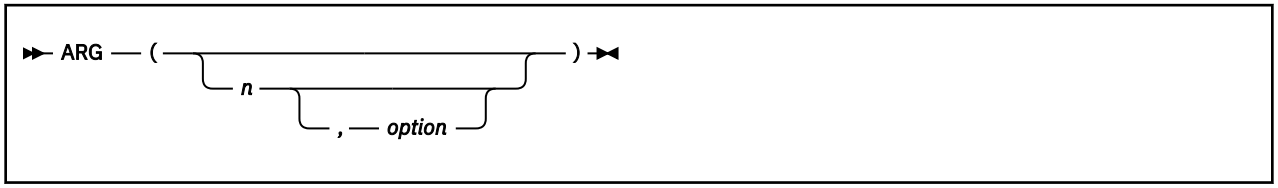
ADDRESS



returns the name of the environment to which commands are currently being submitted. See the ADDRESS instruction (page "ADDRESS" on page 27) for more information. Trailing blanks are removed from the result. Here are some examples:

```
ADDRESS()    ->    'VSE'        /* default under VSE    */
ADDRESS()    ->    'POWER'      /* assumes address change */
```

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
```

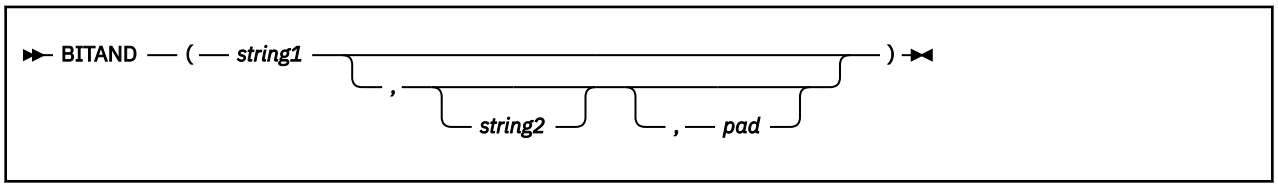
Note:

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 29, "PARSE" on page 44, and "Parsing Rules" on page 105.)

ASSGN

ASSGN is an external function. See page "ASSGN" on page 93 for a description.

BITAND (Bit by Bit AND)

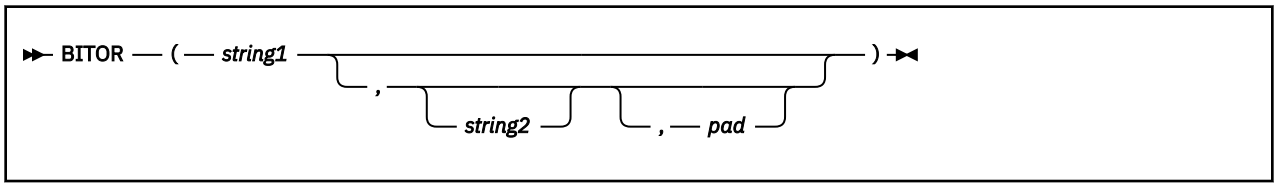


returns a string composed of the two input strings logically ANDed together, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the AND operation stops when the shorter 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  */
```

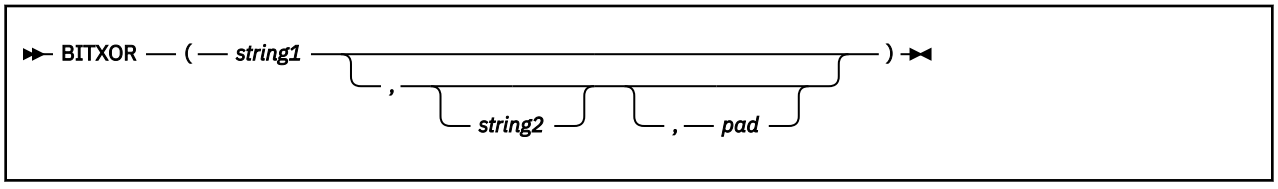
BITOR (Bit by Bit OR)



returns a string composed of the two input strings logically inclusive-ORed together, bit by bit. (The encodings 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 encodings 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.

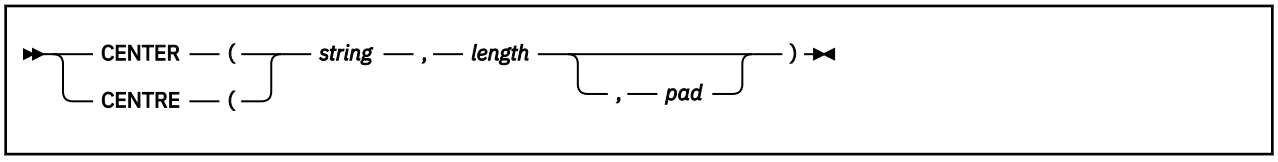
Here are some examples:

```
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



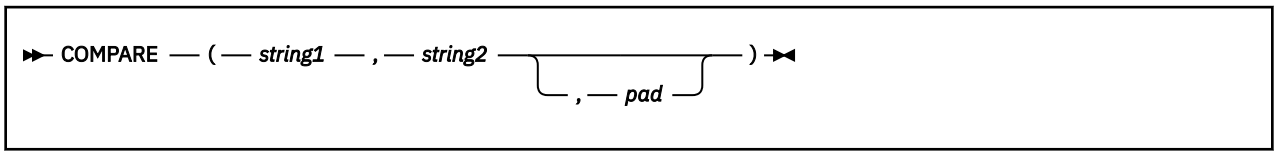
returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up length. The *length* must be a positive whole number or zero. The default *pad* character is blank. If the string is longer than *length*, it is truncated at both ends to fit. If an odd 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 '
```

Note: 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 129 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 132 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 */
```

Note: 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

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 you do not specify *n*, *string* is processed as an unsigned binary number.

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`.

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)

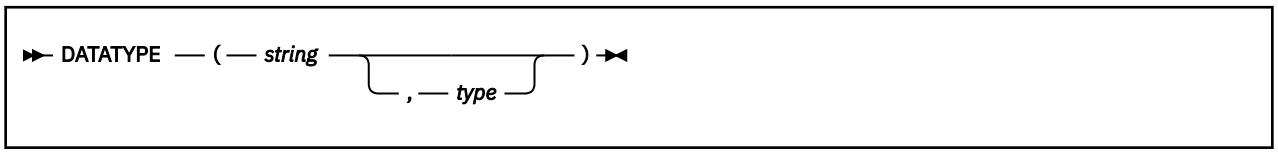
returns a string, in character format, that represents *string* converted to hexadecimal. The returned string contains twice as many bytes as the input string. For example, on an EBCDIC system, C2X(1) returns F1 because the EBCDIC representation of the character 1 is 'F1'X.

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 9.) 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.

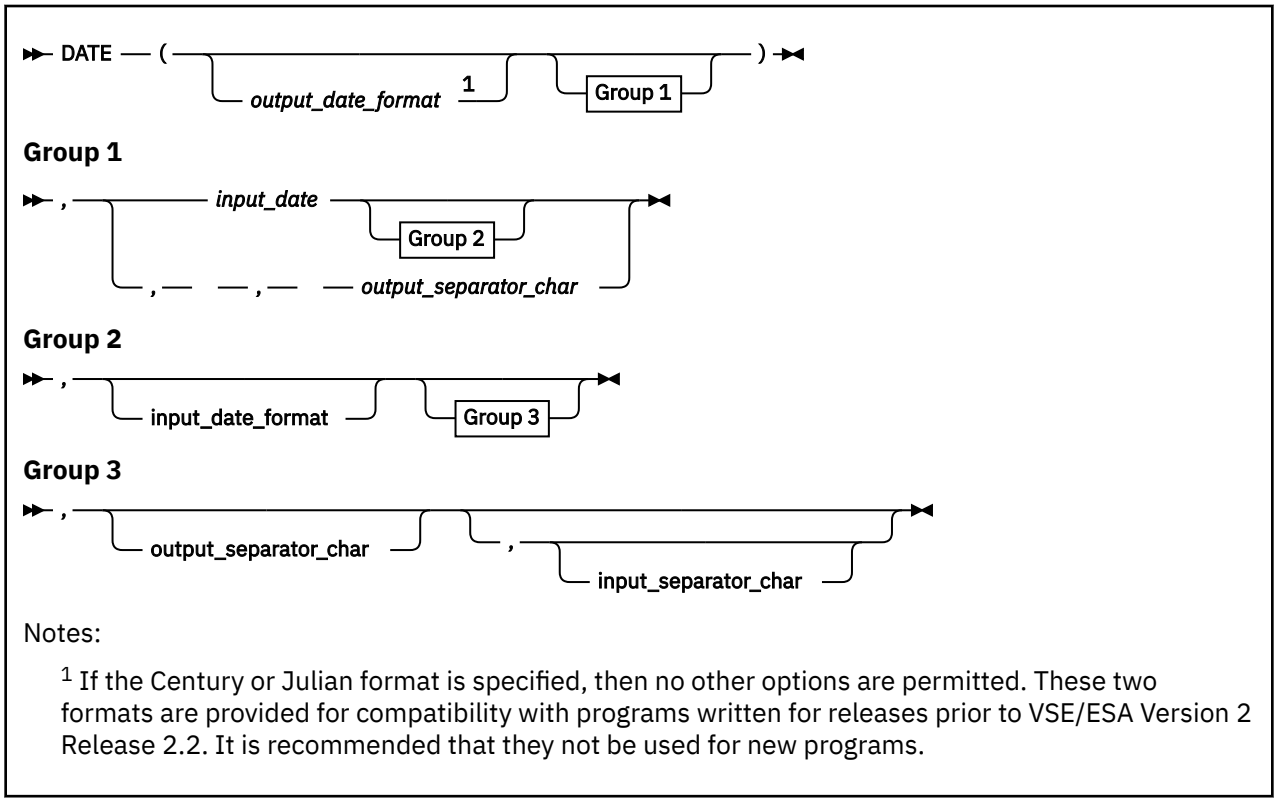
heXadecimal
returns 1 if *string* contains only characters from the ranges a–f, A–F, 0–9, and blank (as long as blanks appear only between pairs of hexadecimal characters). Also returns 1 if *string* is a null string, which is a valid hexadecimal string.

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
```

Note: 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



Notes:

[1] If the Century or Julian format is specified, then no other options are permitted. These two formats are provided for compatibility with programs written for releases prior to VSE/ESA Version 2 Release 2.2. It is recommended that they not be used for new programs.

returns, by default, the local date in the format: *dd mon yyyy* (day, month, year—for example, 25 Dec 1998), with no leading zero or blank on the day. Otherwise, the string *input_date* is converted to the format specified by *output_date_format*. *input_date_format* can be specified to define the current format of *input_date*. The default for *input_date_format* and *output_date_format* is **N**ormal.

input_separator_char and *output_separator__char* can be specified to define the separator character for the input and output dates, respectively. Any single non-alphanumeric character is valid. See note “3” on page 71 for more information.

You can use the following options to obtain specific date formats. (Only the bold character is needed; all characters following it 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. Note that REXX/VSE supports US English only.

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 2 January 2000 returns 2, the number of days from 1 January 2000 to 2 January 2000.

Note: When the Century option is used for input, the output may change, depending on the current century. For example, if DATE('S','1',C) was entered on any day between 1 January 1900 and 31 December 1999, the result would be 19000101. However, if DATE('S','1',C) was entered on any day between 1 January 2000 and 31 December 2099, the result would be 20000101. It is important to understand the above, and code accordingly.

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*.

Month
full English name of the current month, for example, `August`. Only valid for *output_date_format*.

Normal
date in the format: *dd mon yyyy*. **This is the default**. (*dd* cannot have any leading zeros or blanks; *yyyy* must have leading zeros but cannot have any leading blanks). The abbreviated form of the month name is used (for example, "Jan", "Feb", and so on).

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 *output_date_format*.

Here are some examples, assuming today is 13 March 1992:

```
DATE()                              ->    '13 Mar 1992'
DATE(,'19960527','S')               ->    '27 May 1996'
DATE('B')                           ->    '727269'
DATE('B','27 May 1996',)            ->    '728805'
DATE('B','27*May*1996',,,'*')       ->    '728805'
DATE('C')                           ->    '33675'
DATE('E')                           ->    '13/03/92'
DATE('E',,,'+')                     ->    '13+03+92'
```

```
DATE('E','081698','U',,'')          ->    '16/08/98'
DATE('J')                           ->    '92073'
DATE('M')                           ->    'March'
DATE('N')                           ->    '13 Mar 1992'
DATE('N','35488','C')               ->    '28 Feb 1997'
DATE('O')                           ->    '92/03/13'
DATE('S')                           ->    '19920313'
DATE('S',,)                         ->    '19920313'
DATE('S',,,'-')                     ->    '1992-03-13'
DATE('U')                           ->    '03/13/92'
DATE('U','96/05/27','O')            ->    '05/27/96'
DATE('U','97059','J')               ->    '02/28/97'
DATE('U','1.Feb.1998,'N','+','.')   ->    '02+01+98'
DATE('U','1998-08-16','S','','-')   ->    '081698'
DATE('W')                           ->    'Friday'
```

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 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
3. *input_separator_char* and *outputy_separator_char* apply to the following formats, and have the following default values:

Format Name	Format Structure	Default Separator Value
European	dd/mm/yy	'/'
Normal	dd mon yyyy	' '
Ordered	yy/mm/dd	'/'
Standard	yyyymmdd	''
Usa	mm/dd/yy	'/'

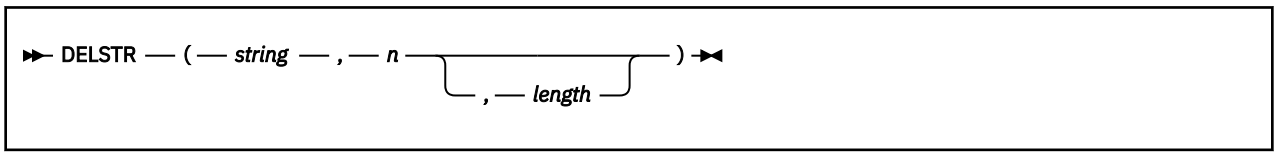
   Note that Null is a valid value for *input_separator_char* and *output_separator_char*.

DBCS (Double-Byte Character Set Functions)

The following are all part of DBCS processing functions. See Chapter 22, “Double-Byte Character Set (DBCS) Support,” on page 479.

DBADJUST	DBRIGHT	DBUNBRACKET
DBBRACKET	DBRLEFT	DBVALIDATE
DBCENTER	DBRRIGHT	DBWIDTH
DBCJUSTIFY	DBTODBCS	
DBLEFT	DBTOSBCS	

DELSTR (Delete String)

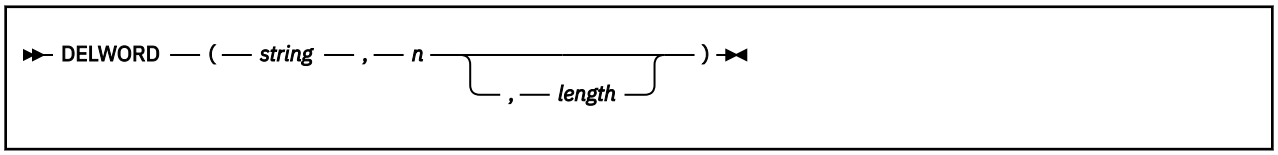


returns *string* after deleting the substring that begins at the *n*th character and is of *length* characters. If you omit *length*, or if *length* is greater than the number of characters from *n* to the end of *string*, the function deletes the rest of *string* (including the *n*th character). The *length* must be 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'
```

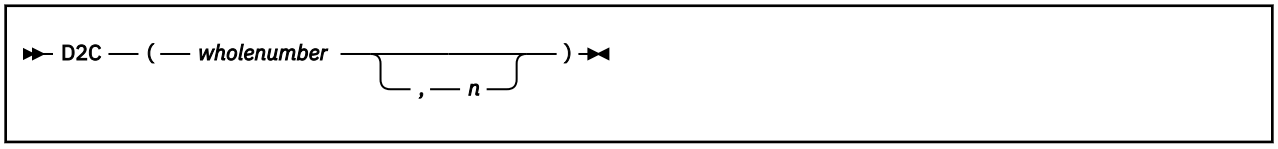
DIGITS

returns the current setting of NUMERIC DIGITS. See the NUMERIC instruction on "NUMERIC" on page 42 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.

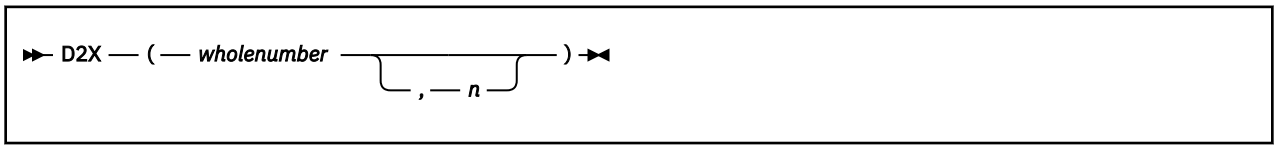
If you omit *n*, *wholenumber* must be a positive whole number or zero, and the result length is as needed. Therefore, the returned result has no leading '00'x characters.

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)



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

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. See z/VSE Messages and Codes for a complete description of error numbers and messages.

Here are some examples:

```
ERRORTEXT(16)    ->    'Label not found'
ERRORTEXT(60)    ->    ''
```

EXTERNALS

This is a non-SAA built-in function. See “EXTERNALS” on page 90 for a description.

FIND

WORDPOS is the preferred built-in function for this type of word search; see page “WORDPOS (Word Position)” on page 88 for a complete description. FIND is a non-SAA built-in function. See “FIND” on page 90 for a description.

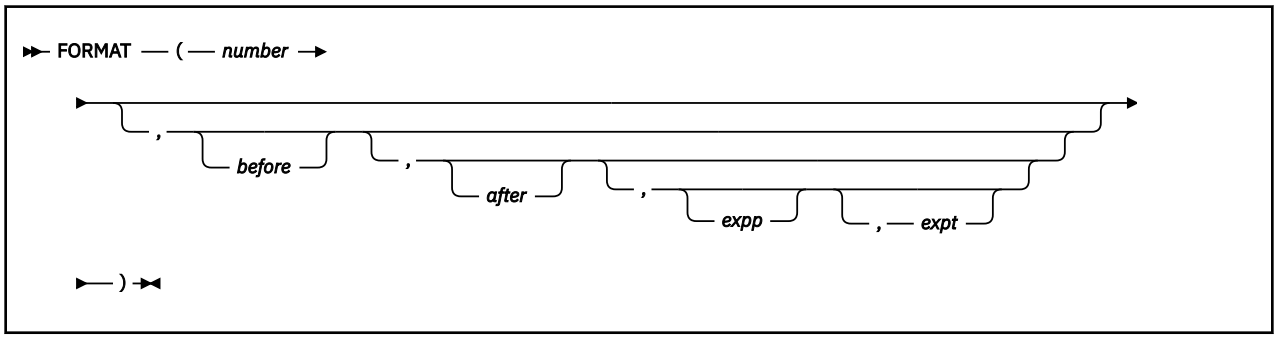
FORM

returns the current setting of NUMERIC FORM. See the NUMERIC instruction on “NUMERIC” on page 42 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

returns the current setting of NUMERIC FUZZ. See the NUMERIC instruction on "NUMERIC" on page 42 for more information.

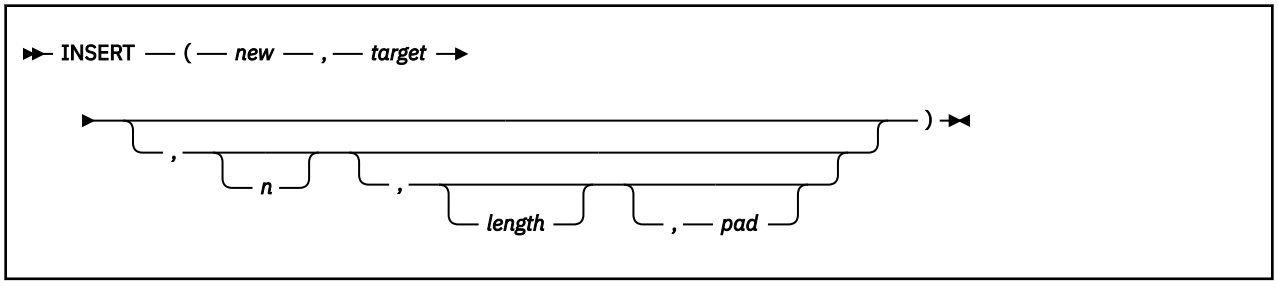
Here is an example:

```
FUZZ()    ->    0     /* by default */
```

INDEX

POS is the preferred built-in function for obtaining the position of one string in another; see "POS (Position)" on page 78 for a complete description. INDEX is a non-SAA built-in function. See "INDEX" on page 91 for a description.

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.

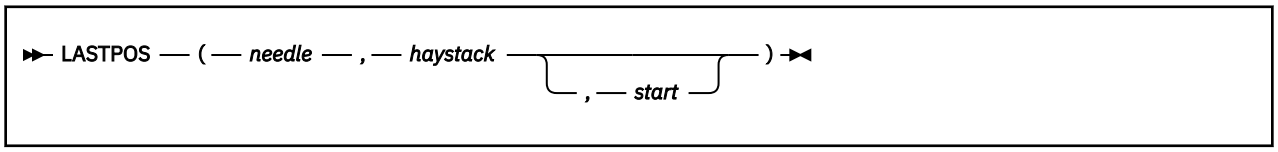
Here are some examples:

```
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

This is a non-SAA built-in function. See "JUSTIFY" on page 91 for a description.

LASTPOS (Last Position)

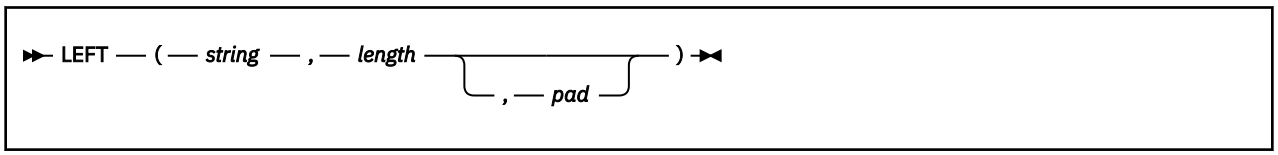


returns the position of the last occurrence of one string, *needle*, in another, *haystack*. (See also the POS function.) Returns `0` if *needle* is the null string or is not found. By default the search starts at the last character of *haystack* and scans backward. You can override this by specifying *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.

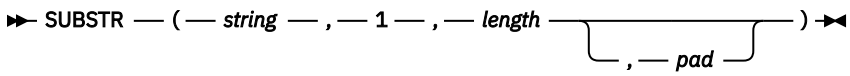
Here are some examples:

```
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

returns the length of *string*.

Here are some examples:

```
LENGTH('abcdefgh')    ->    8
LENGTH('abc defg')    ->    8
LENGTH('')            ->    0
```

LINESIZE

This is a non-SAA built-in function. See "LINESIZE" on page 92 for a description.

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:

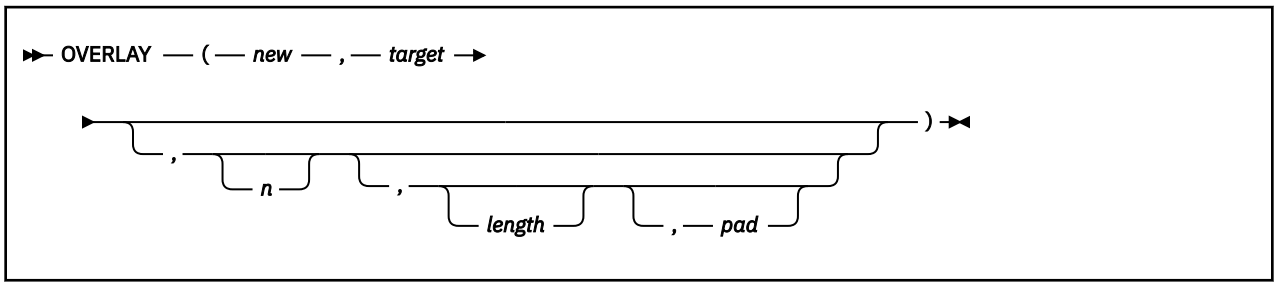
```
MIN(12,6,7,9)                                                    ->   6
MIN(17.3,19,17.03)                                               ->  17.03
MIN(-7,-3,-4.3)                                                  ->  -7
MIN(21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,MIN(2,1))  ->  1
```

Implementation maximum: You can specify up to 20 *number*s, and can nest calls to MIN if more arguments are needed.

OUTTRAP

OUTTRAP is an external function. See page "OUTTRAP" on page 94.

OVERLAY

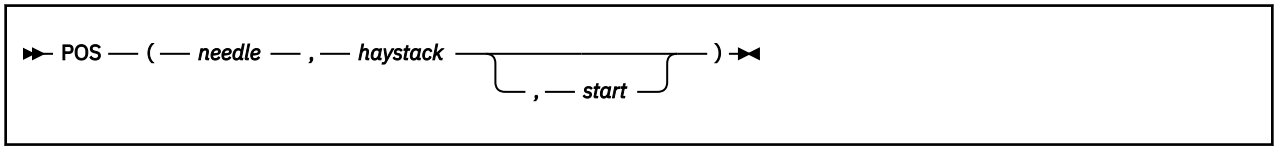


returns the string *target*, which, starting at the *n*th character, is overlaid with the string *new*, padded or truncated to length *length*. (The overlay may extend beyond the end of the original *target* string.) If you specify *length*, it must be a positive whole number or zero. The default value 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
```

QUEUED

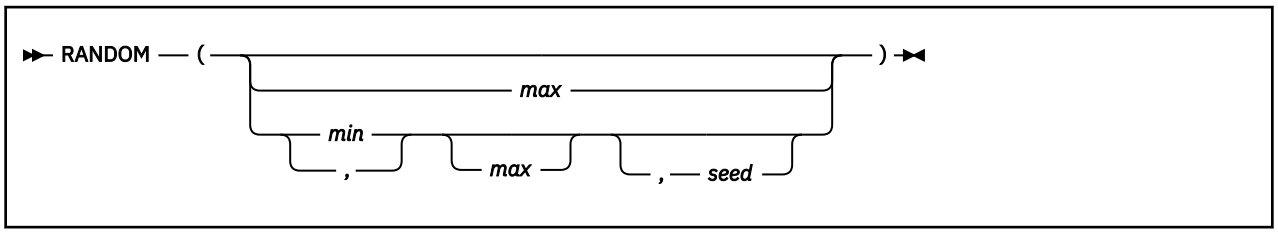
returns the number of lines remaining in the external data queue when the function is called.

The REXX/VSE 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 79. 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.

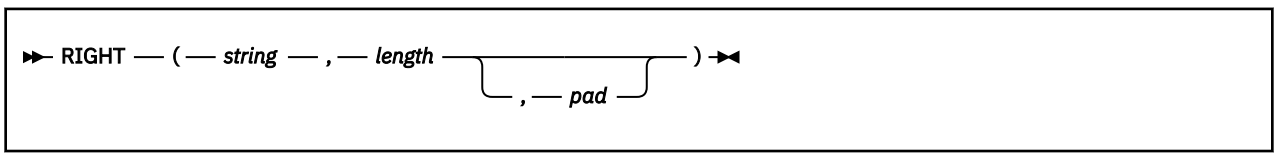
REVERSE

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'
```

REXXIPT

REXXIPT is an external function. See page "REXXIPT" on page 98.

REXXMSG

REXXMSG is an external function. See page "REXXMSG" on page 99.

SETLANG

SETLANG is an external function. See page "SETLANG" on page 99.

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
```

SLEEP

SLEEP is an external function. See page "SLEEP" on page 100.

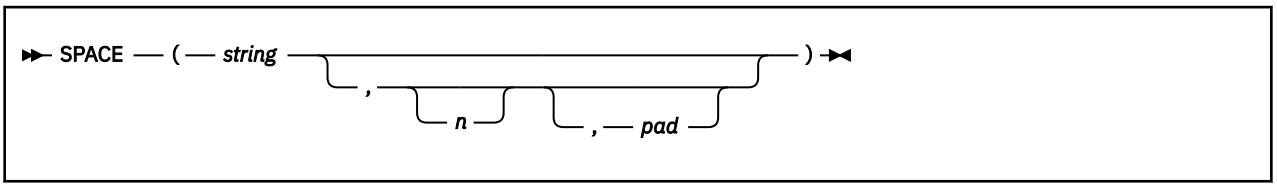
SOURCELINE

returns the line number of the final line in the program if you omit returns the line number of the final line in the program if you omit *n*, or returns the *n*th line in the program if you specify *n*. If specified, *n* must be a positive whole number and must not exceed the number of the final line in the program.

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.

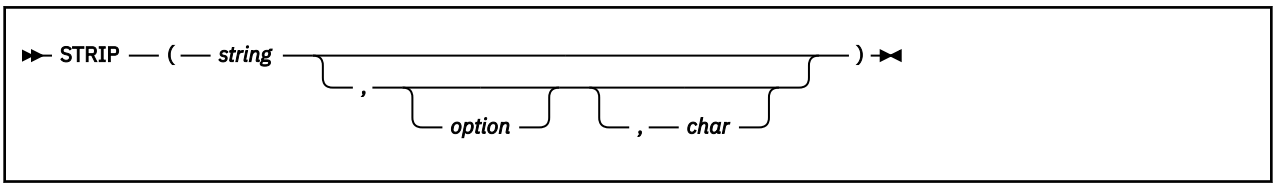
Here are some examples:

```
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 an external function. See page "STORAGE" on page 101.

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*.

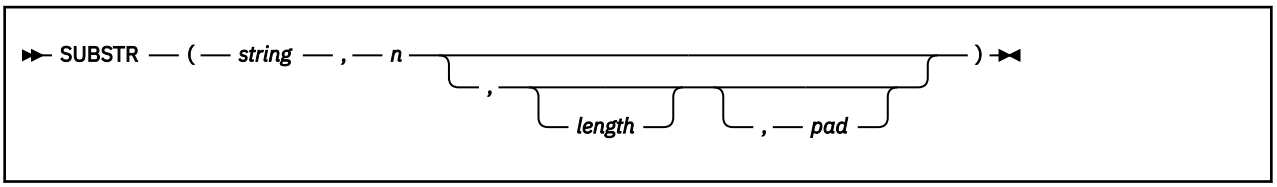
Trailing
removes trailing characters from *string*.

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.

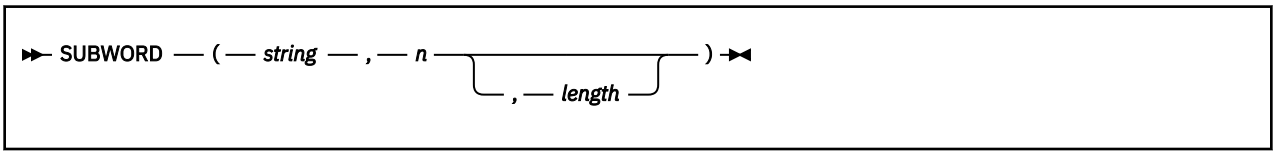
If you omit *length*, the rest of the string is returned. The default *pad* character is a blank.

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

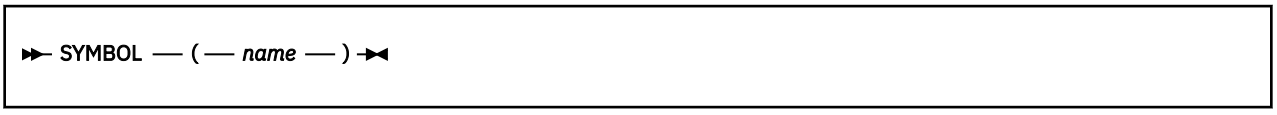


returns the substring of *string* that starts at the *n*th word, and is up to *length* blank-delimited words. The *n* must be a positive whole number. If you omit *length*, it defaults to the number of remaining words in *string*. The returned string never has leading or trailing blanks, but includes 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.

Note: 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 */
```

SYSVAR

SYSVAR is an external function. See page "SYSVAR" on page 102.

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 ssssssss.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 ssssssss.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 */
```

Note: 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.

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.

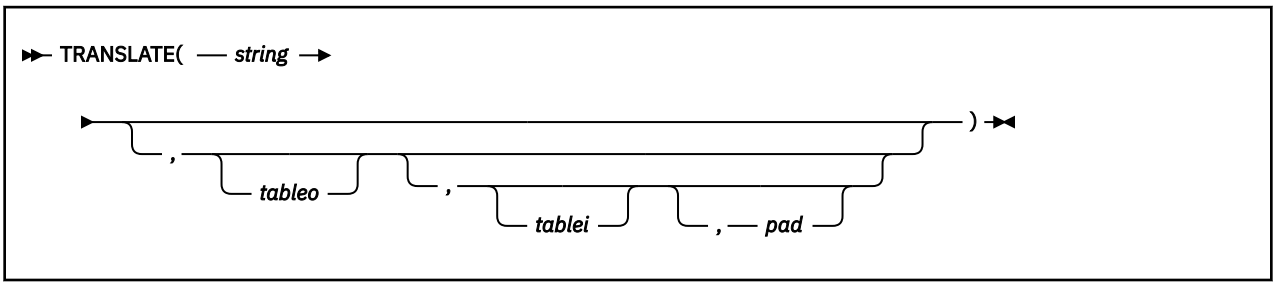
If you specify *option*, it selects the trace setting. It must be one of the valid prefixes `?` or `!` or one of the alphabetic character options associated with the TRACE instruction (that is, starting with A, C, E, F, I, L, N, O, R, or S) or both. (See the TRACE instruction on "Alphabetic Character (Word) Options" on page 54 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*.

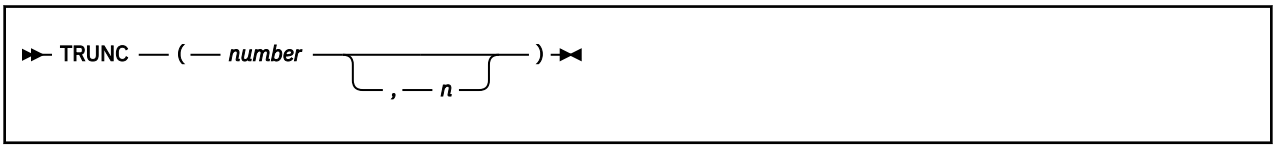
The tables can be of any length. If you specify neither translation table and omit *pad*, *string* is simply translated to uppercase (that is, lowercase a–z to uppercase A–Z), but, if you include *pad*, the language processor translates the entire string to *pad* characters. *tablei* defaults to `XRANGE('00'x,'FF'x)`, 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'
```

Note: 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:

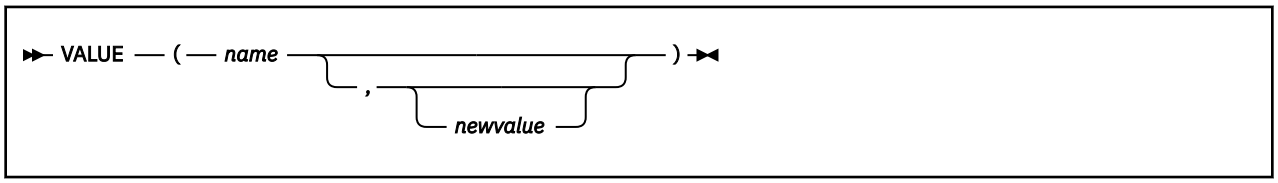
```
TRUNC(12.3)           ->    12
TRUNC(127.09782,3)    ->    127.097
TRUNC(127.1,3)        ->    127.100
TRUNC(127,2)          ->    127.00
```

Note: The *number* is rounded according to the current setting of NUMERIC DIGITS if necessary before the function processes it.

USERID

USERID is a non-SAA built-in function. See "USERID" on page 92 for a description.

VALUE



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. *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 20) 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.

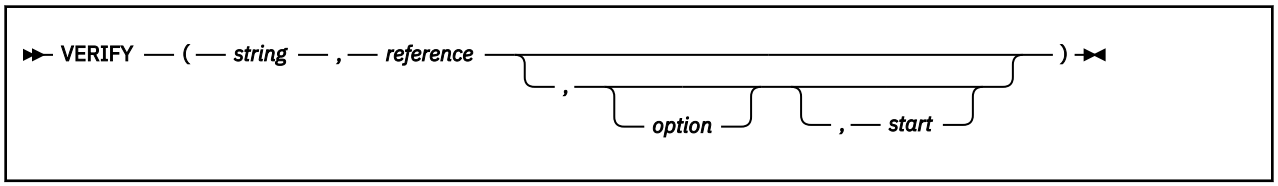
Here are some examples:

```
/* After: Drop A3; A33=7; K=3; fred='K'; list.5='Hi' */
VALUE('a'k)     ->  'A3' /* looks up A3     */
VALUE('a'k||k)  ->  '7'  /* looks up A33    */
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 */
```

Note: 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 string, 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 129.)

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
```

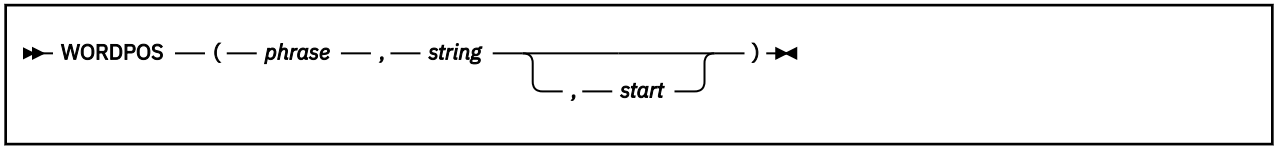
WORDLENGTH

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
```

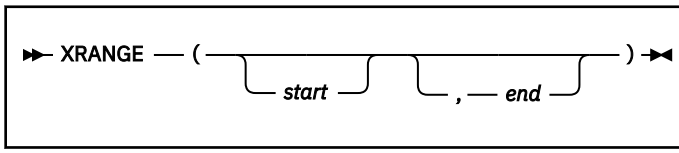
WORDS

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
```

X2B (Hexadecimal to Binary)

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)

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('c6 f0'X)    ->    240          /*  EBCDIC  */
```

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.

Additional Functions Provided in REXX/VSE

In addition to the SAA-defined built-in functions, REXX/VSE provides the following built-in functions:

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 REXX/VSE there is no equivalent buffer. Therefore, the EXTERNALS function always returns a 0.

FIND

WORDPOS is the preferred built-in function for this type of word search. See page "WORDPOS (Word Position)" on page 88 for a complete description.

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:

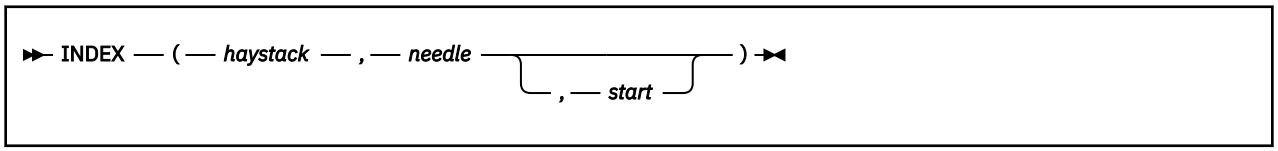
```
FIND('now is the time','is the time')    ->    2
FIND('now is  the time','is    the')     ->    2
FIND('now is  the time','is  time ')     ->    0
```

Note that WORDPOS is the preferred built-in function for this type of word search.

For more complete information, see the z/VM REXX/VM Reference.

INDEX

POS is the preferred built-in function for obtaining the position of one string in another. See page "POS (Position)" on page 78 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.

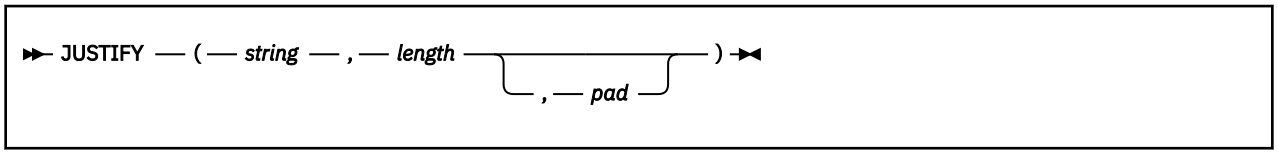
Here are some examples:

```
INDEX('abcdef','cd')      ->    3
INDEX('abcdef','xd')      ->    0
INDEX('abcdef','bc',3)    ->    0
INDEX('abcabc','bc',3)    ->    5
INDEX('abcabc','bc',6)    ->    0
```

Note that POS is the preferred built-in function for obtaining the position of one string in another.

For more complete information, see the z/VM REXX/VM Reference.

JUSTIFY



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'
```

For more complete information, see the z/VM REXX/VM Reference.

LINESIZE

returns the width of the current output device. If the current output destination is SYSLOG, LINESIZE returns 66. If it is SYSLST, LINESIZE returns 120. You can use ASSGN(STDOUT) to return the name of the current output device.

USERID

returns one of the following values:

1. The last user ID specified on the SETUID command, or, if none,
2. The user ID of the calling REXX program, if one REXX program calls another, or, if none,
3. The user ID under which the job is running, or, if none,
4. The job name.

The USERID function returns the first value that does not have a null value. For example, if the user ID specified on SETUID is null, USERID returns the user ID under which the job is running.

There are several ways to specify the user ID, not limited to the following:

- On the POWER JOB card
- The logon userid/password passed through the PWRSPL macro when you submit a job from the interactive interface (ICCF)
- On the REXX/VSE command SETUID. (See page "SETUID" on page 165 for details.)

You can replace the routine (phase) that is called to determine the value the USERID function returns. This is known as the user ID replaceable routine; and "User ID Routine" on page 465 describes it. See Chapter 21, "Replaceable Routines and Exits," on page 439 for details about replaceable routines and any exceptions to this rule.

For more complete information, see the z/VM REXX/Reference.

External Functions

You can use the following external functions to perform different tasks:

- ASSGN
- LOCKMGR (see note)
- MERGE (see note)
- OPERMSG (see note)
- OUTTRAP
- PAUSEMSG (see note)
- REXXIPT
- REXXMSG
- SETLANG
- SLEEP
- SORTSTEM (see note)

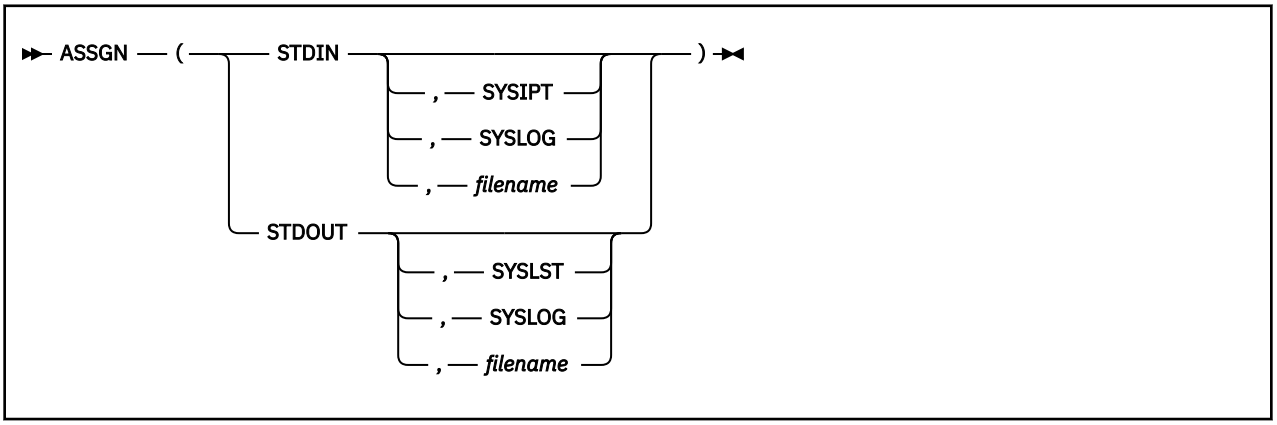
- STORAGE.
- SYSVAR

Note: These are functions packaged with REXX Console Automation. Some more functions which allow a REXX program to work with the REXX console are described in the section "Console-related REXX Functions" on page 229.

This section describes external functions. For general information about the syntax of function calls, see "Syntax" on page 59.

Chapter 18, "Customizing Services," on page 381 describes customization and language processor environments in more detail.

ASSGN



ASSGN returns the name of the current input or output stream and, optionally, changes it. You can use ASSGN(STDIN) or ASSGN(STDOUT) to return the name of the current input or output stream, respectively. If you specify one of the optional items, ASSGN returns the name of the current stream and changes the current stream to the value you specified.

If you specify *filename*, this is the name of the input or output file. The *filename* must be 1 to 8 characters.

Note:

1. Using SYSLST with STDIN or using SYSIPT with STDOUT results in REXX error 40, `Invalid call to routine`.
2. You must provide your own I/O replaceable routine unless you use one of the following file names:
   - SYSLOG
   - SYSIPT
   - SYSLST
   - SYS*xxx* (where xxx is numeric) If you specify a system file SYS*xxx* you might receive an error by the I/O replaceable routine ARXINOUT. See "Input/Output Routine" on page 446 for a list of supported file names.
   - Any other 7-character name.

   Otherwise, you receive an error. See "Input/Output Routine" on page 446 for information about supplying a replaceable routine.

   You need to open a SAM file (using EXECIO... (OPEN) before reading from or writing to the file. SYSIPT, SYSLST, and SAM files you have opened use the replaceable routine ARXINOUT.
3. SAM file names can be 1 to 7 characters.

PARSE EXTERNAL, PARSE PULL, PULL, SAY, TRACE, and error messages use the current input and output streams.

The INDD field in the module name table specifies the default input stream (SYSIPT), and the OUTDD field specifies the default output stream (SYSLST). Instead of using ASSGN to change the input or output stream, you can specify the INDD or OUTDD field in the in-storage parameter list during a call to ARXINIT. See "Module Name Table" on page 398 for a description of the module name table.

Examples:

```
/************************* REXX *********************************/
/* This REXX program gets a word from the input stream and sends   */
/* it to the output stream.                                        */
/*******************************************************************/

oldin = ASSGN('STDIN','SYSLOG')
oldout = ASSGN('STDOUT','SYSLOG')

say 'Enter the word.'
   PULL word                     /* Get the word. */
   SAY word

CALL ASSGN 'STDIN',oldin
CALL ASSGN 'STDOUT',oldout
EXIT
```

LOCKMGR

The LOCKMGR function allows to serialize REXX programs. See the detailed description on "LOCKMGR" on page 235.

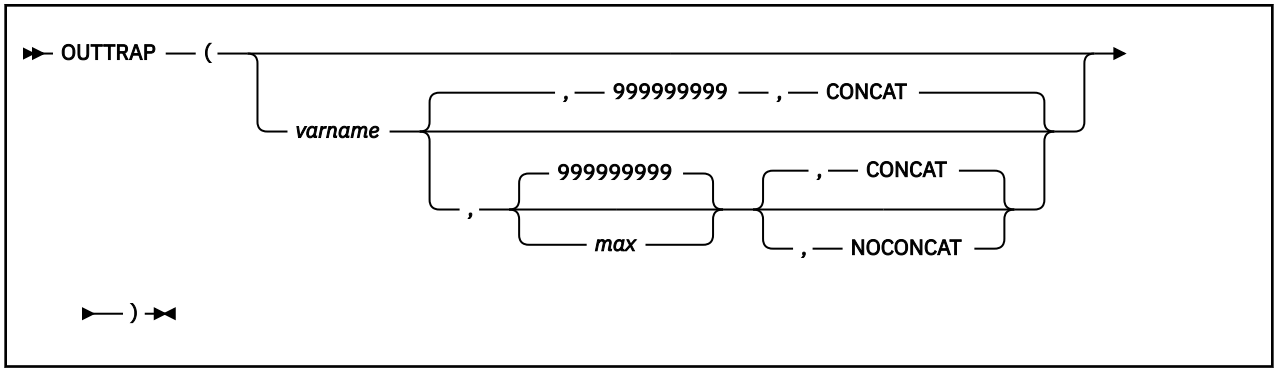
MERGE

The MERGE function creates a new library member using a given skeleton and input variables. See the detailed description on "MERGE" on page 236.

OPERMSG

The OPERMSG function adds or removes an operator communication exit. See the detailed description on "OPERMSG" on page 237.

OUTTRAP



OUTTRAP returns

- OFF (if it has not been previously used)
- the previously used *varname*

if used with arguments the following is trapped in the specified *varname*:

- user data provided by ARXOUT. This is only possible from a user program invoked by LINK or LINKPGM.
- job completion information retrieved by QUERYMSG.
- command output and error information from JCL.
- SYSLST output for LIBR and IDCAMS
- error information from PUTQE (page "PUTQE" on page 186). and GETQE (page "GETQE" on page 182)
- command output from VSE/POWER commands (CTL requests) routed back through the VSE/POWER spool-access services interface, or error information if the command fails.

(See VSE/POWER Application Programming, for a list of POWER commands you can send through a CTL service request. See VSE/POWER Administration and Operation, for the syntax of these commands.)

varname
is the stem of a compound variable (a stem must end with a period). It has no default value (trapping is not in effect until activated).

max
is the maximum number of lines to store in the compound variables. You can specify a number, an asterisk in quotation marks ('*'), or a blank. If you specify '*' or a blank, all the output is stored. The default is 999999999. Once the maximum number of lines are stored, subsequent lines are not stored in compound variables.

CONCAT
specifies storing trapped lines from successive commands in consecutive order until the maximum number of lines is reached. For example, if the first command has three lines of output and the second command has two lines of output, lines are stored in *varname*.1 through *varname.5*, respectively. CONCAT is the default.

NOCONCAT
specifies overwriting stored lines from successive commands. For example, if the first command has three lines of output, they are stored in *varname.1* through *varname.3*. Storing two lines of output from the second command overwrites the lines from the first command in *varname.1* and *varname.2*. (*Varname.3* would no longer contain the third line of the first command's output.) Before OUTTRAP stores output, *varname* is dropped (as if a REXX DROP instruction specifying the name of the stem had been used).

All unused variables have the value of their own names in uppercase. *Varname.0* contains the number of lines that have been stored. For example, if you specify `cmdout.` as the *varname*, the number of lines stored is in `cmdout.0`.

A program written in REXX cannot turn trapping off. Once trapping is turned on, it remains in effect until the program is done running. If a second call to a subsequent program is made, trapping is not in

effect unless the second program turns trapping on. When the second program ends, the trapping for that program ends and trapping for the first program is again in effect. (The REXX variables that trapping affects are in the program that is currently running.)

Additional Variables That OUTTRAP Sets

In addition to the variables that store the lines of output, OUTTRAP stores information in the following variables:

***varname*.0**
contains the total number of lines stored. The number in this variable cannot be larger than *varname*.MAX or *varname*.TRAPPED.

***varname*.MAX**
contains the maximum number of output lines that the user specified or the default. See example "4" on page 96.

***varname*.TRAPPED**
contains the total number of lines of command output. The number in this variable can be larger than *varname*.0 or *varname*.MAX.

***varname*.CON**
contains either CONCAT or NOCONCAT.

Examples:

The following are some examples of using OUTTRAP.

Note: You should use quotation marks around the string you specify for *varname* and around the keywords CONCAT and NOCONCAT.

1. To determine if trapping is in effect:

```
y = OUTTRAP()
SAY y           /* Produces the variable name being used to      */
/* store output or "OFF" if trapping is off      */
```

2. To suppress all command output:

```
y = OUTTRAP('output.',0)
```

   Note: This form of OUTTRAP is best for suppressing command output.

3. To store output from commands in consecutive order, using the stem `output.`, you can use one of the following:

```
y = OUTTRAP('output.','*','CONCAT')

y = OUTTRAP('output.')

y = OUTTRAP('output.',,'CONCAT')
```

4. This example contrasts CONCAT and NOCONCAT. Suppose you use the following to store output lines from two commands:

```
y = OUTTRAP('ABC.',4,'CONCAT')
```

   Command 1 has three lines of output.

```
ABC.0           -->  3       /* total lines stored    */
ABC.1           -->  Command 1 output line 1
ABC.2           -->  Command 1 output line 2
ABC.3           -->  Command 1 output line 3
ABC.4           -->  ABC.4   /* uninitialized variable */
ABC.MAX         -->  4
ABC.TRAPPED     -->  3       /* total output lines     */
ABC.CON         -->  CONCAT
```

Command 2 has two lines of output. They are stored in variables starting after the three lines already stored.

```
ABC.0           -->  4       /* total lines stored     */
ABC.1           -->  Command 1 output line 1
ABC.2           -->  Command 1 output line 2
ABC.3           -->  Command 1 output line 3
ABC.4           -->  Command 2 output line 1
ABC.MAX         -->  4
ABC.TRAPPED     -->  5       /* total lines output     */
ABC.CON         -->  CONCAT
```

(The second line from Command 2 is not stored because *max* is 4.)

However, if you use:

```
y = OUTTRAP('ABC.',4,'NOCONCAT')
```

to store the same two commands, this produces different results:

Results after Command 1 are the same (except for ABC.CON):

```
ABC.0           -->  3       /* total lines stored      */
ABC.1           -->  Command 1 output line 1
ABC.2           -->  Command 1 output line 2
ABC.3           -->  Command 1 output line 3
ABC.4           -->  ABC.4   /* uninitialized variable */
ABC.MAX         -->  4
ABC.TRAPPED     -->  3       /* total lines output     */
ABC.CON         -->  NOCONCAT
```

However, output lines from Command 2 overwrite lines from Command 1.

```
ABC.0           -->  2       /* total lines stored     */
ABC.1           -->  Command 2 output line 1
ABC.2           -->  Command 2 output line 2
ABC.3           -->  ABC.3   /* becomes uninitialized  */
ABC.4           -->  ABC.4
ABC.MAX         -->  4
ABC.TRAPPED     -->  2       /* total lines output     */
ABC.CON         -->  NOCONCAT
```

5. The following example uses OUTTRAP to capture error information from PUTQE:

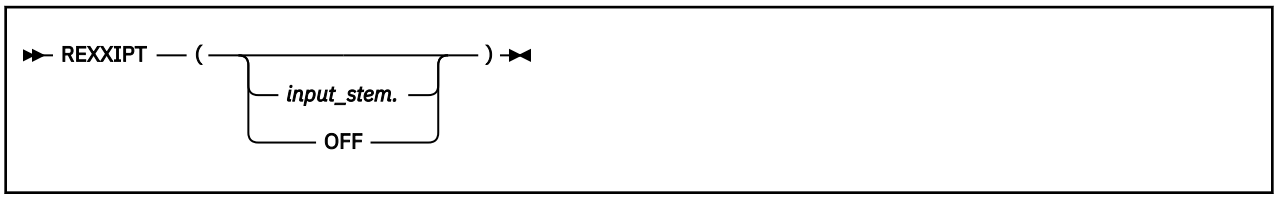
```
y = OUTTRAP('mystem.')
ADDRESS POWER "PUTQE RDR MEMBER member1 WAIT 3 CLASS 0"
```

If Class 0 is busy, so that the three-second interval elapses before the job can be put on the RDR queue, OUTTRAP stores error information in the compound variables whose names begin with the stem `mystem.`.

PAUSEMSG

The PAUSEMSG function issues a console message and waits for an operator reply. See the detailed description on “PAUSEMSG” on page 238.

REXXIPT



REXXIPT lets a program (called with ADDRESS JCL, ADDRESS LINK or ADDRESS LINKPGM) read data stored in compound variables as if it were SYSIPT data. It returns a previously defined input stem or 'OFF'.

REXXIPT cannot be used in a REXX program running in a subtask. If a second call to a subsequent REXX program is made, REXXIPT is off unless a stem is assigned to REXXIPT. When the second program ends, REXXIPT data is deleted and REXXIPT for the first program is on again.

The *input_stem* is the name of a stem (it must end with a period). It is used as the SYSIPT input stream for the specified host command environment. OFF specifies that no stem contains SYSIPT data.

To use the REXXIPT function:

1. Store the lines of data into compound variables.
2. Store the number of lines in input_stem *stem.0*.
3. Call the REXXIPT function.
4. Use the ADDRESS instruction to call the program.

In the following example, the ADDRESS instruction specifies the LINK environment and calls the program MYPHASE:

```
line.1="Now is the time"
line.2="for all good men"
line.3="to come to the aid of their country."
line.0=3   /* total number of lines of data */
oldstem = REXXIPT(line.)
ADDRESS LINK "MYPHASE"
```

The REXXIPT function call specifies name of the stem. In this example, `line.` is the name of the stem. To use the SYSIPT information provided by a stem, the REXXIPT function call must precede an ADDRESS instruction that loads and calls another program. You can use REXXIPT for the following environments:

- ADDRESS JCL
- ADDRESS LINK
- ADDRESS LINKPGM.

When MYPHASE reads a record from SYSIPT, it reads the contents of the compound variables in order. That is, it reads `line.1`, then `line.2`, and finally `line.3`.

The called program uses the VSE/ESA OPEN, GET, and CLOSE macros using a DTFDI-eqivalent from SYSIPT to read the data. A record containing fewer than 128 bytes is padded with blanks. A record containing more than 128 bytes is truncated. See z/VSE System Macro Reference for detailed information.

Reading the last record acts as the end of file condition. The *input_stem*.0 contains the total number of records. Reading a record whose number is one more than the contents of *input_stem*.0 indicates the end of data.

If you call a program a second time and it reads the records again, reading starts at the first record. Each time you start reading SYSIPT data you start at the first record again.

Note:

1. To have access to SYSIPT data, you need to use the JCL card `// EXEC REXX=` to call the program that contains the REXXIPT function call. (Otherwise, you receive error 40.)
2. The called program uses the OPEN, GET, and CLOSE macros using DTFDI from SYSIPT to read the data.

3. The *input_stem*.0 contains the total number of records.
4. Supported command environments can use REXXIPT from the main task. The REXX program must be called by the JCL statement // EXEC REXX.

REXXMSG

This function is intended for the general user. REXXMSG specifies the output destination where REXX/VSE messages are routed to. This destination is valid for all REXX programs running under the same language program environment. REXXMSG also enables the complete supression of REXX/VSE messages. REXXMSG sets the NOMSGWTO and NOMSGIO flags. These two flags control where REXX error messages are routed.

symbol can be one of the following:

ON
switches all REXX messages on. This is equal to NOMSGWTO=OFF and NOMSGIO=OFF.

STDOUT
REXX error messages are written to the standard output device STDOUT. The messages are surpressed if the current output is SYSLOG. This is equal to NOMSGWTO=ON and NOMSGIO=OFF.

SYSLOG
REXX error messages cannot be written to the standard output device STDOUT. Messages are written to SYSLOG. This is equal to NOMSGWTO=OFF and NOMSGIO=ON.

OFF
all REXX error messages are supressed. This is equal to NOMSGWTO=ON and NOMSGIO=ON.

REXXMSG returns the previous symbol set by REXXMSG. Here is an example

```
previous = REXXMSG('ON')    /* -> returns 'OFF' and sets messages on */
result   = REXXMSG(previous)/* -> returns 'ON'  and sets msg off        */
```

REXXMSG() just returns the current REXX error message destination without setting anything. REXXMSG() is set to ON as shipped by IBM. You may, however, have customized your installation to different settings.

Here is an example

```
 previous = REXXMSG('STDOUT') /* -> returns 'OFF' and sets STDOUT */
 current  = REXXMSG()         /* -> returns 'STDOUT'              */
```

Return Codes: Any invalid input results in a return code of 40.

Overruling REXXMSG

The REXX administrator can overrule the REXXMSG function and can suppress messages by setting NOPMSGS=ON and ALTMSGS=OFF in the ARXPARMS parameters module. You may now specify REXXMSG('ON'), the function is processed, REXXMSG() returns 'ON' but no messages are written.

SETLANG

SETLANG returns a three-character code that indicates the language in which REXX messages are currently being displayed. Table 1 on page 100 shows the language codes and the corresponding languages for each code.

You can optionally specify one of the language codes as an argument on the function. This sets the language in which REXX messages are displayed. SETLANG returns the code of the language in which REXX messages are currently displayed and changes the language in which subsequent messages will be displayed.

Table 1. Language Codes for SETLANG Function

Language Code	Language
ENP	US English - all uppercase
ENU	US English - mixed case (upper and lowercase) (This is the default.)

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)                */
```

After a program uses SETLANG to set a specific language, any REXX message the system issues is displayed in that language. If the program calls another program (either as a function or subroutine or using the EXEC command), any REXX messages are displayed in the language you specified on the SETLANG function. The language you specified on SETLANG is the language for displaying REXX messages until the program processes another call to SETLANG or the environment in which the program 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 page “LANGUAGE ” on page 392). 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.

SLEEP

Use SLEEP to wait for a number of seconds. *n* specifies the number of seconds a REXX program is requested to wait. After this time has elapsed, the REXX program continues processing. The highest allowed value is 55924. Any invalid input results in return code 40 and message ARX0040I.

Examples: The result of the SLEEP function is zero.

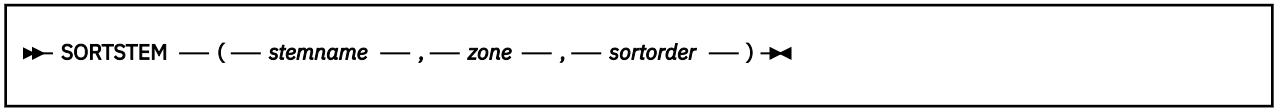
```
fc = SLEEP(1)
```

assigns the variable *fc* the value zero.

```
CALL SLEEP 1
```

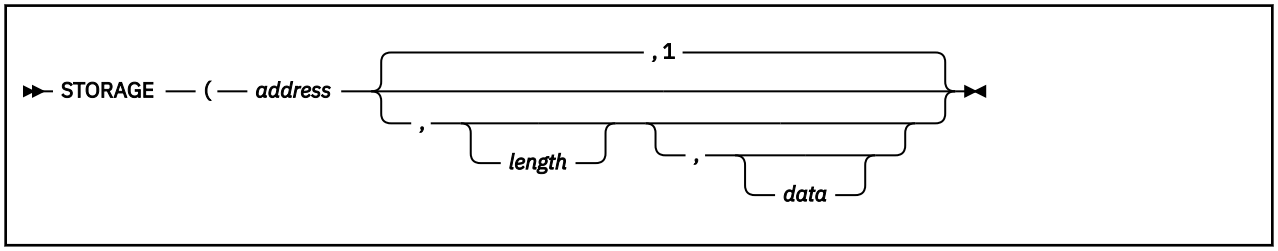
assigns the variable *result* the value zero.

SORTSTEM



The SORTSTEM function allows to sort contents of a stem variable. See the detailed description on "SORTSTEM" on page 240.

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.

Optionally, you can specify *length*, the decimal number of bytes to be retrieved from *address*. The default *length* is 1 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 does not affect how much storage is overwritten; the entire *data* is written.

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.

Note: Virtual storage addresses can be fetch protected or update protected, or they may not be valid addresses to VSE/ESA. An abend results if STORAGE references a nonexistent address or tries to update nonexistent storage, retrieve the contents of fetch-protected storage, or update store-protected storage.

The STORAGE function returns a null string if any part of the request fails. Because the STORAGE function can 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

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 REXX/VSE, use the following STORAGE function:

   ```
   storrep = STORAGE(0035D41F,,'REXX/VSE')
   ```

   This example first returns 1 byte of information found at address 0035D41F and then replaces the data beginning at address 0035D41F with the characters REXX/VSE.

   Note: Information is retrieved before it is replaced.

SYSVAR

SYSVAR returns system information about VSE/ESA. This information is stored in a REXX variable. The information returned depends on the *arg_name* specified on the function call. Any invalid input results in return code 40 and message ARX0040I. *arg_name* can be the following:

SYSMRC
stores the highest return code from VSE JCL in the variable SYSMRC. The return code may be up to 4 characters long.

SYSJOBNAME
the variable SYSJOBNAME returns the VSE JCL jobname (// JOB *jobname*). *jobname* may be from 1 to 8 characters long.

SYSJCLPROC
returns the JCL procedure name if the REXX program is invoked from a nested JCL procedure. Otherwise it will return a null string.

SYSLIBRCODE
returns the Librarian return and reason code of an EXECIO command for Libr members. It is a string consisting of two words. Each word consists of four digits. The first word shows the return code, the second word shows the reason code, e.g. '0016 0067'.

SYSPOWJNM
the variable SYSPOWJNM stores the VSE/POWER jobname (* $$ JOB JNM=*jobname*). *jobname* may be from 1 to 8 characters long. This variable may be only used if the VSE/POWER partition control block is available.

SYSPOWJNUM
the variable SYSPOWJNUM stores the jobnumber of the VSE/POWER job calling the REXX program. This variable may be only used if the VSE/POWER partition control block is available.

SYSPOWJCLS
the variable SYSPOWJCLS stores the jobclass of the VSE/POWER job calling the REXX program. This variable may be only used if the VSE/POWER partition control block is available.

SYSPID
the variable SYSPID returns the partition ID. It is 2 bytes long.

SYSVERSION
the variable SYSVERSION returns the VSE/ESA supervisor version (3 digits).

SYSERRCODES
relates to the VSE console environment: this variable contains the return and reason codes (see “Return and Reason Codes” on page 267) of the VSE system macro (such as MGCRE, MCSOPER, or WTO) which is used to issue a VSE console command. An example is shown in section “SYSVAR” on page 242.

SYSCPUID
stores the CPUID of your VSE system in the variable SYSCPUID.

Examples: Return the VSE JCL jobname: if a REXX exec runs under the JCL job with jobcard "// JOB REXXJOB".

```
fc=SYSVAR('SYSJOBNAME')
SAY SYSJOBNAME    /*  Displays REXXJOB */
```

Return the VSE/ESA supervisor version:

```
fc=SYSVAR('SYSVERSION')
SAY SYSVERSION    /*  Displays 610  */
```

Return the Librarian return and reason code:

```
" EXECIO * PRD2.PROD.myfile.Z (FINIS "
IF RC=20  THEN
   DO
    CALL SYSVAR|'SYSLIBRCODE'|
    IF (WORD(syslibrcode,1)='0016' &
            (syslibrcode,2)='0067'
    THEN SAY EXECIO failed as member is locking
```

Return Codes: Table 2 on page 103 shows the return codes for the SYSVAR function.

Table 2. Return Codes for the SYSVAR function

Return Code	Description
0	Processing was successful.
4	Processing was not successful. System information could not be retrieved.
8	Processing was not successful. System information could not be stored into a REXX variable.
40	Any invalid input was entered.

Chapter 5. Parsing

Parsing Rules

The parsing instructions are ARG, PARSE, and PULL (see “ARG” on page 29, “PARSE” on page 44, and “PULL” on page 48).

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 107 for details.)

Positional patterns
Indicate the character positions at which to split the source string. (See “Templates Containing Positional (Numeric) Patterns” on page 108 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 111 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. (A summary of all the parsing instructions is on page “Parsing Instructions Summary” on page 112.)

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 111 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. (There is a special case (on page "Combining String and Positional Patterns: A Special Case" on page 114) 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 111 for details on *variable positional patterns*.)

The number specifies the absolute character position at which to split the source string.

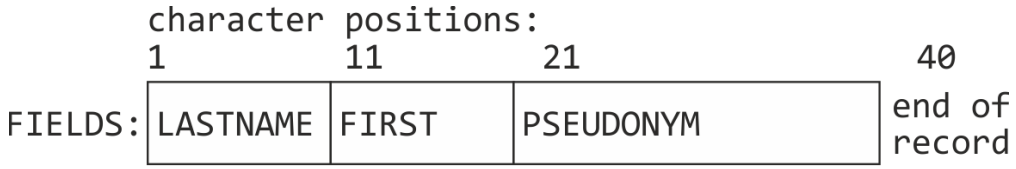
Here is a template with absolute positional patterns:

```
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
```

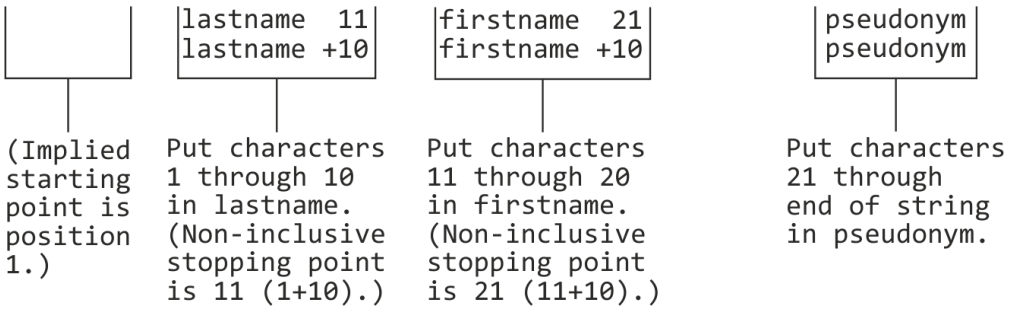
A *relative positional pattern* is a number with a plus (+) or minus (-) sign preceding it. (It can also be a variable within parentheses, with a plus (+) or minus (-) sign preceding the left parenthesis; for details see "Parsing with Variable Patterns" on page 111.)

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 (on page "Combining String and Positional Patterns: A Special Case" on page 114) 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.



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"                           */
```

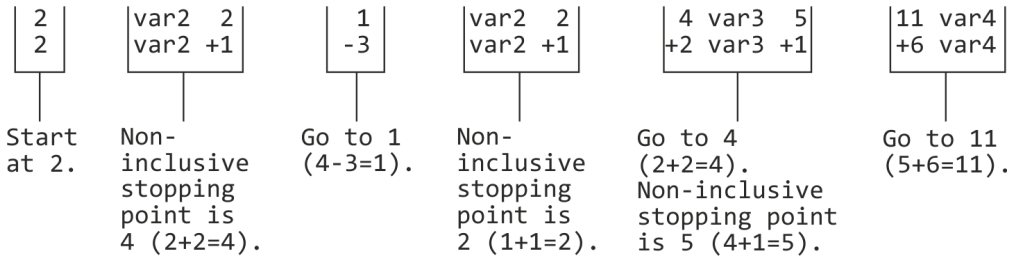
The absolute positional pattern 1 backs up to the first character in the source string.

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.



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 105.)

```
/* 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 105.)

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 the current input stream. ASSGN(STDIN) returns the name of the current input stream.
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.) input.
PARSE SOURCE	REXX/VSE-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	REXX/VSE-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 114.

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:       "VSE"    */
```

PARSE VALUE example is on “Simple Templates for Parsing into Words” on page 105.

PARSE VAR examples are throughout the chapter, starting on “Simple Templates for Parsing into Words” on page 105.

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

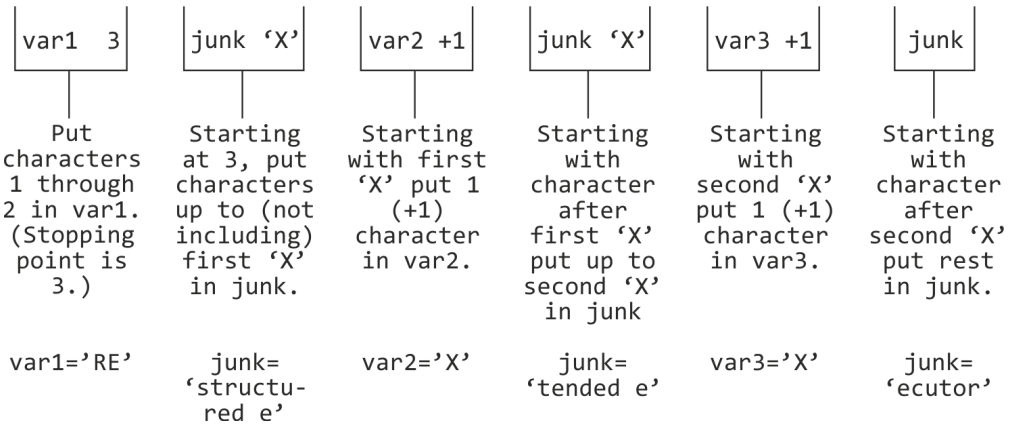
There is a special case in which absolute and relative positional patterns do not work identically. We have shown how parsing with a template containing a string pattern skips over the data in the source string that matches the pattern (see “Templates Containing String Patterns” on page 107). But a template 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:



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 482 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.

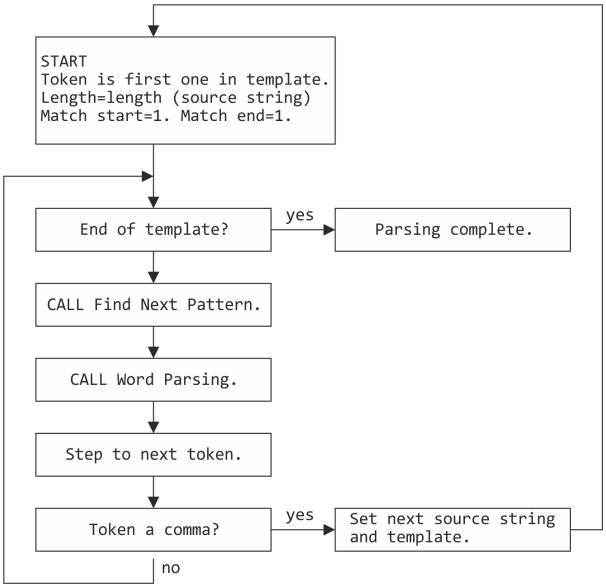


Figure 3. Conceptual Overview of Parsing

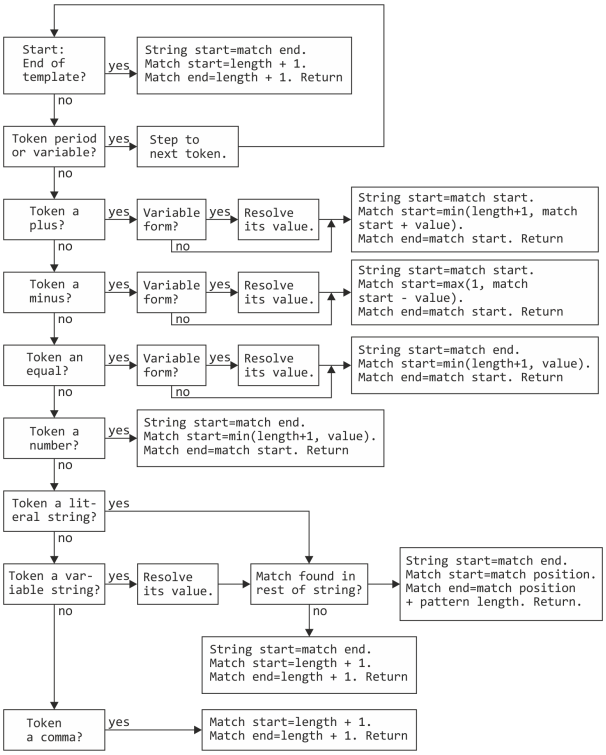


Figure 4. Conceptual View of Finding Next Pattern

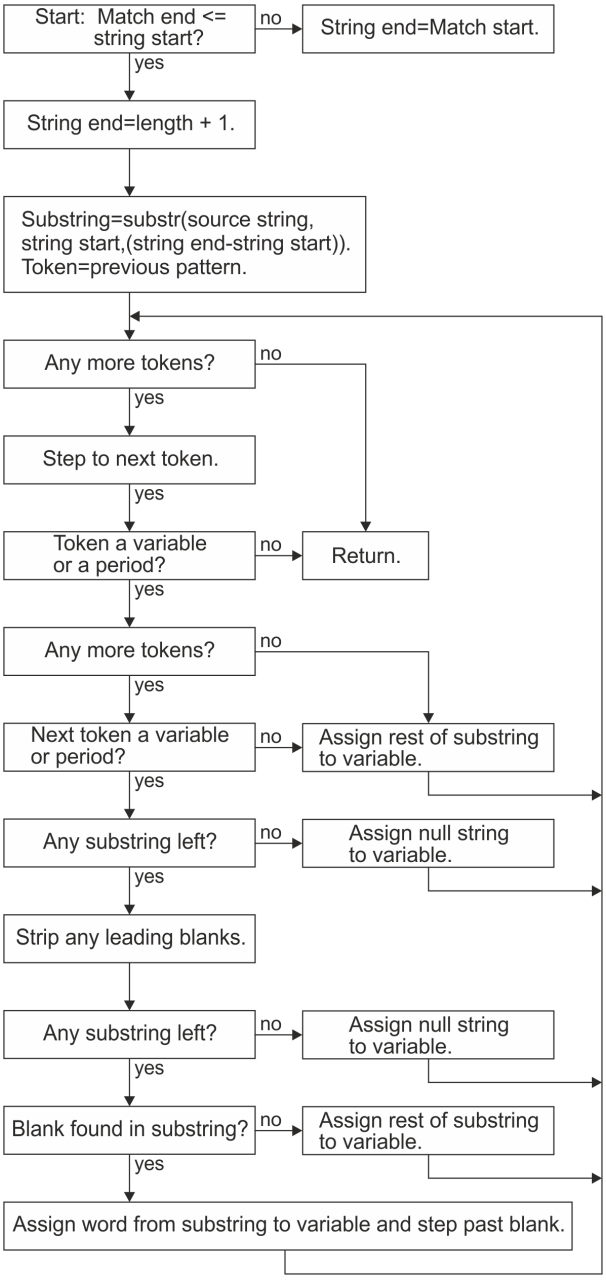


Figure 5. Conceptual View of Word Parsing

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 81), 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 0.000000000333333333 */
```

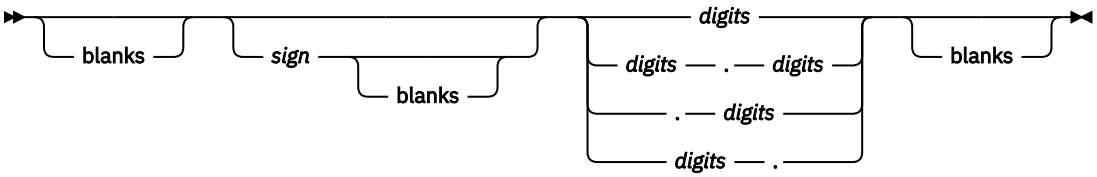
Definition

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 124 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

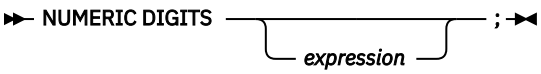
sign
is either + or -

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 74) 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
```

Note: 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.

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 126.) 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 123 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 15. 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:.

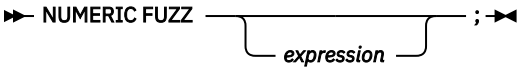
```
A ? Z
```

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

```
0.000000000000000000001
```

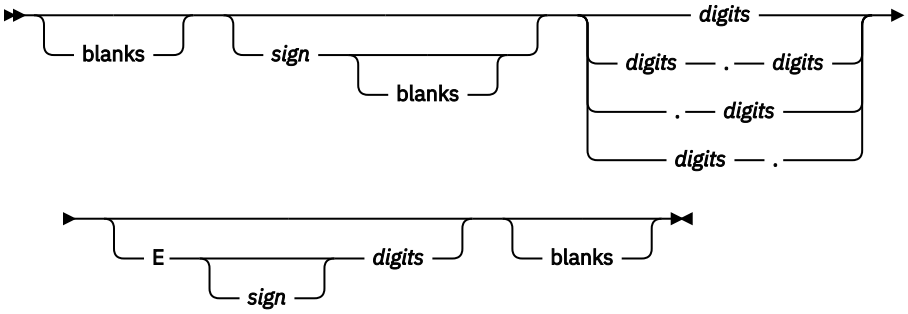
For both large and small numbers some form of exponential notation is useful, both to make long numbers more readable, and to make execution possible in extreme cases. In addition, exponential notation is used whenever the "simple" form would give misleading information.

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 raised to the 123 power) and `1E342` (1 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.

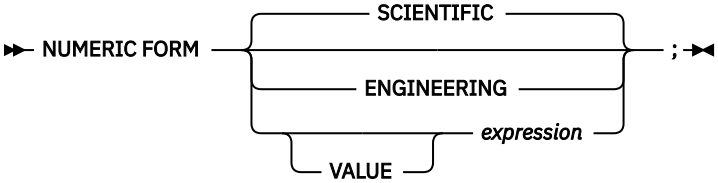
Here are some examples:

```
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 page “FORMAT” on page 74).

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 the largest number you can use is 999999999.

- 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, you can use exponents in the range -999999999 through 999999999.

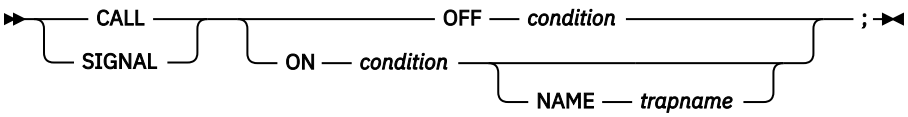
  Because this allows for (very) large exponents, overflow or underflow is treated as a syntax error.

- 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 30 and "SIGNAL" on page 51).



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.

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: See "The VSE Host Command Environment" on page 25 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.

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 immediate command HI (Halt Interpretation) raises a halt condition. The RXHLT exit ("REXX Exit Data Areas and Parameters" on page 473) also raises a halt condition. See "Interrupting Program Processing" on page 321.

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 "run-time" 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/VSE Messages and Codes) 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.

Question

At the 9/93 ARB, this wording changed from 'occurred.' to 'occurred, and the caller usually receives an indication of failure.' Should this change print for VSE?

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 page “SIGNAL” on page 51).

  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 page "CALL" on page 30) 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 (page "CALL" on page 30) 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 page "CONDITION" on page 66).

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 on 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 page "RETURN" on page 49.)

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 routine (such as, a REXX program) 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 "The VSE Host Command Environment" on page 25.

For SIGNAL ON SYNTAX, RC is set to the syntax error number.

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

The REXX language consists of keyword instructions and built-in functions that you use in a REXX program. The keyword instructions and built-in functions are described in Chapter 3, “Keyword Instructions,” on page 27 and Chapter 4, “Functions,” on page 59, respectively.

You can also use external functions and REXX/VSE commands in a REXX program. The functions are described in “External Functions” on page 92. The REXX/VSE commands provide additional services that let you:

- Control I/O processing
- Perform data stack requests
- Change characteristics that control how a REXX program runs
- Check for the existence of a specific host command environment.

See Chapter 10, “REXX/VSE Commands,” on page 143 for details.

See “Writing Programs” on page 137 for information about services you can use in programs.

REXX/VSE is a partial implementation of Level 2 SAA REXX on the VSE/ESA system. By using the keyword instructions and functions that are defined for the SAA REXX language, you can write REXX programs that can run in any of the supported SAA environments. See the *SAA Common Programming Interface REXX Level 2 Reference* for more information.

Additional REXX Support

REXX/VSE also provides:

programming services
You can use these to interface with REXX and the language processor.

customizing services
These let you customize REXX processing and accessing and using system services.

Programming Services

The REXX/VSE programming services are:

ARXEXCOM – Variable Pool Access
ARXEXCOM lets you access and manipulate the current generation of REXX variables. Commands and programs can call ARXEXCOM to inspect, set, and drop REXX variables. See page “Variable Pool – ARXEXCOM” on page 352 for a description.

ARXSUBCM – Maintain Host Command Environments
ARXSUBCM is a programming interface to the *host command environment table*. This table contains the names of the environments and routines that handle the processing of host commands. You can use ARXSUBCM to add, change, delete, and query entries in the table. See page “Maintain Entries in the Host Command Environment Table – ARXSUBCM” on page 357 for a description.

ARXIC – Trace and Execution Control
ARXIC, the trace and execution control routine, is an interface to the immediate commands HI, HT, RT, TQ, TS, and TE. A program can call ARXIC to use these commands to affect the processing and tracing of REXX programs. See page “Trace and Execution Control Routine – ARXIC” on page 361 for a description.

ARXRLT – Get Result
ARXRLT gets the result from a REXX program. that the ARXEXEC routine called. ARXRLT also allows a non-REXX program to get an EVALBLOK to return a result to REXX. See page “Get Result Routine – ARXRLT” on page 363 for a description.

ARXJCL and ARXEXEC – Exec Processing
The ARXJCL and ARXEXEC routines call a REXX program. These routines are programming interfaces to the language processor. You can run a program in batch by specifying ARXJCL as the program name on the JCL EXEC statement. You can call either ARXJCL or ARXEXEC from an application program to call a REXX program. See page "Calling REXX" on page 328 for descriptions.

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. Or you can write them in any programming language that supports the system-dependent interfaces that the language processor uses to call a function or subroutine.

You can also group frequently used external functions and subroutines into a *package*. This allows quick access to the functions and subroutines. To include an external function or subroutine in a function package, the function or subroutine must be link-edited into a phase. See page "External Functions and Subroutines and Function Packages" on page 344 for a description of the system-dependent interfaces for writing external functions and subroutines and how to define function packages.

ARXOUT – OUTTRAP Interface Routine
ARXOUT lets programs write a character string to the REXX stem specified by the OUTTRAP external function. Programs using this interface must have been invoked by the ADDRESS LINK or ADDRESS LINKPGM host command environment. See page "OUTTRAP Interface Routine – ARXOUT" on page 378 for a description.

ARXSAY – SAY Instruction Routine
ARXSAY lets you write a character string to the same output stream as the REXX SAY instruction. See page "SAY Instruction Routine – ARXSAY" on page 368 for a description.

ARXHLT – Halt Condition Routine
ARXHLT queries or resets the halt condition. See page "Halt Condition Routine – ARXHLT" on page 370 for a description.

ARXTXT – Text Retrieval Routine
ARXTXT retrieves data from the message repository. This is the same text that the language processor uses for the ERRORTEXT built-in function and for certain options of the DATE built-in function. For example, a program can use ARXTXT to retrieve the name of a month or the text of a syntax error message. See page "Text Retrieval Routine – ARXTXT" on page 372 for a description.

ARXLIN – LINESIZE Function Routine
ARXLIN lets you retrieve the same value that the LINESIZE built-in function returns. See page "LINESIZE Function Routine – ARXLIN" on page 376 for a description.

Customizing Services

There are services you can use to customize REXX processing. Many services let you change how a program 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
Various routines and services allow you to customize the environment in which the language processor processes a REXX program. This environment is known as the *language processor environment* and defines various characteristics relating to program processing and how to access and use system services. There are default environment characteristics that you can change and also a routine you can use to define your own environment.

Replaceable Routines
When a REXX program runs, various system services are used, such as services for loading and freeing a program, I/O, obtaining and freeing storage, and data stack requests. *Replaceable routines* handle these types of system services. (They are called *replaceable routines* because you can provide your own routine that either replaces the REXX/VSE routine or that performs pre-processing and then calls the REXX/VSE routine.)

Exit Routines
You can provide exit routines to customize various aspects of REXX processing.

The chapters Chapter 18, “Customizing Services,” on page 381 through “Installing the Exec Processing, Exec Initialization, and Exec Termination” on page 477 describe the different ways in which you can customize REXX processing.

Writing Programs

You can use the following in a program:

- Assignment
- All keyword instructions that are described in Chapter 3, “Keyword Instructions,” on page 27
- All built-in functions that are described in Chapter 4, “Functions,” on page 59
- The external functions ASSGN, OUTTRAP, REXXIPT, REXXMSG, SETLANG, SLEEP, STORAGE, and SYSVAR. See “External Functions” on page 92 for more information.
- The following REXX/VSE commands:
  - DELSTACK - Deletes the most current data stack that was created with NEWSTACK.
  - DROPBUF - Drops (discards) a buffer that was previously created on the data stack with MAKEBUF.
  - EXEC - runs a REXX program in the active PROC chain. (See “Writing Programs” on page 137 for an example.)
  - EXECIO - Reads data from and writes data to files. You can use EXECIO to read data from and write data to the data stack or stem variables.
  - MAKEBUF - Creates a buffer on the data stack.
  - NEWSTACK - Creates a new data stack and effectively isolates the current data stack that the program is using.
  - QBUF - Queries how many buffers are currently on the active data stack.
  - QELEM - Queries how many elements are on the data stack above the most recently created buffer.
  - QSTACK - Queries the number of data stacks currently in existence.
  - SETUID - Lets you specify the user ID and password associated with a request through the VSE/POWER spool-access services interface.
  - SUBCOM - Determines whether a particular host command environment is available to process host commands.
  - TE (Trace End) - Ends tracing of the program.
  - TS (Trace Start) - Starts tracing of the program.

  See Chapter 10, “REXX/VSE Commands,” on page 143 for details on these commands.
- Instructions to call a program

  You can call a REXX program from another REXX program using the following instructions (the examples assume that the current host command environment is VSE):

```
"EXEC program_name p1 p2 …"
"EX program_name p1 p2 …"
"program_name p1 p2 …"  /* Implicit EXEC command */
```

- ADDRESS POWER commands:
  - GETQE – retrieves an entry from a POWER queue and stores the lines it retrieves.
  - PUTQE – places a job on a POWER queue.
  - QUERYMSG – returns job completion messages into the stem specified by OUTTRAP.

  - CTL service requests to POWER (sent through the VSE/POWER spool-access services interface). See "Commands to External Environments" on page 23 for more information.

- Instructions that load and call programs

  You can use the LINK and LINKPGM host command environments to load and call a phase from the active PHASE search chain. For example:

  ```
  ADDRESS LINK "PROGRAM p1 p2 ..."
  ```

  For more information, see Chapter 13, "Host Command Environments for Loading and Calling Programs," on page 205.

- JCL commands

  You can use the JCL host command environment to issue JCL commands via a REXX program. For example:

  ```
  ADDRESS JCL "jcl_command"
  ```

  For more information, see "The JCL Host Command Environment" on page 201.

- Console commands

  You can use the CONSOLE host command environment to issue console commands via a REXX program. For example:

  ```
  ADDRESS CONSOLE "console_command"
  ```

  For more information, see Chapter 14, "REXX/VSE Console Automation," on page 217.

- Programming services

  See Chapter 17, "Programming Services," on page 323 for descriptions of programming services such as ARXEXEC, ARXJCL, ARXEXCOM, and ARXIC.

Running a Program

You can call a REXX program directly by using the JCL EXEC command (see "Calling REXX Directly with the JCL EXEC Command" on page 329). Or you call a REXX program by using the ARXJCL or ARXEXEC routine. These routines are programming interfaces to the language processor. See "The ARXEXEC Routine" on page 334 and "The ARXJCL Routine" on page 331 for details about these programming interfaces and information about using ARXJCL to run a REXX program.

You can use ARXJCL to call a REXX program from a non-REXX program (for example, a PL/I program).

To call a REXX program from another REXX program, you can use the REXX/VSE EXEC command. Here are some examples using the ADDRESS command. The *environment* following the ADDRESS keyword is POWER. This specifies sending the expression within quotation marks to the POWER environment.

```
ADDRESS POWER "EXEC program_name p1 p2 …"
ADDRESS POWER "EX program_name p1 p2 …"
```

See "The VSE Host Command Environment" on page 25 for more information about environments for issuing host commands.

Communicating with a User Console

With the ECHO parameter in the VSE/POWER $$ JOB statement REXX can communicate with a user console. In the following example all messages REXX writes to SYSLOG are routed to a user console named REXX. Replies given on the user console are routed to the REXX exec.

```
* $$ JOB JNM=REXXJOB,...,ECHO=(ALL,REXX)
// JOB REXXJOB
```

```
// EXEC REXX=RXPGM
/&
* $$ EOJ
```

The demo program REXXTRY, which is described on “REXXTRY” on page 262, provides an interactive testing facility of REXX statements.

Chapter 9. Reserved Keywords, Special Variables, and Command Names

This topic describes reserved keywords, special variables, and reserved command names. Where there is no ambiguity, you can use keywords as symbols; the precise rules are given here.

REXX has three special variables: RC, RESULT, and SIGL. (The names of the special variables are not reserved.)

The names of REXX/VSE commands are reserved.

Reserved Keywords

The syntax of REXX implies that some symbols are reserved for the language processor's use in certain contexts.

Within particular instructions, some symbols may be reserved to separate the parts of the instruction. These symbols are called keywords. Examples of REXX keywords are the WHILE in a DO instruction, and the THEN (which ends a clause in this case) following an IF or WHEN clause.

Apart from these cases, the language processor checks only simple symbols that are the first token in a clause and that are not followed by an equal sign (=) or colon (:) to see if they are instruction keywords. You can use the symbols freely elsewhere in clauses without their being treated as keywords.

However, you are not recommended to use 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. You may want to enclose an entire host command in quotation marks. This ensures that the language processor processes the expression as a host command.

Special Variables

There are three special variables that the language processor can set automatically:

RC
is the return code from any executed host command (or subcommand). Following the SIGNAL events SYNTAX, ERROR, or 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 from input during debug mode do not change the value of RC.

The special variable RC can also be set to a -3 if the host command could not be found. See “The VSE Host Command Environment” on page 25 for information about issuing commands from a program.

The REXX/VSE 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. Chapter 10, “REXX/VSE Commands,” on page 143 describes the commands.

RESULT
is set by a RETURN instruction in a called subroutine, 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 call, or a trapped error condition could cause this.)

None of these variables has an initial value. You can change their values, just as with any other variable. You can access them using the variable pool access interface ARXEXCOM (page “Variable Pool –

ARXEXCOM" on page 352). 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 called and the source of the program, which is available using the PARSE SOURCE instruction. See page "PARSE SOURCE " on page 45 for details about the information PARSE SOURCE returns.

PARSE VERSION provides information about the version and date of the language processor code that is running. (See page "PARSE VERSION " on page 46.)

The TRACE built-in function returns the current trace setting. The ADDRESS built-in function returns the name of the host command environment.

Finally, you can obtain the current NUMERIC settings with the DIGITS, FORM, and FUZZ built-in functions.

Reserved Command Names

You can also use REXX/VSE commands in REXX programs. The names of these commands are reserved. It is recommended that you do not use these names for names of your REXX programs or phases. The REXX/VSE commands are in the next chapter.

Chapter 10. REXX/VSE Commands

REXX/VSE provides commands to perform different services, such as I/O and data stack requests. You can use the REXX/VSE commands in both the VSE and the POWER environment. “The VSE Host Command Environment” on page 25 and “The POWER Host Command Environment” on page 25 describe these environments.

The REXX/VSE commands perform services, such as:

- Performing data stack services (MAKEBUF, DROPBUF, QBUF, QELEM, NEWSTACK, DELSTACK, QSTACK)
- Changing characteristics that control tracing (immediate commands TE and TS)

  Note: See “Immediate Commands” on page 143 for details about use of immediate commands.
- Checking for the existence of a host command environment (SUBCOM).

Note: The names of the REXX/VSE commands are reserved. It is recommended that you do not use these names for names of your REXX programs or phases.

Immediate Commands

The immediate commands are:

- HI – Halt Interpretation
- HT – Halt Typing
- RT – Resume Typing
- TE – Trace End
- TQ – Trace Query.
- TS – Trace Start.

You can use HI, HT, RT, and TQ only by including them on a call from a non-REXX program to the programming interface ARXIC. You can use TE and TS by including them in a REXX program or specifying them on a call to ARXIC from a non-REXX program.

The operator can send a message to a particular partition. A partition that is running a REXX program ignores the message.

For information about the syntax of each immediate command, see the description of the command in this chapter.

DELSTACK

DELSTACK deletes the most recent data stack NEWSTACK has created and all elements on it. If a new data stack was not created, DELSTACK removes all the elements from the original data stack.

You can create a new data stack with NEWSTACK and delete that data stack with DELSTACK. Or your program can call an external function or subroutine that is written in REXX and includes 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, you can find out how many data stacks were created and delete all but the original data stack using NEWSTACK, QSTACK, and DELSTACK 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

DROPBUF removes the most recently created (with MAKEBUF) data stack buffer and all elements on the data stack in the buffer. If you specify *n*, DROPBUF removes a specific data stack buffer and all buffers created after it.

Operands:

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 because MAKEBUF does not create it. DROPBUF 0 removes all buffers that were created on the data stack with MAKEBUF and all elements that were put on the data stack. DROPBUF 0 effectively clears the data stack including the elements on buffer 0.

The following table shows how DROPBUF sets the REXX special variable RC.

Return Code	Meaning
0	DROPBUF was successful.
1	An incorrect 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.

Examples: A subroutine (sub2) in a REXX program issues MAKEBUF to create four buffers. Before the subroutine returns, it removes buffers two and above and all elements within the buffers.

```
/*  REXX program  */
      ⋮
     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
```

EXEC

EXEC runs a REXX program in the active PROC chain.

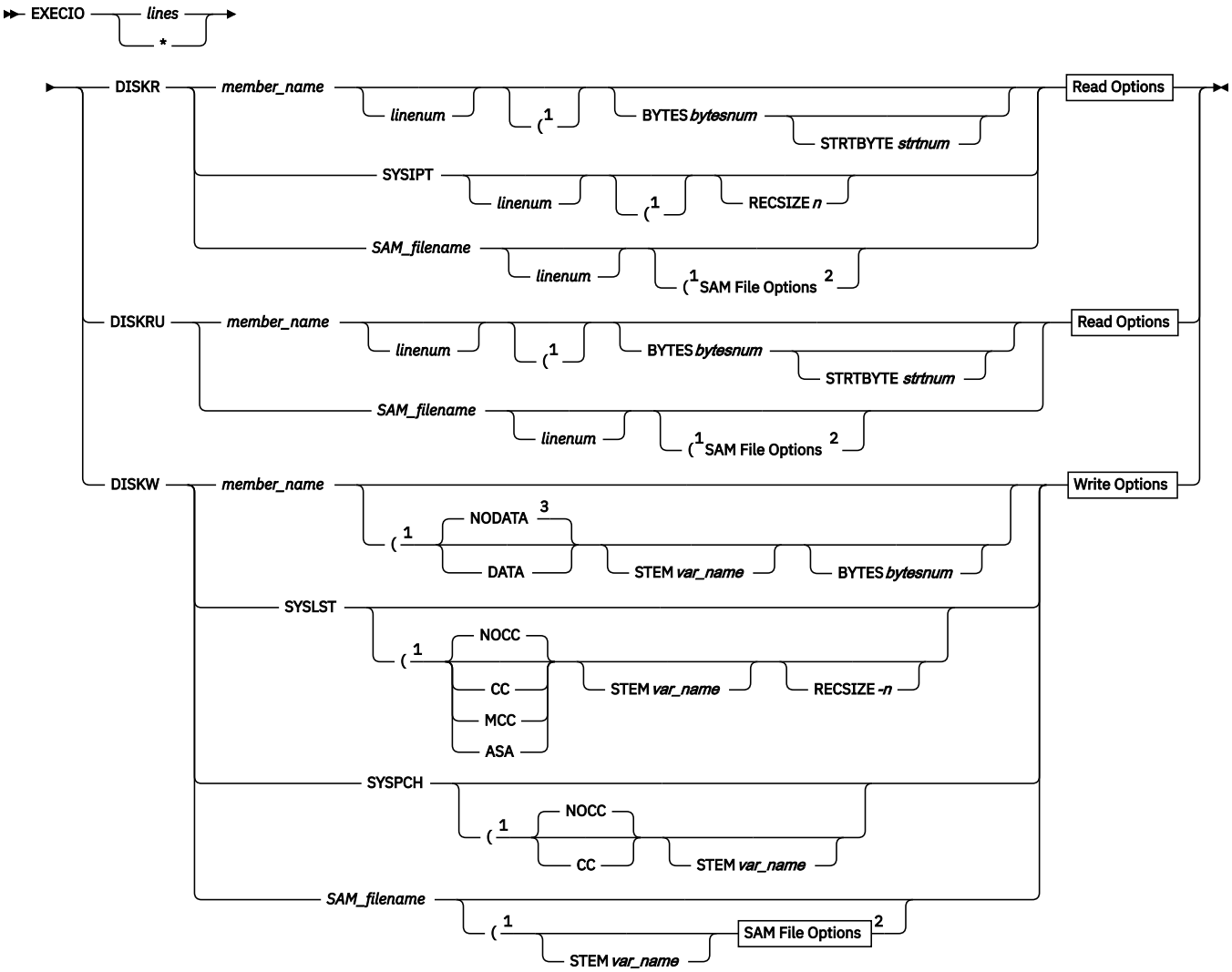
Operands:

pgm_name
is the name of the program. It is 8 characters or fewer.

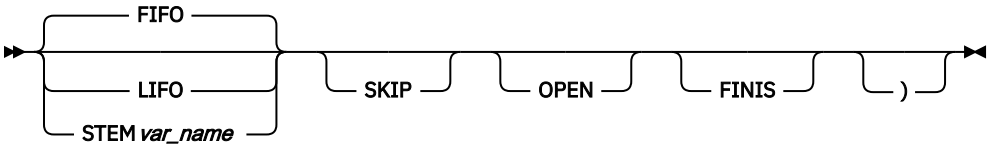
string
is an argument string. The language processor treats this as a single argument. The *string* is optional.

See “The VSE Host Command Environment” on page 25 for examples.

EXECIO

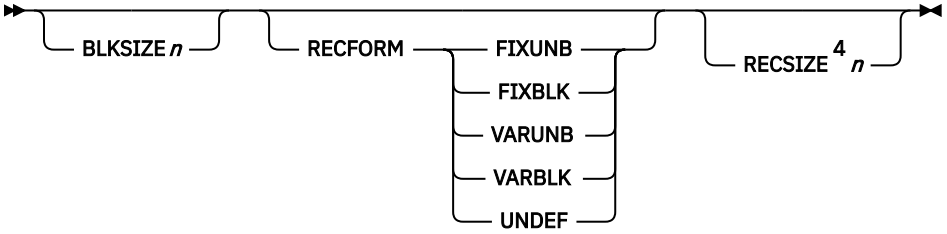


Read Options



Write Options

SAM File Options



Notes:

[1] You can enter the options between the parentheses in any order.

[2] SAM files require additional options for opening a file explicitly or implicitly.
[3] The default is NODATA for a new member. For a member that already exists, the default is its value from when it was created.
[4] RECSIZE is required with RECFORM FIXUNB or RECFORM FIXBLK; do not use it with other types of record formats.

EXECIO controls the input and output (I/O) of information to and from a file. Supported operations are DISKR, DISKW and DISKRU (read and update).

EXECIO can read or write data on the program stack or in REXX variables directly. You can use EXECIO for I/O tasks such as copying information to and from a file to add, delete, or update information. A program can read information from a file to the data stack for serialized processing or to a list of variables for random processing. A program can write information from the data stack or a list of variables to a file.

EXECIO operates on the following types of files:

- Sublibrary members of any type. The REXX program must specify the full name of the member on the EXECIO command. (The full name consists of a library name, sublibrary name, member name, and member type, for example: `mylib.mysublib.myfile.typea`.) An example of reading a sublibrary member is on "1" on page 154. Usually library members have a logical record format "fixed.". But some types, for example DUMP, PHASE, have a logical record format "string". In this case the member consists of 1 record only with arbitrary length.
- SYSIPT , SYSLST, and SYSPCH. These names are reserved words on the EXECIO command. You must specify DISKR (not DISKRU) with SYSIPT. Note that REXX/VSE reads SYSIPT data until encountering an end-of-file indicator, such as `/*`. See "Calling REXX Directly with the JCL EXEC Command" on page 329 for an example of input lines in SYSIPT. If a REXX program is invoked from a nested JCL procedure, EXECIO from SYSIPT cannot read from the current procedure.
- SAM files. Only SAM files on disk are supported.

  Before using EXECIO to perform I/O to or from a SAM file, you need to assign a name to the file. You do this by using DLBL to associate the file with a file name. Accessing SAM files requires additional options on the EXECIO command that are not needed for other files. See "Additional Options Required for SAM Files" on page 152 for details. See "3" on page 154 for an example.

Put quotation marks around any operands, such as DISKW, STEM, FINIS, or LIFO.

Operands:

lines
is the number of lines to be processed. This operand can be an integer or *, which indicates an arbitrary number. When the operand is * and EXECIO is reading from a file, input is read until EXECIO reaches the end of the file.

If you specify a value of 0, no I/O operations are performed unless you also specify OPEN or FINIS or both.

- If you specify OPEN and the file is closed, EXECIO opens the file but does not read or write any lines. If you specify OPEN and the file is open, EXECIO does not read or write any lines.

  In either case, if you are reading from a file and specify a nonzero value for *linenum*, EXECIO sets the current record number to the record number *linenum* indicates.

  Note: The current record number is the number of the next record EXECIO will read. By default, the current record number is set to the first record when a file is opened. However, if you specify OPEN and a nonzero value for *linenum*, EXECIO sets the current record number to the record number *linenum* indicates.
- If you specify FINIS and the file is open, EXECIO does not read or write any lines, but it closes the file. If you specify FINIS and the file is not already open, EXECIO does not open the file and then close it.
- If you specify both OPEN and FINIS, EXECIO processes the OPEN first and then the FINIS.

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 and the stack becomes empty, EXECIO continues with the current input stream. ASSGN(STDIN) returns the name of the current input stream. When end-of-file is reached, EXECIO ends.

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 has not been assigned a value).

DISKR
opens a file for input (if it is not already open) and reads the specified number of lines from the file and places them on the data stack.

If you specify the STEM operand, the lines are placed in a list of variables instead of on the data stack. While a file is open for input, you cannot write information back to the same file.

The file is not automatically closed unless:

- The task, under which the file was opened, ends
- The last language processor environment associated with the task, under which the file was opened, is terminated. (See Chapter 19, “Language Processor Environments,” on page 387 for information about language processor environments).

DISKRU
opens a file for update (if it is not already open) and reads the specified number of lines from the file and places them on the data stack.

If you specify the STEM operand, the lines are placed in a list of variables instead of on the data stack. While a file is open for update, the last record read can be changed and then written back to the file with a corresponding EXECIO DISKW command. Typically, you open a file for update when you want to change information in the file.

The file is not automatically closed unless:

- The task, under which the file was opened, ends
- The last language processor environment associated with the task, under which the file was opened, is terminated.

After you open a file for update (by issuing a DISKRU as the first operation against the file), you can use either DISKR or DISKRU to fetch subsequent records for update.

DISKW
opens a file for output (if it is not already open) and writes the specified number of lines to the file. The lines are from the data stack or, if you specify STEM *var_name*, from a list of variables.

You can use the DISKW operand to write information to a different file from the one opened for input, or to update, one line at a time, the same file opened for update.

When you write data to a library member with logical record format "string" and the number specified with option BYTES is smaller than the length of the string to be written, then the data is truncated and the return code is set to 1. If the BYTES number is greater than the available string length, only the available number of bytes is written and the return code is set to zero.

When a file is open for update, you can use DISKW to rewrite the last record read. The *lines* value must be 1 when doing an update. For *lines* values greater than 1, the user receives an error message and a return code of 20, and the program is ended. Once a line is written, the program cannot rewrite the line; attempting to do so causes an error.

When a file with logical record format "string" is open for update, you can use DISKW to rewrite the latest portion read. The new string may have a different length than the one being replaced. No padding or truncation of the new string takes place. After one portion of the record has been updated, the attempt to write another portion without a DISKRU operation in between causes an error.

The file is not automatically closed unless:

- The task, under which the file was opened, ends.

- The last language processor environment associated with the task, under which the file 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 the original line length.
2. You can read a DUMP or a PHASE either as a whole or broken up into portions using options BYTES and STRTBYTE. Writing or reading for update of a PHASE is not possible. The open will fail with return code 20 and error messages containing LIBRM OPEN feedback (code 236, incorrect phase handling). When you write a DUMP or other string-type members, use option BYTES. Otherwise, the library member is defined with fixed logical record format and only 80 bytes of the first record will be written. You also get a truncation return code of 1.

member_name
SYSIPT
SYSLST
SYSPCH
SAM_filename
The file name is a sublibrary member, SYSIPT, SYSLST, SYSPCH, or the name assigned to a SAM file. The name of a sublibrary member is in the format: library.sublibrary.member.filetype. For input or output to a SAM file, you must use DLBL to assign the file a name before using EXECIO. The DLBL statement can either refer to a SAM-file in VSAM-managed space, or directly to a disk. If you are processing a file directly on a disk, an *ASSGN SYS007,cuu* is necessary for DISKW and an *ASSGN SYS006,cuu* for DISKR and DISKRU.

linenum
is the line number in the file at which EXECIO is to begin reading. When a file is closed and reopened because of specifying a record number preceding the current record number, the file is open for:

- input, if you specify DISKR
- update, if you specify DISKRU.

When a file 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 file, you need to close and reopen the file to reposition the current record number at *linenum*. When this occurs and the file was not opened at the same task level as that of the program running, trying to close the file at a different task level causes an EXECIO error. Do not use the *linenum* operand in this case.

Specifying a value of `0` for *linenum* is the same as 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 just opened, EXECIO begins reading with the first line of the file.

You have to write a user or application I/O replaceable routine to have EXECIO exploit files such as SYSIN, SYSOUT, SYSRDR, or SYSLOG.

EXECIO DISKW on SYSLST and EXECIO DISKR from SYSIPT are supported. For SYSLST and SYSIPT you do not need to specify BLKSIZE, RECSIZE, or the RECFORM options.

The following values are used:

File name	BLKSIZE	RECSIZE	RECFORM
SYSLST (option CC)	121	121	FIXUNB
SYSLST (otherwise)	256	256	FIXUNB

File name	BLKSIZE	RECSIZE	RECFORM
SYSPCH	81	81	FIXUNB
SYSIPT	128	128	FIXUNB

For SYSLST with CC option specified, a record greater than 121 bytes is truncated with a return code rc=1. For SYSLST without CC option specified, a record greater than 120 bytes is truncated with a return code rc=1. If you want to use a different record length for SYSLST, you can specify operand RECSIZE *n* with 0<n<=256 (including carriage control character). In this case, you cannot specify option CC.

For SYSIPT, if a record size smaller than 128 bytes is desired, you can specify operand RECSIZE *n* with 0<n<=128.

BYTES *bytesnum*
If you want to process (read or write) a library member of type "string" in separate units, specify the BYTES operand followed by the number of bytes you want to handle as one unit. Smaller units require less storage to execute the command. The BYTES operand is only valid for library members with logical record format "string".

If you store (write) a new library member, the option BYTES implies that logical record format "string" is used for this library member.

STRTBYTE *strtnum*
Specifies a byte number within a library member of logical record format "string". STRTBYTE specifies the byte number where reading is to start. It is only valid together with operand BYTES.

Note that changing the position where reading should continue is always accompanied by an implicit close and re-open of the file.

Note:

1. You can read a DUMP or a PHASE either as a whole or broken up into portions using options BYTES and STRTBYTE.
2. Writing or reading for update of a PHASE is not possible. The open will fail with return code 20 and error messages containing LIBRM OPEN feedback (code 236, incorrect phase handling).
3. When you write a DUMP or other string-type members, use option BYTES. Otherwise, the library member is defined with fixed logical record format and only 80 bytes of the first record will be written. You also get a truncation return code of 1.

FINIS
closes the file after EXECIO completes. You can close a file only if it was opened at the same task level as the program issuing EXECIO.

You can use FINIS with a *lines* value of 0 to have EXECIO close an open file without first reading or writing a record.

The language processor environment is terminated after the end of a step in a batch job that called REXX. Therefore, all files a REXX program opens are typically closed automatically when the top level program ends. However, it is a good programming practice to explicitly close all files when finished with them.

OPEN
opens the specified file if it is not already open. For reading from a file, you can use OPEN with a *lines* value of 0 to have EXECIO do one of the following:

- Open a file without reading any records
- Set the current record number (that is, the number of the next record EXECIO will read) to the record number the *linenum* operand indicates, by specifying a value for *linenum*.

For writing to a file, you must use OPEN with a *lines* value of 0 to have EXECIO open a file without writing any records.

NODATA
DATA

This option is valid only for DISKW and is required only for opening a member of a sublibrary. (It is ignored for other types of files.) NODATA indicates the sublibrary member does not contain SYSIPT DATA. DATA indicates the sublibrary member contains SYSIPT DATA.

The default is NODATA for a new member. For a member that already exists, the default is its value from when it was created.

CC
NOCC
MCC
ASA

CC, NOCC, MCC, and ASA are valid only with SYSLST. CC, MCC, and ASA indicate treating the first character as a carriage control character. (The first character must be a valid American Standards Association (ASA) or machine control character. See the z/VSE System Macro Reference for a list of valid carriage control characters.)

NOCC indicates that EXECIO provides carriage control for the next line. NOCC is the default.

You can use CC, MCC, ASA, or NOCC for each single I/O request. This means your program can contain multiple EXECIO commands with different control character options for SYSLST.

STEM *var_name*

specifies the stem of the list of variables into which to place information or from which to write information. Compound variables permit indexing. To use compound variables, make sure the *var_name* ends with a period, for example, `myvar.`.

If you specify * as the number of lines to write, EXECIO stops writing information to the file when it finds a null line or an uninitialized compound variable. For example, if the list contains 10 compound variables, EXECIO stops at `myvar.11`.

In the following example, the list of compound variables has the stem `myvar.` and lines of information (records) are placed in variables `myvar.1`, `myvar.2`, `myvar.3`, and so forth.

```
"EXECIO * DISKR MYLIB.MYSUB.MYFILE.TYPEA (FINIS STEM MYVAR."
```

For reading from a file, the number of variables in the list is placed in `myvar.0`. Suppose 10 lines of information are read into the `myvar.` variables. Then `myvar.0` contains the number 10 (indicating that 10 records are read), and `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 value that was specified before issuing the EXECIO command.

To avoid confusion about whether a residual stem variable value is meaningful, you may want to clear the entire stem variable before issuing the EXECIO command. To clear all compound variables whose names begin with a particular stem, you can:

- Use the DROP instruction (for example, DROP *myvar.*) to set all possible compound variables whose names begin with that stem to the values of their own names in uppercase.
- Use an assignment to set all possible compound variables whose names begin with that stem to nulls (for example, *myvar.* = '').

Example “5” on page 154 shows using EXECIO with stem variables, and example “15” on page 157 illustrates the effect of residual data.

When writing an arbitrary number of lines from a file, *var_name*.0 has no effect on controlling the number of lines written.

Note: For reading from a file, if *var_name* does not end with a period, the variable names must be appended with numbers, but an index in a loop cannot access them. For writing to a file, if *var_name* does not end with a period, the variable names must be appended with consecutive numbers, such as `myvar1, myvar2, myvar3.`

Read Options

FIFO
places information on the data stack in FIFO (first in first out) order. FIFO is the default.

LIFO
places information on the data stack in LIFO (last in first out) order.

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 file.

Additional Options Required for SAM Files

Accessing SAM files requires additional information that is not needed for other files. Block size, record format, and (for certain record formats) record size are necessary for opening a file explicitly or implicitly (for example, to perform positioning within a file). You specify this information in the following additional options on the EXECIO command. These options are required whenever a file is opened. A file is opened explicitly if you specify the OPEN option. It is opened implicitly if:

- The file is not currently open.
- You switch from input processing (DISKR) to outprocessing (DISKRU or DISKW) or from output processing to input processing.
- *linenum* specifies a record number that precedes the current record number.

BLKSIZE *n*
specifies the block size of the file. The maximum size is 32761. See z/VSE System Macros User's Guide for details about the block size.

RECFORM FIXUNB
RECFORM FIXBLK
RECFORM VARUNB
RECFORM VARBLK
RECFORM UNDEF
specifies whether the record format is fixed unblocked, fixed blocked, variable unblocked, variable blocked, or undefined.

RECSIZE *n*
specifies the record size. This is required for FIXUNB and FIXBLK format records. Do not use RECSIZE for other record formats. Records are blank-extended if they are too short. If the records are too long, EXECIO ends with an error.

Closing Files

If you specify FINIS on EXECIO, the file is closed after EXECIO completes processing. If you do not specify FINIS, the file is closed:

- When the task, under which the file was opened, is terminated, or
- When the last language processor environment associated with the task, under which the file was opened, is terminated (even if the task itself is not terminated).
- Before a file is implicitly opened.

Whenever the file in VSAM-managed space is closed or opened (explicitly or implicitly) the file is processed according the open and close disposition on the DLBL statement, that is, the file may be defined, allocated, reset, or deleted. The initial positioning is handled according to the open disposition.

In general, when a REXX program is called, any files that the program opens are closed when the top-level program completes. For example, suppose you are running a program (top-level program) that calls another program. The second program uses EXECIO to open a file and then returns control to the first program without closing the file. The file is still open when the top-level program regains control. The top-level program can then read the same file continuing from the point where the nested program

finished EXECIO processing. When the top-level program ends, the file is automatically closed. (Example "12" on page 156 illustrates this.)

EXECIO Input Checking

The EXECIO options CC, NOCC, DATA, NODATA, BLKSIZE, RECFORM, and RECSIZE are ignored if they are specified differently than described in the EXECIO syntax diagram. For example, if you specify

```
EXECIO * DISKR SYSIPT 5 (CC
```

the CC option is ignored.

EXECIO only does a minimum of input checking. It is your reponsibility to correctly set up the input parameters. EXECIO uses the DTFDI, the DTFCP, or the DTFPR for SYSIPT, SYSLST, and SYSPCH, and the DTFSD for all other SAM filenames. Refer to the manuals *VSE/ESA System Macros User's Guide* and *VSE/ESA System Macros Reference* for details about DTFSD and DTFDI. EXECIO takes care of the 8 extra bytes required for output by the DTFSD macro for the BLKSIZE parameter. For example, if you use BLKSIZE 4096 to write a record you use BLKSIZE 4096 to read the record.

The largest size you can specify with the RECSIZE or BLKSIZE parameter in REXX/VSE is 32761.

REXX procedures using EXECIO to access library members may cause unusable library blocks if they are canceled. Use the librarian TEST command to restore those blocks.

An EXECIO return code rc=20 may have various reasons, for example

1. partition storage may be exhausted. Try a failing procedure in a larger partition.
2. a library member which is not accessible may be already opened by another partition.
3. end of extent has been reached.
4. a WRITE was issued for a file opened for READ.

Message ARX0565I may provide you with additional information why the EXECIO command failed.

Return Codes

The following table shows how EXECIO sets the REXX special variable RC.

Return Code	Meaning
0	Successful 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 you use * for number of lines because the remainder of the file is always read. For a member of a sublibrary, this return code may indicate the file is empty.
20	Severe error. EXECIO completed unsuccessfully and a message is issued. For a SAM file: • The file may not exist • You may have specified a record format, block size, or record size that does not match the file • A new file could not be defined.

Examples:

1. This example reads from a sublibrary member. The EXECIO command reads an entire PROC member into INPUT.1, INPUT.2, and so on, and closes the file when done.

```
'EXECIO * DISKR LIBNAME.SUBLIB.MEMBER.PROC (STEM INPUT. FINIS'
```

2. This example reads one line from SYSIPT and puts it on the stack in LIFO order. The EXECIO command does not close the file.

```
'EXECIO 1 DISKR SYSIPT (LIFO'
```

3. This example writes to a SAM file. You must previously use DLBL, for example

```
// DLBL FILE01,'MY.OUTPUT.FILE'
// EXTENT ,SYSWK1,,,13260,15
// ASSGN SYS007,231
```

   to assign a name (FILE01) to the file:

```
'EXECIO * DISKW FILE01 (STEM SAMFILE. BLKSIZE 64 RECFORM FIXBLK' ,
'RECSIZE 64'
```

   The file definition above refers to a specific disk location.

   Or you can specify the DLBL within your REXX procedure:

```
ADDRESS JCL "//DLBL FILE01,'MY.OUTPUT.FILE',,VSAM,CAT=VSESPUC," || ,
   "RECSIZE=65,RECORDS=(10,5),DISP=(NEW,KEEP)"
ADDRESS JCL "/*"
'EXECIO * DISKW FILE01 (STEM SAMFILE. BLKSIZE 64 RECFORM FIXBLK' ,
'RECSIZE 64'
```

   Here the file definition refers to a SAM-file in VSAM-managed space.

4. This example copies an entire existing SAM file named USERID.MY.INPUT into a member of an existing library named DEPT5.MEMO.MAR2.TEXT. You must previously use DLBL (for example, `// DLBL MYIPT,'USERID.MY.INPUT'`) to assign a name (MYINPUT) to the file USERID.MY.INPUT. The library member DEPT5.MEMO.MAR22.TEXT does not need any previous DLBL.

```
 "NEWSTACK" /* Create a new data stack for input only */

 "EXECIO * DISKR MYINPUT (FINIS BLKSIZE 64 RECFORM FIXUNB RECSIZE 64"
 QUEUE ''   /* Add a null line to indicate the end of information */
 "EXECIO * DISKW DEPT5.MEMO.MAR22.TEXT (FINIS"

 "DELSTACK" /* Delete the new data stack */
```

5. This example copies an arbitrary number of lines from an existing SAM file, USERID.TOTAL.DATA, into a list of compound variables. `DATA.` is the stem. You must previously use `// DLBL ALLDATA,'USERID.TOTAL.DATA'` to assign the name ALLDATA to the file USERID.TOTAL.DATA.)

```
 ARG lines
 "EXECIO" lines "DISKR ALLDATA (STEM data. BLKSIZE 64 RECFORM FIXUNB
RECSIZE 64"
 SAY data.0 'records were read.'
```

6. This example updates the second line in file DEPT5.EMPLOYEE.LIST. (You must previously use `// DLBL EMPLIST,'DEPT5.EMPLOYEE.LIST'`) to assign the name EMPLIST to the file.

```
 "EXECIO 1 DISKRU EMPLIST 2 (BLKSIZE 400 RECFORM FIXBLK RECSIZE 80"
 PULL line
 PUSH 'Crandall, Amy        AMY       5500'
 "EXECIO 1 DISKW EMPLIST (FINIS"
```

7. This example reads from a SAM file to find the first occurrence of the string "Jones". (You must previously use DLBL to associate the sequential file with the file name, INPUT.) The program ignores upper and lowercase distinctions. 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'
lineno = 0
DO WHILE done = 'no'
  "EXECIO 1 DISKR INPUT (BLKSIZE 100 RECFORM FIXBLK RECSIZE 100"
   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 INPUT (FINIS"
EXIT 0
```

8. This program copies records from the SAM file MY.INPUT.DATA to MY.OUT.DATA. (You must previously use DLBL to assign MY.INPUT.DATA the name INFILE and assign MY.OUT.DATA the name OUTFILE.) The program assumes that the input file has no null lines.

```
SAY 'Copying ...'

"EXECIO * DISKR INFILE (FINIS BLKSIZE 64 RECFORM VARBLK"
QUEUE ''   /* Insert a null line at the end to indicate end of file */
"EXECIO * DISKW OUTFILE (FINIS BLKSIZE 64 RECFORM VARBLK"

SAY 'Copy complete.'

EXIT 0
```

9. This program starts at the third record and reads five records from a SAM file to which you have assigned the name MYINPUT. 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 program is finished.

```
"EXECIO 5 DISKR MYINPUT 3 (BLKSIZE 64 RECFORM VARBLK"

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
```

10. This program reads the first 100 records (or until EOF) of the SAM file assigned the name INVNTOR. It places records on the data stack in LIFO order. It issues a message about the result of the EXECIO operation.

```
eofflag = 2             /* Return code to indicate end of file */

"EXECIO 100 DISKR INVNTOR (LIFO BLKSIZE 80 RECFORM VARBLK FINIS"
return_code = RC

IF return_code = eofflag THEN
   SAY 'Premature end of file.'
ELSE
   SAY '100 Records read.'
DROPBUF 0
EXIT return_code
```

11. This program erases any existing data from the SAM file FRED.WORKSET.FILE by opening the file and then closing it without writing any records. Doing this means EXECIO simply writes an end-of-file marker, which erases any existing records in the file. (You must previously use DLBL to assign FRED.WORKSET.FILE the name NAMES.)

```
/* Open the file for writing, but do not write a record.        */
"EXECIO 0 DISKW NAMES (OPEN BLKSIZE 64 RECFORM VARBLK"

/* Close the file.  This completes erasing any existing records */
"EXECIO 0 DISKW NAMES (FINIS"
```

    Note that in this example, `EXECIO ... (OPEN` followed by the `EXECIO ... (FINIS` is equivalent to:

```
"EXECIO 0 DISKW NAMES (OPEN FINIS BLKSIZE 64 RECFORM VARBLK"
```

12. The next example includes two programs. The first (top-level) program, PROG1, calls PROG2. PROG2 opens the file, reads the first three records, and then returns control to PROG1. Note that PROG2 does not specify FINIS on EXECIO, so the file remains open.

    When the PROG1 regains control, it issues EXECIO and gets the fourth record because the file is still open. If PROG2 had specified FINIS on EXECIO, PROG1 would have read the first record. In the example, both programs run at the same task level.

```
/*   PROG1 -- This program calls PROG2 to open a file.               */
/*   The file is a SAM file, and you must use DLBL to                */
/*   assign it a name before using EXECIO; for example:              */
/*       // DLBL myinput,'userid.my.input'                           */
/*   PROG1 then continues reading the same file.                     */
say 'Executing the first program PROG1'
/*                                                                   */
/*   Now call PROG2 to open the file.                                */
/*   This program uses a CALL instrucion to call the second program. */
/*   The REXX/VSE EXEC command would have the same result.           */
/*                                                                   */
/*   If PROG2 opens a file and does not close the file before        */
/*   returning control to PROG1, the file remains open when          */
/*   control is returned to PROG1.                                   */
/*                                                                   */
say 'Calling the second program PROG2'
call prog2                          /* Call PROG2 to open file      */
say 'Now back from the second program PROG2.  Issue another EXECIO.'
"EXECIO 1 DISKR MYINPUT (STEM Z.      /* EXECIO reads record 4       */
say z.1
say 'Now close the file'
"EXECIO 0 DISKR MYINPUT (FINIS"      /* Close file so it can be freed */
EXIT 0



/*  PROG2 -- This program opens the file MYINPUT, reads 3 records,   */
/*  and returns control to PROG1 without closing the file.           */
/*                                                                   */
say "Now in the second program PROG2"
DO I = 1 to 3                     /* Read and produce first 3 records */
   "EXECIO 1 DISKR MYINPUT (STEM Y. BLKSIZE 120 RECFORM VARUNB"
   say y.1
END
Say 'Leaving second program PROG2.  Three records were read from file.'
RETURN
```

13. This program opens the SAM file MY.INVNTORY without reading any records. The program then uses a main loop to read records from the file and process the records. (You must have previously used DLBL to assign the file the name INPUT and to assign MY.AVAIL.FILE the name OUTPUT.)

```
/* Open INPUT file for input, but do not read any records             */
"EXECIO 0 DISKR INPUT (OPEN BLKSIZE 100 RECFORM FIXBLK RECSIZE 100"

eof = 'NO'                          /* Initialize end-of-file flag    */
avail_count = 0                /* Initialize counter             */

DO WHILE eof = 'NO'                 /* Loop till EOF of input file    */
 "EXECIO 1 DISKR INPUT (STEM LINE." /* Read a line                    */
  IF RC = 2 THEN                    /* If end of file is reached,     */
```

```
    eof = 'YES'                     /* set end-of-file (eof) flag;     */
  ELSE                              /* otherwise, a record is read.    */
    DO
      IF INDEX(line.1,'AVAILABLE') THEN /* Look for records            */
                                        /* marked "available"          */
        DO                              /* "Available" record found       */

          /* Write record to available file          */
          "EXECIO 1 DISKW OUTPUT (STEM LINE. BLKSIZE 100 RECFORM FIXBLK
           RECSIZE 100"
          avail_count = avail_count + 1 /* Increment "available"
          counter  */

        END
    END
END

"EXECIO 0 DISKR INPUT (FINIS"  /* Close currently open INPUT file.    */

"EXECIO 0 DISKW OUTPUT (FINIS" /* Close OUTPUT file if currently open.    */
                               /* If OUTPUT file is not open,             */
                               /*   EXECIO has no effect.                 */


EXIT 0
```

14. This program opens SYSIPT and sets the current record number to record 8 so that the next EXECIO DISKR command begins reading at the eighth record.

```
"EXECIO 0 DISKR SYSIPT 8 (OPEN" /* Open file SYSIPT 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 EXECIO set
                                 the current record number to 8.   */
"EXECIO 0 DISKR SYSIPT (FINIS" /* Close the SYSIPT file.           */
EXIT

read_next_record:
      "EXECIO 1 DISKR SYSIPT (STEM Z."
       say z.1
       return
```

15. This program uses EXECIO to successively append the records from SAMPLE1.DATA and then from SAMPLE2.DATA to the end of the file ALL.SAMPLE.DATA. It illustrates the effect of residual data in STEM variables. SAMPLE1.DATA contains 20 records; SAMPLE2.DATA contains 10 records. (You must previously use DLBL to assign SAMPLE1.DATA the name IN1, SAMPLE2.DATA the name IN2, and ALL.SAMPLE.DATA the name OUT.)

```
/*********************************************************************/
/* Read all records from IN1 and  append them to the               */
/* end of OUT.                                                       */
/*********************************************************************/

program_RC = 0                /* Initialize exec return code      */

/* Read all records      */
"EXECIO * DISKR IN1 (STEM NEWVAR. FINIS BLKSIZE 80 RECFORM VARBLK"

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, and so on  */
  /* through newvar.20 contain the 20 records that were read.      */
  /*******************************************************************/
    say "-------------------------------------------------"
    say newvar.0 "records have been read from first input file."
    say
    do i = 1 to newvar.0      /* Loop through all records         */
      say newvar.i            /* Produce the ith record           */
    end

    /* Write exactly the number of records read       */
    "EXECIO" newvar.0 "DISKW OUT (STEM NEWVAR. BLKSIZE 80 RECFORM VARBLK"
    if rc = 0 then            /* If write was successful          */
      do
```

```
        say
        say newvar.0 "records were written to the output file."
      end
    else
      do
        program_RC = RC           /* Save program_return code   */
        say
        say "Error during 1st EXECIO ... DISKW, return code is " RC
        say
      end
  end
else
  do
    program_RC = RC               /* Save program_return code  */
    say
    say "Error during 1st EXECIO ... DISKR, return code is " RC
    say
  end

  If program_RC = 0 then       /* If no errors so far... continue  */
    do
    /*****************************************************************/
    /* At this time, the stem variables newvar.0 through newvar.20 */
    /* contain residual data from the previous EXECIO.              */
    /* The "DROP newvar." instruction clears these residual         */
    /* values from the stem.                                        */
    /*****************************************************************/
    DROP newvar.               /* Set all stems variables to their */
                               /* uninitialized state              */
    /*****************************************************************/
    /* Read all records from IN2 and append them to the             */
    /* end of OUTPUT.                                               */
    /*****************************************************************/
    /*Read all records*/
    "EXECIO * DISKR IN2 (STEM NEWVAR. FINIS BLKSIZE 80 RECFORM VARBLK"
     if rc = 0 then             /* If read was successful          */
      do
      /*****************************************************************/
      /* Now newvar.0 should be 10, indicating 10 records have     */
      /* been read. Stem variables newvar.1 through newvar.10      */
      /* 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 file.        */
      /* However, we would know that the last EXECIO DISKR did     */
      /* not read these values because the current newvar.0        */
      /* variable indicates the last EXECIO read only 10 records.  */
      /*****************************************************************/
        say
        say
        say "-----------------------------------------------------"
        say newvar.0 "records have been read from second input file."
        say
        do i = 1 to newvar.0   /* Loop through all records         */
          say newvar.i         /* Produce the ith record           */
        end

        /* Write exactly the number of records read  */
        "EXECIO" newvar.0 "DISKW OUT (STEM NEWVAR. BLKSIZE 80 RECFORM VARBLK"
        if rc = 0 then         /* If write was successful          */
         do
           say
           say newvar.0 "records were written to output file."
         end
        else
          do
            program_RC = RC  /* Save exec_return code           */
            say
            say "Error during 2nd EXECIO ...DISKW, return code is " RC
            say
          end
      end
    else
      do
        program_RC = RC       /* Save program_return code       */
        say
        say "Error during 2nd EXECIO ... DISKR, return code is " RC
        say
      end
  end

/* Close output file             */
```

```
"EXECIO 0 DISKW OUT (FINIS BLKSIZE 80 RECFORM VARBLK"

 exit 0
```

16. This example reads bytes 100 to 199 from a dump file, inserts string "REXX_CHANGE", and rewrites the dump.

```
'EXECIO 1 DISKRU SYSDUMP.BG.DBG00000.DUMP (STEM DUMP. BYTES 100',
   'STRBYTE 100 OPEN'
dump.1 = 'REXX_CHANGE' || dump.1
'EXECIO 1 DISKW SYSDUMP.BG.DBG00000.DUMP (STEM dump. BYTES 111',
   'FINIS'
```

HI

Note: This immediate command is available only from an application program. You specify HI on a call to ARXIC (see page “Trace and Execution Control Routine – ARXIC” on page 361) from a non-REXX program.

HI (Halt Interpretation) is an immediate command that halts the interpretation of all currently running programs. HI is available only if a program is running.

After HI, program processing ends or control passes to a routine or label if the halt condition trap has been turned on in the program. For example, if the program contains a SIGNAL ON HALT instruction and HI interrupts processing, control passes to the HALT: label in the program. See Chapter 7, “Conditions and Condition Traps,” on page 129 for information about the HALT condition.

HT

Note: This immediate command is available only from an application program. You specify it in a call to ARXIC (see page “Trace and Execution Control Routine – ARXIC” on page 361) from a non-REXX program.

HT (Halt Typing) is an immediate command that suppresses output that a program generates. The HT immediate command is available only if a program is running.

After HT, the program that is running continues processing, but the only output written to the current output device is output from commands that the program issues. All other output from the program is suppressed.

MAKEBUF

MAKEBUF creates a new buffer on the data stack.

Initially, the data stack contains one buffer, which is known as buffer 0. You can create additional buffers by using MAKEBUF. MAKEBUF returns the number of the buffer it has created in the REXX special variable RC. For example, the first time a program issues MAKEBUF, it creates the first buffer and returns a 1 in the special variable RC. The second time a program issues MAKEBUF, it creates another buffer and returns a 2 in the special variable RC.

The following table shows how MAKEBUF sets the REXX special variable RC.

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.

To remove buffers created with MAKEBUF from the data stack, use the DROPBUF command (see "DROPBUF" on page 144).

Example: A program places two elements, elem1 and elem2, on the data stack. The program calls a subroutine (sub3) that also places an element, elem3, on the data stack. The program 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 DROPBUF to remove the buffer it created.

```
/*  REXX program to ...    */
      ⋮
    "MAKEBUF"                                /* Creates buffer. */
    SAY 'The number of buffers created is' RC       /* RC = 1  */
    PUSH elem1
    PUSH elem2
    CALL sub3
      ⋮

exit
sub3:
     "MAKEBUF"                       /* Creates second buffer. */
    buffnum=RC
     PUSH elem3
      ⋮
     "DROPBUF" buffnum        /* Deletes second buffer created */
      ⋮
     RETURN
```

NEWSTACK

NEWSTACK creates a new data stack and hides or isolates the current data stack. A program cannot access elements on the previous data stack until it issues a DELSTACK command to delete the new data stack and any elements remaining in it.

After a program issues NEWSTACK, any element placed on the data stack with a PUSH or QUEUE instruction is placed on the new data stack. If a program calls a routine (function or subroutine) after issuing NEWSTACK, 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 use a NEWSTACK command, you must use a corresponding DELSTACK command to delete the data stack NEWSTACK created.

When there are no more elements on the new data stack, PULL obtains information from the input stream even though elements remain in the previous data stack. ASSGN(STDIN) returns the name of the current input device. (By default, this is SYSIPT.) To access elements on the previous data stack, use a DELSTACK command. If a new data stack was not created, DELSTACK removes all elements from the original data stack.

You can create multiple new data stacks but can access only elements on the most recently created data stack. To find out how many data stacks you have created, use the QSTACK command (page "QSTACK" on page 163). To find out the number of elements on the most recently created stack, use the QUEUED built-in function (page "QUEUED" on page 78).

If multiple language processor environments are chained together and you create a new data stack with NEWSTACK, the new data stack is available only to programs that run 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 NEWSTACK and DELSTACK as follows:

```
PUSH element1
PUSH element2
   ⋮
"NEWSTACK"     /* Creates data stack 2. */
CALL sub
"DELSTACK"     /* Deletes data stack 2. */
   ⋮
PULL stackelem
   ⋮
PULL stackelem
EXIT
```

2. To run a program named ABC that is a member in REXXLIB.SAMPLES.PROGRAM1.PROC specify REXX=*program_name* on the JCL EXEC statement.

```
// LIBDEF *,SEARCH=(PRD1.BASE,REXXLIB.SAMPLES)
// EXEC REXX=PROGRAM1
```

   Alternately, you could use the ARXJCL routine to run a REXX program. Specify ARXJCL on the JCL EXEC statement and specify in the PARM field the member name of the program and arguments:

```
// LIBDEF *,SEARCH=(PRD1.BASE,REXXLIB.SAMPLES)
// EXEC ARXJCL,PARM='PROGRAM1'
```

   This creates a new data stack. You can then put two elements on the new data stack for the program PROGRAM2.

```
"NEWSTACK"                /* Creates data stack 2. */
PUSH elem1
PUSH elem2
ADDRESS LINK "PROGRAM2"
   ⋮
"DELSTACK"                /* Deletes data stack 2. */
   ⋮
```

QBUF

QBUF queries the number of buffers that have been created on the data stack with the MAKEBUF command. QBUF returns the number of buffers in the REXX special variable RC. If you have not used MAKEBUF to create any buffers on the data stack, QBUF sets the special variable RC to 0. This is the only buffer the data stack initially contains.

QBUF returns the current number of data stack buffers created by a program and other routines (functions and subroutines) the program calls. You can issue QBUF from the calling program or from a called routine. For example, if a program 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.

Return Code	Meaning
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 a program creates two buffers on the data stack using MAKEBUF, deletes one buffer using DROPBUF, and then issues QBUF, RC is set to 1.

```
"MAKEBUF"              /* Creates buffer.                */
⋮
"MAKEBUF"              /* Creates second buffer.         */
⋮
"DROPBUF"              /* Deletes second buffer created. */
"QBUF"
SAY 'The number of buffers created is' RC      /* RC = 1 */
```

2. Suppose a program 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 program issues QBUF, this 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 returns the number of elements in the buffer that MAKEBUF most recently created. QELEM returns the number of elements in the REXX special variable RC. If MAKEBUF has not created any buffers, QELEM returns 0 in RC, regardless of the number of elements on the data stack. Thus, when QBUF returns 0, QELEM also returns 0.

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 a program issues DROPBUF, because DROPBUF removes the most recently created buffer and all elements in it.

The QUEUED built-in function (see page "QUEUED" on page 78) returns the total number of elements in the data stack, not including buffers.

The following table shows how QELEM sets the REXX special variable RC.

Return Code	Meaning
0	Either the MAKEBUF command has not been issued or the buffer that MAKEBUF most recently created contains no elements.
1	MAKEBUF has been issued, and there is one element in the current buffer.
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 a program creates a buffer on the data stack using MAKEBUF and then puts three elements on the data stack, QELEM returns the number 3.

```
"MAKEBUF"             /* Creates buffer.        */
PUSH one
PUSH two
PUSH three
"QELEM"
SAY 'The number of elements in the buffer is' RC        /* RC = 3 */
```

2. Suppose a program 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 program issues QELEM, it returns the number 1. The QUEUED function, however, which returns the total number of elements on the data stack, returns the number 3.

```
"MAKEBUF"             /* Creates buffer.        */
QUEUE one
PUSH two
"MAKEBUF"             /* Creates second buffer. */
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"      /* Deletes most recently created buffer    */
ELSE
   DO numelem
      PULL elem
      SAY  elem
   END
```

QSTACK

QSTACK queries the number of data stacks in existence for a program that is running. QSTACK returns the number of data stacks in the REXX special variable RC. The value QSTACK returns is the total number of

data stacks, including the original data stack. If you have not used NEWSTACK to create a new data stack, QSTACK returns 1 in the special variable RC.

QSTACK returns the current number of data stacks created by a program and by other routines (functions and subroutines) the program calls. You can issue QSTACK from the calling program or from a called routine. Suppose a program issues one NEWSTACK command and then calls a routine that issues another NEWSTACK command; if none of the new data stacks is deleted with DELSTACK, 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 459.
1	Only the original data stack exists.
2	The original data stack and one new data stack exist.
3	The original data stack and two new data stacks exist.
n	The original data stack and *n* - 1 new data stacks exist.

Examples:

1. Suppose a program creates two new data stacks using NEWSTACK and then deletes one data stack using DELSTACK. If the program issues QSTACK, QSTACK returns 2 in the REXX special variable RC.

```
"NEWSTACK"               /* Creates data stack 2. */
⋮
"NEWSTACK"               /* Creates data stack 3. */
⋮
"DELSTACK"               /* Deletes data stack 3. */
"QSTACK"
SAY 'The number of data stacks is' RC   /* RC = 2 */
```

2. Suppose a program 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 program issues QSTACK, it returns 5 in the REXX special variable RC. The data stack that is active is data stack 5.

```
"NEWSTACK"               /* Creates data stack 2. */
CALL sub1
"QSTACK"
SAY 'Data stacks =' RC   /* RC = 5 */
EXIT

sub1:
"NEWSTACK"               /* Creates data stack 3. */
CALL sub2
"QSTACK"
SAY 'Data stacks =' RC   /* RC = 5 */
RETURN

sub2:
"NEWSTACK"               /* Creates data stack 4. */
⋮
"NEWSTACK"               /* Creates data stack 5. */
"QSTACK"
SAY 'Data stacks =' RC  /* RC = 5 */
RETURN
```

RT

Note: This immediate command is available only from an application program. You specify it on a call to ARXIC (see page "Trace and Execution Control Routine – ARXIC" on page 361) from a non-REXX program.

The RT (Resume Typing) immediate command resumes producing output that was previously suppressed. The RT immediate command is available only if a program is running. Output that the program generated after the HT command and before the RT command is lost.

SETUID

SETUID lets you specify the user ID and password to be associated with an ADDRESS POWER command. You can set the user ID for the life of a REXX program or can modify it at any time during a REXX program.

Operands:

userid
is the user ID to use on subsequent requests to POWER. The *userid* must be from 1 to 8 characters. If you omit the *userid* or specify a *userid* of more than 8 characters, you receive return code -6.

password
is the password to associate with the given *userid* and subsystem communication request. The *password* must be from 1 to 8 characters. Supply a password when an ADDRESS POWER command would require it, for example when a VSE/POWER master password is needed for unlimited access (refer to the MPWD operand of the POWER generation macro described in the *VSE/POWER Application Programming*, SC33-6736 manual) or when a password protects a POWER queue entry. If master password has been specified, POWER does no longer use the *userid* for access checking; *userid* can be any string from 1 to 8 characters in this case.

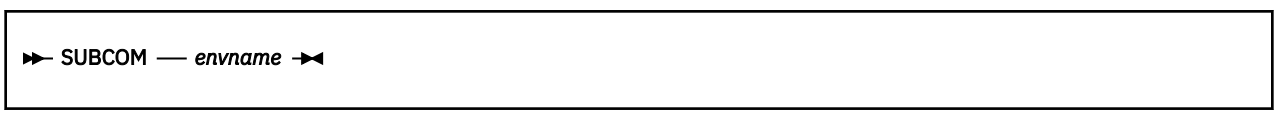
The *userid* and *password* combination are associated with each subsequent POWER command (VSE/POWER spool-access services CTL request), PUTQE command (VSE/POWER spool-access services PUT request), or GETQE command (VSE/POWER spool-access services GET request). The initial value of the *userid* is what the USERID built-in function would return (see "USERID" on page 92). If one REXX program calls another, the user ID in the calling REXX program is the initial user ID in the called program. The initial password is all blanks, or, if one REXX program calls another, the initial password is that of the calling REXX program. If you invoke SETUID without specifying a password, then the password is reset to the default of blanks.

Some ADDRESS POWER commands check the userid and the password and do not permit processing to continue if these do not match. See VSE/POWER Application Programming, for details about the scope of access. Any information you specify after the *password* causes a return code of -4. In this case, REXX does not change the *userid* and *password* values.

Examples:

```
"SETUID MYNAME1A"            /* Sets the user ID to MYNAME1A     */
"SETUID MYNAME1B"            /* Sets the user ID to MYNAME1B     */
"SETUID MYNAME1C MYPASSWD"   /* Sets the user ID to MYNAME1C     */
                             /*  and specifies password MYPASSWD */
```

SUBCOM



SUBCOM 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. If RC contains 0, the environment exists. If RC contains 1, the environment does not exist.

Before a program runs, a default host command environment is defined to process the commands that the program issues. You can use the ADDRESS keyword instruction (page "ADDRESS" on page 27) to change this environment to another environment if the environment is defined in the host command environment table. Use SUBCOM 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 (page "ADDRESS" on page 62) to determine the name of the environment to which host commands are currently being submitted.

Operands:

envname
is the name of the host command environment for which SUBCOM is to search.

REXX/VSE provides the following host command environments:

- VSE
- POWER
- LINK
- LINKPGM
- JCL
- CONSOLE

When you call a program, the default host command environment is VSE.

The following table shows how SUBCOM sets the REXX special variable RC.

RC Value	Description
0	The host command environment exists.
1	The host command environment does not exist.

Examples: To check whether the POWER environment is available before using the ADDRESS instruction to change the environment, use the SUBCOM command as follows:

```
"SUBCOM power"
IF RC = 0 THEN
  ADDRESS power
ELSE NOP
```

TE

Note: You can use TE in a REXX program or specify it in a call to ARXIC from a non-REXX program.

TE (Trace End) is an immediate command that ends tracing REXX programs. The TE immediate command is available if a program is running. The program continues processing, but tracing is off.

If you are running in interactive debug, you can also use TE in the current input stream to end tracing.

Example: A program 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 program.

TQ

Note: You can use TQ to test if tracing in a REXX program was set on or off.

TQ (Trace Query) is an immediate command available only from an application program. The program continues processing.

The following table shows how TQ sets the REXX special variable RC.

RC Value	Description
0	Processing was successful. REXX trace was set OFF by TE.
4	Processing was successful. REXX trace was set ON by TS.

TS

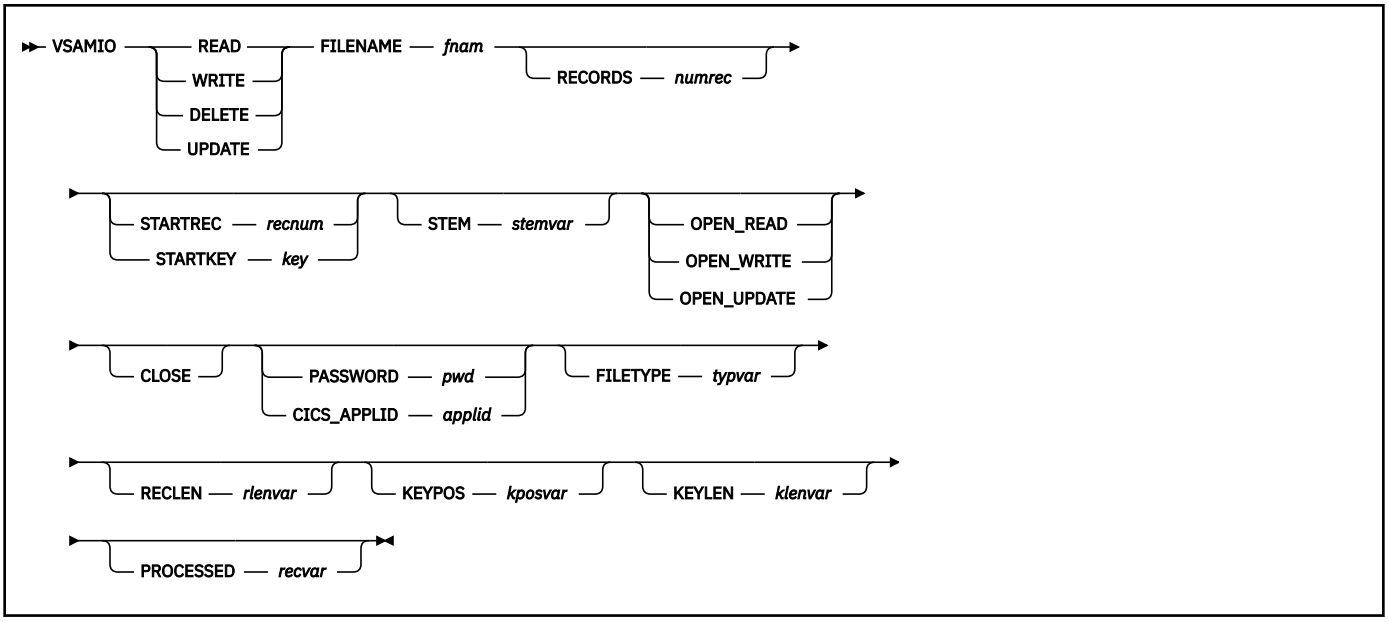
Note: You can use TS in a REXX program or specify it on a call to ARXIC from a non-REXX program.

TS (Trace Start) is an immediate command that starts tracing REXX programs. Tracing lets you control the execution of a program and debug problems. The TS immediate command is available if a program is running. The language processor writes trace output to the current output stream. ASSGN(STDOUT) returns the name of the current output stream.

To end tracing, you can use the TRACE OFF instruction or the TE immediate command. You can also use TE in the program to stop tracing at a specific point. If you are running in interactive debug, you can use TE to end tracing.

For more information about tracing, see the TRACE instruction on “TRACE” on page 53 and Chapter 16, “Debug Aids,” on page 319.

VSAMIO



VSAMIO controls the input and output (I/O) of information to and from a VSAM file. Supported operations are READ, WRITE, DELETE, and UPDATE.

VSAMIO can read or write data in REXX stem variables. If you use VSAMIO to read information from a VSAM file to a list of variables, the first file line is stored in *variable.1*, the second file line is stored in *variable.2*, and so on.

The various operands and combination of operands of the VSAMIO command permit you to do many types of I/O. For example, you can use the VSAMIO command to:

- Read information from a VSAM file
- Write information to a VSAM file
- Open a VSAM file without reading or writing any records.
- Empty a VSAM file
- Copy information from one VSAM file to another
- Copy information to and from a list of compound variables (REXX stem)
- Add information to a VSAM file
- Update information in a VSAM file
- Delete information in a VSAM file

There are three types of VSAM data sets supported by VSAMIO:

Key-Sequenced Data Set (KSDS)
is used when a record is accessed through a key field within the record. Every record in a KSDS must have a unique key value. An additional alternate index (AIX) via an additional unique or non-unique key field is possible.

Entry-Sequenced Data Set (ESDS)
is used for data that is primarily accessed in the order it was created. An additional alternate index (AIX) via an extra unique or non-unique key field is possible.

Relative Record Data Set (RRDS)
is used for data in which every item has a particular number, called Relative Record Number (RRN).

RRDS records in REXX/VSE consist of a prefix containing the RRN as the first word followed by the record data itself. When reading RRDS records, REXX/VSE returns them with a 12 character prefix starting with a blank, followed by a 10-digit-representation of the RRN, followed by another blank, for instance:

```
' 0000000022 data...data...data...'
```

When writing or updating a RRDS record, you have to specify the RRN-number as the first word of arbitrary length within the first 12 bytes of the record, for instance:

```
'22 data...data...data...data...'
'      00022 data...data...data...'
```

VSAM data sets can either be defined and used only via batch applications or they can be defined and used via one of the installed CICSes. VSAMIO can handle both: pure batch VSAM data sets, and CICS-defined data sets.

Besides the usual batch interface, an alternative processing is provided for CICS-defined VSAM files via cross partition communication with a CICS partition. This is especially useful, if read/write access is necessary for both, CICS and the REXX program. Due to access through CICS, the VSAM cluster must only be opened once by CICS; thus there is no need for defining the VSAM data set with shareoption 4.

Before VSAMIO can perform I/O to or from a VSAM file in batch mode, you have to use DLBL to associate the file with the file name. The following example associates *USERID.MY.INPUT* of catalog *MYCAT* with the file *MYINP*:

```
ADDRESS JCL "// DLBL MYINP,'USERID.MY.INPUT',,VSAM,CAT=MYCAT"
ADDRESS JCL "/*"
```

Operands:

READ
Copy records from a VSAM data set into a REXX stem variable starting with *stemvar.1*, *stemvar.2*, ... Variable *stemvar.0* contains the number of really copied records. The number of records copied are determined by operand RECORDS. Default is 1 record.

For VSAM data sets of types KSDS, KSDS AIX, and ESDS AIX, records are retrieved according to the key sequence. For ESDS data sets records are retrieved according to the sequence they were written, and RRDS data sets according to their relative record number sequence.

If data set is already opened and neither STARTREC / STARTKEY nor one of the options OPEN_READ, OPEN_WRITE, or OPEN_UPDATE are specified, reading starts at the current position. Reading increases the current position in the VSAM data set accordingly.

If none of the keywords OPEN_xxxx are specified for a closed VSAM data set and operation is READ, the data set is automatically opened for reading first.

WRITE
Copy a REXX stem variable to a VSAM data set. Every variable is copied, starting with *stemvar.1* and finishing with *stemvar.nnn*, where *nnn* is either the *numrec*-value (if operand RECORDS has been specified as a number), or the value of *stemvar.0* (if set to a whole number), or the predecessor of the first uninitialized stem variable (if RECORDS is specified as '*' and *stemvar.0* is not set), or the default value 1.

For VSAM data sets of types KSDS, and KSDS AIX, records are written according to the value contained in the key field. You can write in any key order, but it is most efficient to do it in key sequence. For ESDS and ESDS AIX data sets records are written to the end of the file, and RRDS data set records are written according to their value in the relative record number field.

If none of the keywords OPEN_xxxx are specified for a closed VSAM data set and operation is WRITE, the data set is automatically opened for updating first.

UPDATE
Replace records within a VSAM data set by new values provided in a REXX stem variable. Every variable is copied, starting with *stemvar.1* and finishing with *stemvar.nnn*, where *nnn* is either the *numrec*-value (if operand RECORDS has been specified as a number), or the value of *stemvar.0* (if set to a whole number), or the predecessor of the first uninitialized stem variable (if RECORDS is specified as '*' and *stemvar.0* is not set), or the default value 1.

For VSAM data sets of types KSDS and KSDS AIX, you can change the length of the record being updated. Stem values longer than the maximum size are truncated and RC is set to -1. You cannot change the key field of a record.

For ESDS and ESDS AIX data sets you cannot change the length of the record being updated. For ESDS AIX files you cannot change the reference key, too. If the updating stem value is smaller than the current record size, the initial part is changed and the rest remains the same as before. Stem values longer than the current record size are truncated and RC is set to -1.

RRDS files have fixed record length, thus the same record length rules apply as for ESDS files.

If an ESDS data set is already opened and neither STARTREC nor one of the options OPEN_READ, OPEN_WRITE, or OPEN_UPDATE are specified, updating starts at the current position. Updating increases the current position in the VSAM data set accordingly.

If none of the keywords OPEN_xxxx are specified for a closed VSAM data set and operation is UPDATE, the data set is automatically opened for updating first.

DELETE
Delete records from a non-ESDS VSAM data set. If data set is already opened for updating and neither STARTREC / STARTKEY nor one of the options OPEN_READ, OPEN_WRITE, or OPEN_UPDATE are specified, deleting starts at the current position in the VSAM data set.

If none of the keywords OPEN_xxxx are specified for a closed VSAM data set and operation is DELETE, the data set is automatically opened for updating first.

FILENAME *fnam*
refers to a DLBL name for the VSAM data set to be processed in batch mode, or the CICS-defined filename for the VSAM data set if processed through CICS.

For batch-processed data sets before using VSAMIO to perform I/O to or from a VSAM file, you need to assign a name to the file. You do this by using DLBL to associate the file with a file name.

RECORDS *numrec*
specifies the number of VSAM data set records to be processed.

Use '*' to read or delete the starting record together with all following records in the data set. Default for reading and deleting is 1 record, if operand RECORDS is not given.

For writing and updating, the default is the value specified in *stemvar.0*, if operand RECORDS is not mentioned. If even *stemvar.0* is not set, the default for writing and updating is 1 record. If RECORDS is specified as '*' and *stemvar.0* is not set, writing and updating stops when it reaches a null value or an uninitialized variable (one that has not been assigned a value).

If you specify a RECORDS value of 0, no I/O operations are performed unless you also specify OPEN_READ, OPEN_WRITE, OPEN_UPDATE, or CLOSE:

- If you specify OPEN_xxxx and the file is closed, VSAMIO opens the file in the given mode, but does not read, write, update, or delete any records. If you specify OPEN and the file is open in a different mode, VSAMIO reopens the file in the given mode.

  In either case, if you are processing a file and specify operand STARTREC or STARTKEY, VSAMIO sets the current record to the record indicated by STARTREC or STARTKEY. The current record is the record VSAMIO is to be read next without repositioning. By default, the current record is set to the first record when a file is opened.
- If you specify CLOSE and the file is open, VSAMIO does not process any records, but it closes the file.

STARTREC *recnum*
Positions to record number *recnum* for ESDS-filetype data sets and to the first record with a relative record number greater than or equal to *recnum* for RRDS-filetype data sets. (Re-)positioning always starts with a reset to the first record.

STARTKEY *key*
Positions to the first record with a key greater than or equal to the specified key for KSDS-filetype data sets or AIX-filetype data sets. (Re-)positioning always starts with a reset to the record with the smallest key.

If the key contains blanks, enclose it in single quotation marks. You can also specify a key in hexadecimal format, for example: X'C1C2C3'. If you specify a key smaller than the defined key length, only the initial part of the key is used for positioning (a "generic key search").

STEM *stemvar*
Specifies a REXX stem variable used to copy data from a VSAM data set to REXX (READ) or from REXX to a VSAM data set (WRITE, UPDATE).

OPEN_READ
Opens the VSAM data set only for reading. You can open a closed file without further processing, if you use OPEN_READ with a *numrec* value of 0.

If the data set is currently open in a different mode, the data set is reopened for reading. If the data set is already open for reading, the current record is reset to the record defined with STARTREC or STARTKEY if specified, otherwise it is reset to the first record.

If you process the VSAM data set via CICS, the CICS file definitions determine whether reading is allowed. CICS authorizations for Browsing, and Reading should be set to YES.

OPEN_WRITE
Opens the VSAM data set for (re-)writing. You can open a closed file without further processing, if you use OPEN_WRITE with a *numrec* value of 0.

If the data set is currently open in a different mode, the data set is reopened for writing.

If you process the VSAM data set via CICS, the CICS file definitions determine whether writing is allowed. CICS authorizations for Adding, Browsing, Deleting, Reading, and Updating should be set to YES. In this case there is no difference between OPEN_WRITE and OPEN_UPDATE.

OPEN_UPDATE
Opens the VSAM data set for updating and appending. You can open a closed file without further processing, if you use OPEN_WRITE with a *numrec* value of 0.

If the data set is currently open in a different mode, the data set is reopened for updating and appending. If operation is READ, DELETE, or UPDATE, the current record is reset to the record defined with STARTREC or STARTKEY if specified, otherwise it is reset to the first record.

If you process the VSAM data set via CICS, the CICS file definitions determine whether writing and updating is allowed. CICS authorizations for Adding, Browsing, Deleting, Reading, and Updating should be set to YES. In this case there is no difference between OPEN_WRITE and OPEN_UPDATE.

CLOSE
Closes the VSAM data set after VSAMIO completes. You can close an open file without further processing, if you use CLOSE with a *numrec* value of 0.

The language processor environment is terminated after the end of a step in a batch job calling REXX. Within this termination all still open files are closed automatically. However, it is good programming practice to explicitly close files no longer needed.

If you process the VSAM data set via CICS, this data set is only removed from the REXX internal administration, but it is kept open within the CICS partition.

PASSWORD *pwd*
Specifies the password for the VSAM data set. It consists of one through eight characters.

Password specification is not supported for CICS-processed VSAM files.

CICS_APPLID *applid*
Specifies the CICS Applid of the CICS used to process the operation on the VSAM data set. It consists of one through eight characters.

Specify this operand when "opening" a VSAM data set that should be processed via CICS. REXX/VSE saves attributes of opened data sets till they are closed; thus if CICS_APPLID has been specified at open time, succeeding operations are always performed via the given CICS even without extra specification of CICS_APPLID on the following VSAMIO commands.

A specific server task must be running within CICS to handle CICS-processed access to VSAM files. This corresponding CICS server task is usually started automatically within CICS. If not, it can be started explicitly invoking transaction ICVA. The CICS server task can be stopped explicitly using transaction ICVP.

The status of the file in CICS should be ENABLED.

FILETYPE ***typvar***
returns one of the values ESDS, KSDS, RRDS, ESDP (ESDS_Path), KSDP (KSDS_Path), NOTV (NotVSAM), or UNKN (Unknown)

RECLEN ***rlenvar***
returns the maximum length of a record of the given VSAM data set.

KEYPOS ***kposvar***
returns position of the VSAM key within records of the given VSAM data set.

KEYLEN ***klenvar***
returns the length of the VSAM key.

PROCESSED ***recvar***
returns the number of processed (read, written, updated, deleted) records.

Return Codes

Command VSAMIO returns one of the following return codes in the REXX special variable RC:

Return Code	Meaning
0	Successful processing
1	Successful processing, but at least one of the written or updated records has been truncated.
2	End-of-file has been reached.
3	A key problem has been detected. Possible reasons are: • A KSDS file is to be updated, but there exists no record with the given key value. • A RRDS file is to be updated, but there exists no record with the given Relative Record Number. • For KSDS files is the given STARTKEY higher than all existing keys in the file. • For RRDS files is the given STARTREC higher than all existing Relative Record Numbers in the file.
4	An empty data set is tried to be opened for reading only (VSAM RC 8, VSAM error code 110 from OPEN).
7	The VSE/VSAM file cannot be opened, since it is currently in use by another program (VSAM RC 8, VSAM error code 168 from OPEN).
8	An error occurred during a VSE/VSAM I/O operation. Messages ARX0690E and ARX0691E contain more information.
9	The record to be written contains a key that already exists in the file (VSAM RC 8, VSAM error code 8).
12	One of the VSAMIO functions WRITE, UPDATE, or DELETE is specified, but the file is opened only for reading.

Return Code	Meaning
16	VSAMIO fails because of a storage problem. Use a partition with more GETVIS space to run the REXX program.
20	A syntax error is detected in the VSAMIO command. One of the operand keywords may have a typing error.
21	None of the possible VSAMIO functions READ, WRITE, DELETE, or UPDATE is specified.
22	Specification of operand STEM is invalid due to one of the reasons: • For functions READ, WRITE, or UPDATE operand STEM is not specified. • Keyword STEM is specified without a token following. • Stem name is specified without a dot '.' at the end.
23	Operand FILENAME specifies an invalid filename: • FILENAME is not specified. • Keyword FILENAME is specified without a token following. • The length of the given filename is greater than 7.
24	Specification of operand PASSWORD or operand CICS_APPLID is invalid: • Operand PASSWORD specifies an invalid password, i.e. its length is greater than 8, or keyword PASSWORD is specified without a token following. • Operand CICS_APPLID specifies an invalid CICS application, i.e. its length is greater than 8, or keyword CICS_APPLID is specified without a token following.
25	Specification of operand STARTKEY is invalid due to one of the reasons: • STARTKEY is specified together with VSAM function WRITE. • STARTKEY is specified for a non-KSDS file. • STARTKEY is specified for function UPDATE of a KSDS-file. • Length of the STARTKEY value is greater than the keylength defined for the given KSDS file.
26	Specification of operand STARTREC is invalid due to one of the reasons: • STARTREC is specified together with VSAM function WRITE. • STARTREC is specified for a KSDS file. • STARTREC is specified for function UPDATE of a RRDS file.
27	Operand RECORDS specifies an invalid number of records.
28	Specification of the opening mode is invalid due to one of the reasons: • More than one of the keywords OPEN_READ, OPEN_WRITE, and OPEN_UPDATE are specified. • OPEN_READ is specified together with one of the functions WRITE, UPDATE, or DELETE. • OPEN_WRITE is specified together with one of the functions UPDATE, or DELETE.

Return Code	Meaning
29	Specification of a Relative Record Number as the first word of an RRDS record is invalid due to one of the reasons: • All the first 12 characters are blanks. • The first word in the record does not start with a digit. • The first word starts with more than 10 digits.
30	Function DELETE is specified for an ESDS file.
44	CEEPIPI invocation returns with an error. Message ARX0693 contains more information.
48	Invocation of ARXENTRY fails. This is usually an internal error. Messages contain more information.
52	A problem occurred with the Variable Pool Access Interface (ARXEXCOM) of REXX. Possible reasons are: • The value of a variable should be fetched, but the buffer for the copy is too small. • A variable name is not valid. • A variable value is not valid; it is may be too long.
99	Internal error, which should not occur. Please contact IBM.

Using the VSAMIO Command

Reading Information from a VSAM file

To read information from a VSAM file to a list of variables, use VSAMIO with the READ operand. To read all records from the VSAM file *MYINP*, you could use:

```
"VSAMIO READ FILENAME MYINP CICS_APPLID DBDCCICS",
"RECORDS * STEM newvar. OPEN_READ CLOSE"
```

VSAMIO READ places records from the file in compound variables. The name after keyword STEM must end with a period. If 10 lines of information are read, *newvar.1* contains record 1, *newvar.2* contains record 2, and so forth, up to *newvar.10*, which contains record 10. The number of items in the list of compound variables is in the special variable *newvar.0*. Thus, if 10 lines of information a read into the *newvar.* variables, *newvar.0* contains the number 10. Every stem variable beyond *newvar.10* is dropped, i.e. reset to its initial variable name value.

If *MYINP* is an RRDS file, the Relative Record Number of every record is stored within a 12-character prefix of *newvar.1*, *newvar.2*, and so on.

```
' 0000000004 Crandall, Amy           AMY        5421'
```

How to specify the number of records to read: In the preceding example, the asterisk after RECORDS specifies reading the entire file. To read a specific number of lines, put the number immediately after RECORDS:

```
"VSAMIO READ FILENAME MYINP RECORDS 25 STEM newvar. OPEN_READ CLOSE"
```

To read just one record, you can omit specification of RECORDS, since reading 1 record is the default.

```
"VSAMIO READ FILENAME MYINP STEM newvar."
```

To open a file without reading any records, specify 0 immediately after RECORDS and specify the OPEN_READ or OPEN_UPDATE operand.

```
"VSAMIO READ FILENAME RECORDS 0 OPEN_READ"
```

Using OPEN_READ or OPEN_UPDATE: Depending on the purpose you have for the input file, use either the OPEN_READ or OPEN_UPDATE operand.

- OPEN_READ - Reading Only

  To start I/O from a file that you want only to read, use the OPEN_READ operand. The CLOSE option closes the file after the information is read.

  ```
  "VSAMIO READ FILENAME ... RECORDS * ... CLOSE"
  ```

  Note: Do not use the CLOSE option if you want the next VSAMIO in your program to continue reading at the record immediately following the last record read.

- OPEN_UPDATE - Reading and Updating

  To start I/O to a file that you want to read and update, use the OPEN_UPDATE operand without the CLOSE option.

  More about using OPEN_UPDATE appears in "Updating Information...".

Option of specifying a starting record: If you want to start reading at a record other than the beginning of the file, specify operand STARTKEY for KSDS or AIX-files and operand STARTREC for ESDS/RRDS-files. For example, to read all the records of an ESDS- or RRDS-file starting at record 100, you could use:

```
"VSAMIO READ FILENAME MYINP RECORDS * STEM newvar. STARTREC 100 CLOSE"
```

To start at record 100 and read only 5 records, use:

```
"VSAMIO READ FILENAME MYINP RECORDS 5 STEM newvar. STARTREC 100 CLOSE"
```

To open a file at record 100 without reading any records, use:

```
"VSAMIO READ FILENAME MYINP RECORDS 0 STARTREC 100 OPEN_READ"
```

To read all the records of a KSDS-file starting with record "Smith ", you could use:

```
"VSAMIO READ FILENAME MYINP RECORDS * STEM newvar. STARTKEY 'Smith   '"
```

Writing Information to a VSAM File: To write information to a VSAM file from a list of variables, use VSAMIO with the WRITE operand. To write from the compound variables *newvar.1*, *newvar.2*, *newvar.3*, and so on to the VSAM file *MYINP*, you could use:

```
"VSAMIO WRITE FILENAME MYINP RECORDS * STEM newvar. CLOSE"
```

To write records to a VSAM RRDS file, specify the Relative Record Number as first word in *newvar.1*, *newvar.2*, and so on.

```
'0004 Crandall, Amy           AMY        5421'
```

How to specify the number of records to write: There exist several ways to define the number of records to write. You can specifiy a number immediately after VSAMIO:

```
"VSAMIO WRITE FILENAME MYINP RECORDS 25 STEM newvar."
```

You can assign a numeric value to *stemvar.0*

```
newvar.0 = 25
"VSAMIO WRITE FILENAME MYINP STEM newvar."
```

An asterisk after RECORDS means to write all stem variables starting with *stemvar.1, stemvar.2, ...* until a null value or an uninitialized compound variable is reached:

```
Drop newvar.
Do i=1 to 25; newvar.i = 'some data'; End
"VSAMIO WRITE FILENAME MYINP STEM newvar. RECORDS *"
```

If neither RECORDS, nor *stemvar.0* is specified, only data in *stemvar.1* is written.

To open a file without writing records to it, specify 0 after RECORDS and specify the OPEN_WRITE or OPEN_UPDATE operand.

```
"VSAMIO WRITE FILENAME MYINP RECORDS 0 OPEN_WRITE"
```

Note: To empty a batch-processed file, you can use the VSAMIO command:

```
"VSAMIO WRITE FILENAME MYINP RECORDS 0 OPEN_WRITE CLOSE"
```

Copying Information from One File to Another::

Copying an entire file: To copy the entire VSAM file *MYINP* to file *JOESINP*, you could use the following instructions:

```
"VSAMIO READ FILENAME MYINP RECORDS * OPEN_READ CLOSE STEM newvar."
"VSAMIO WRITE FILENAME JOESINP RECORDS * OPEN_WRITE CLOSE STEM newvar."
```

Copying a specified number of lines to a new file: To copy 10 lines of data from the VSAM file *MYINP* to the file *JOESINP*, you could use:

```
"VSAMIO READ FILENAME MYINP RECORDS 10 CLOSE STEM newvar."
"VSAMIO WRITE FILENAME JOESINP RECORDS 10 OPEN_WRITE CLOSE STEM newvar."
```

Adding lines to a file: To add 5 records from the VSAM file *MYINP* to the file *JOESINP*, you could use:

```
"VSAMIO READ FILENAME MYINP RECORDS 5 CLOSE STEM newvar."
"VSAMIO WRITE FILENAME JOESINP RECORDS 5 OPEN_UPDATE CLOSE STEM newvar."
```

Updating Information in a VSAM File:

Updating a KSDS file: Suppose you have a VSAM KSDS file named *MYKSDS* that contains a list of employee names, user IDs, and phone extensions. Its key starts at position 0 with a length of 24 bytes. One record is this one:

```
Crandall, Amy           AMY        5421
```

You can change this information. For example, to change phone extension to 5500, you could use:

```
"VSAMIO READ FILENAME MYKSDS STARTKEY 'Crandall, Amy' RECORDS 1",
 "STEM newvar. OPEN_UPDATE"
newvar.1 = Substr(newvar.1,1,WORDINDEX(newvar.1,4)-1) || '5500'
"VSAMIO UPDATE FILENAME MYKSDS RECORDS 1 STEM newvar. CLOSE"
```

Updating an ESDS file: Suppose you have a CICS-defined VSAM ESDS file named *MYESDS* that contains a list of employee names, user IDs, and phone extensions. The 5th record is this one:

```
Crandall, Amy           AMY        5421
```

You can change this information. For example, to change phone extension to 5500, you could use:

```
.
 "VSAMIO READ CICS_APPLID DBDCCICS FILENAME MYESDS STARTREC 5 RECORDS 1",
  "STEM newvar. OPEN_UPDATE"
 newvar.1 = Substr(newvar.1,1,WORDINDEX(newvar.1,4)-1) || '5500'
 "VSAMIO UPDATE FILENAME MYESDS STARTREC 5 RECORDS 1 STEM newvar. CLOSE"
```

Updating an RRDS file: Suppose you have a VSAM RRDS file named *MYRRDS* that contains a list of employee names, user IDs, and phone extensions. The record with Relative Record Number 5 is this one:

```
Crandall, Amy           AMY        5421
```

You can change this information. For example, to change phone extension to 5500, you could use:

```
 "VSAMIO READ FILENAME MYRRDS STARTREC 5 RECORDS 1",
  "STEM newvar. OPEN_UPDATE"
newvar.1 = Substr(newvar.1,1,WORDINDEX(newvar.1,5)-1) || '5500'
"VSAMIO UPDATE FILENAME MYRRDS RECORDS 1 STEM newvar. CLOSE"
```

The VSAMIO READ returns in *newvar.1*:

```
' 0000000005 Crandall, Amy           AMY        5421      '
```

Deleting Information in a VSAM File:

Deleting records in a KSDS file: Suppose you have a VSAM KSDS file named *MYKSDS* that contains a list of employee names, user IDs, and phone extensions. Its key starts at position 0 with a length of 24 bytes. One record is this one:

```
Crandall, Amy           AMY        5421
```

You can change this information. For example, to change phone extension to 5500, you could use:

```
 "VSAMIO DELETE FILENAME MYKSDS STARTKEY 'Crandall, Amy' RECORDS 1"
```

Deleting Information in an ESDS file: It is not possible to delete ESDS file records!

Deleting Information in an RRDS file: Suppose you have a VSAM RRDS file named *MYRRDS* that contains a list of employee names, user IDs, and phone extensions. The record with Relative Record Number 5 is this one:

```
Crandall, Amy           AMY        5421
```

To delete this record, you could use:

```
 "VSAMIO DELETE FILENAME MYRRDS STARTREC 5 RECORDS 1"
```

Examples:

1. This example reads an entire VSAM file into input.1, input.2, and so on, and closes the file when done. As always, you must previously use DLBL:

   ```
   ADDRESS JCL "// DLBL VSMFILE,'VSAM.CLUSTER1',,VSAM,CAT=VSESPUC"
   ADDRESS JCL "/*"
   'VSAMIO READ FILENAME VSMFILE STEM input. RECORDS * PASSWORD THISPW CLOSE'
   SAY input.0 'records have been read.'
   ```

2. This example creates a VSAM ESDS file with 10 records.

   ```
   input.0 = 10
   Do i=1 to 10
     input.i = right(i,i,'0') || ' this is record ' i
   End
   'VSAMIO WRITE FILENAME VSMESDS STEM input. OPEN_WRITE CLOSE' ,
     'FILETYPE ftyp RECLEN recl'
   ```

   Three records are updated.

   ```
   Do i=1 to 3
     update_input.i = right(i+3,i+3,'0') || ' this is update ' i+3
   End
   'VSAMIO UPDATE FILENAME VSMESDS STEM update_input. RECORDS 3 STARTREC 4'
   ```

   The file is closed.

```
'VSAMIO READ FILENAME VSMESDS RECORDS 0 CLOSE'
```

3. This example creates a VSAM KSDS file with 10 records. Key consists of the first 8 bytes in the record.

```
input.0 = 10
Do i=1 to 10
  input.i = right(i,8,'0') || ' this is record ' i
End
'VSAMIO WRITE FILENAME VSMKSDS STEM input. OPEN_WRITE CLOSE' ,
  'FILETYPE ftyp RECLEN recl KEYLEN keyl KEYPOS keyp'
```

   Then records with keys 00000009 and 00000010 are deleted.

```
'VSAMIO DELETE FILENAME VSMKSDS RECORDS 2 STARTKEY 00000009'
```

   Three other records are updated.

```
Do i=1 to 3
  update_input.i = right(2*i,8,'0') || ' this is updated record ' 2*i
End
'VSAMIO UPDATE FILENAME VSMKSDS STEM update_input. RECORDS 3'
```

   The file is closed.

```
'VSAMIO DELETE FILENAME VSMKSDS RECORDS 0 CLOSE'
```

4. This example creates a VSAM RRDS file with 10 records.

```
input.0 = 10
Do i=1 to 10
  input.i = right(i,3,'0') || ' this is record ' i
End
'VSAMIO WRITE FILENAME VSMRRDS STEM input. OPEN_WRITE CLOSE' ,
  'FILETYPE ftyp RECLEN recl'
```

   Then records 9 and 10 are deleted.

```
'VSAMIO DELETE FILENAME VSMRRDS RECORDS 2 STARTREC 9'
```

   Three other records are updated.

```
Do i=1 to 3
  update_input.i = right(2*i,8,'0') || ' this is updated record ' 2*i
End
'VSAMIO UPDATE FILENAME VSMRRDS STEM update_input. RECORDS 3'
```

   The file is closed.

```
'VSAMIO DELETE FILENAME VSMRRDS RECORDS 0 CLOSE'
```

5. This example copies VSAM file VSMESDA into file VSMESDS, and appends another VSAM file VSMESDB to this file VSMESDS.

```
'VSAMIO READ FILENAME VSMESDA STEM content. RECORDS * CLOSE'
'VSAMIO WRITE FILENAME VSMESDS STEM content. OPEN_WRITE'
'VSAMIO READ FILENAME VSMESDB STEM content. RECORDS * CLOSE'
'VSAMIO WRITE FILENAME VSMESDS STEM content. CLOSE'
```

6. This example copies again VSAM file VSMESDA into file VSMESDS, and appends another VSAM file VSMESDB to this file VSMESDS. Only 10 records are copied at once.

```
records_at_once = 10
```

   Files VSMESDA and VSMESDS are opened.

```
'VSAMIO READ FILENAME VSMESDA  RECORDS 0 OPEN_READ'
'VSAMIO WRITE FILENAME VSMESDS RECORDS 0 OPEN_WRITE'
```

   VSMESDS is copied into VSMESDA.

```
recnum_A = records_at_once
Do Until recnum_A < records_at_once
  'VSAMIO READ FILENAME VSMESDA STEM content. RECORDS ' records_at_once ,
    'PROCESSED recnum_A'
  'VSAMIO WRITE FILENAME VSMESDS STEM content.'
End
```

File VSMESDA is closed.

```
'VSAMIO READ FILENAME VSMESDA  RECORDS 0 CLOSE'
```

File VSMESDB is opened.

```
recnum_B = records_at_once
'VSAMIO READ FILENAME VSMESDB  RECORDS 0 OPEN_READ'
```

File VSMESDB is appended to VSMESDS.

```
Do Until recnum_B < records_at_once
  'VSAMIO READ FILENAME VSMESDB STEM content. RECORDS ' records_at_once ,
    'PROCESSED recnum_B'
  'VSAMIO WRITE FILENAME VSMESDS STEM content.'
End
```

Files VSMESDB and VSMESDS are closed.

```
'VSAMIO READ FILENAME VSMESDB  RECORDS 0 CLOSE'
'VSAMIO WRITE FILENAME VSMESDS RECORDS 0 CLOSE'
```

Chapter 11. ADDRESS POWER Commands

The POWER host command environment exploits VSE/POWER spool-access services requests, GET, PUT, and CTL. (See “The POWER Host Command Environment” on page 25 for details about the POWER environment.)

ADDRESS POWER commands include.

- GETQE command, which performs the GET function. This retrieves an entry from a POWER queue.
- PUTQE command, which performs the PUT function. This places a job on a POWER queue.
- QUERYMSG command, which returns job completion message(s) into the stem specified by OUTTRAP.
- POWER commands that you can issue through a CTL service request. You can use ADDRESS POWER to send these commands to VSE/POWER. “CTL” on page 196 lists these commands. You can also find them in VSE/POWER Application Programming, SC33-6736. The documentation VSE/POWER Administration and Operation, SC33-6733 contains their syntax.

Output for these commands or error information is trapped by the OUTTRAP function. Please refer to the description of this function on “OUTTRAP” on page 94.

Accessing Entries in VSE/POWER Queues

When using GETQE or a CTL command, follow the programming interface rules for the VSE/POWER spool-access services interface GET/CTL Service. That is, provide a user ID and password when needed. (See VSE/POWER Application Programming, for details.) The user ID associated with the GETQE request is the one determined according to the rules described for function USERID on page “USERID” on page 92. It must match the user ID associated with the queue entry (the job) being retrieved. The password associated with the GETQE/CTL request is the last one specified in a SETUID command.

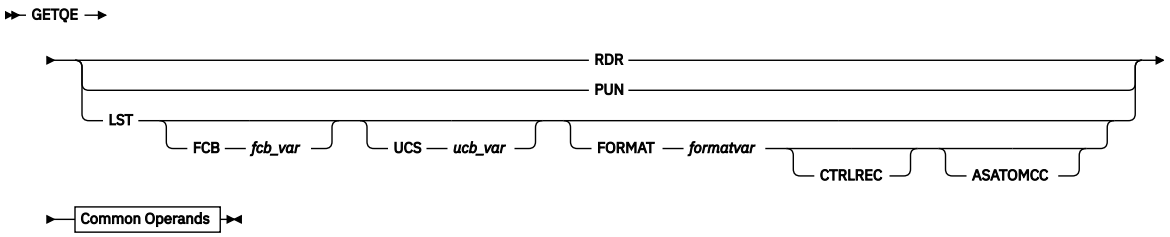
Access is possible to the following job entries in the VSE/POWER RDR queue and to the following output entries in the VSE/POWER LST or PUN queue:

- Job or output entries with the same node and user ID as origin
- output entries with the same node and user ID as destination
- Jobs and their output entries which contained the FROM parameter specifying the origin user ID in the * $$ JOB statement
- For GETQE only: output entries with destination user ID ANY

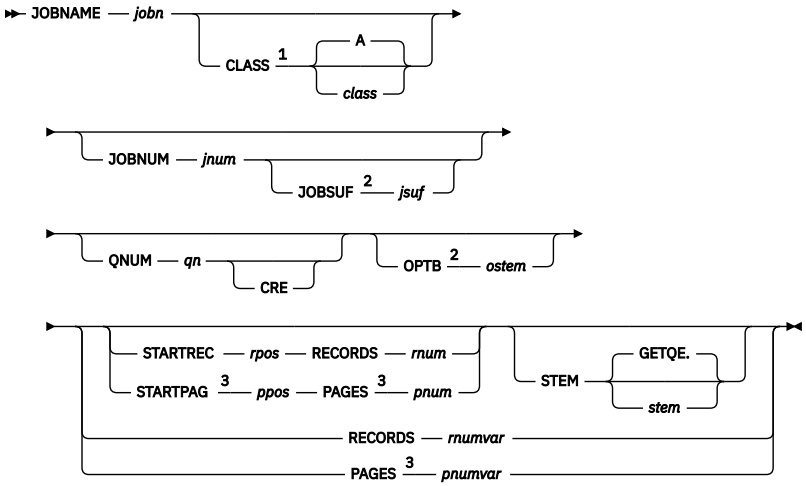
If an installation specific POWER master password has been defined in the system, unlimited access is available by specifying this POWER master password with the SETUID command (see “SETUID” on page 165). It provides access to all queue entries even if the user ID does not match. This is the only way to access queue entries without a specified FROM/TO user ID.

See also the VSE/POWER Application Programming, for information about the scope of access to queue entries and how POWER sets the user ID for a job. See page “USERID” on page 92 for information about the USERID function.

GETQE



Common Operands



Notes:

[1] You can include optional parameters in any order.
[2] LST, PUN Queue only.
[3] LST Queue only.

GETQE retrieves an entry from a POWER queue (RDR, LST, or PUN) and stores the lines it retrieves in compound variables.

Note:

1. If GETQE is not successful **stem.0** is not set.
2. If you use an operand but do not specify a corresponding value, no default is assumed. But
   - if you do not specify the STEM operand it defaults to GETQE
   - If you do not specify the CLASS operand it defaults to A.
3. Carriage control (CC) characters are only processed if the FORMAT parameter is specified.
4. When you retrieve an entry from the LST or PUN queue that you put on the queue with PUTQE, the record format of the entry is MCC. This applies if FORMAT is not specified or FORMAT=MCC (default).
5. For VSE/POWER there are 2 types of records within a LST queue entry:
   - logical records containing user data, also often named 'lines'
   - immediate control records like 'Skip to Channel immediately', or 'Space 1 line immediately'

   If the CTRLREC operand has not been specified, only the logical records (the 'lines') are considered. If the CTRLREC operand has been specified, both types of records are taken into account.

Operands

RDR
LST
PUN
is the queue from which to obtain the specified entry.

FCB *fcb_var*
specifies a variable where you want to receive the name of the FCB-image phase VSE/POWER is to use for printing the related LST output.

UCS *ucb_var*
specifies a variable, where you want to receive the UCB (universal character set buffer) information assigned to a LST queue entry. The information is returned in the format *(<phasenam>,<op>)*, where *<phasenam>* is the name of the UCB-image phase loaded into the UCB of the printer, and *<op>* is either 'F ', 'C ', or 'FC'. An 'F' indicates that the UCB is to be loaded with the folding operation code causing lowercase letters to be printed in uppercase, and 'C' prevents data checks from being generated because of print-line mismatches with the UCB. If none of the two options are set, two blanks are set. If no UCB-image phase has been defined for a LST queue entry, <phasenam> consists of 8 blanks.

FORMAT *formatvar*
specifies a variable where you want to receive the type of printer output the LST queue entry contains. *formatvar* stores one of the following values:

- SCS
- MAP
- T3270
- CPDS
- ESC
- MCC
- ASA

If a FORMAT operand has been specified, the print record will have a one byte prefix containing the print control character followed by at least one byte containing the data.

CTRLREC
requests delivery of immediate control records. If CTRLREC has not been specified, those records are skipped.

Especially if output of GETQE is used later on to generate another entry in the VSE/POWER LST queue with command PUTQE, carriage control characters might be important because they determine the page count maintained by VSE/POWER for formats like MCC and ASA.

ASATOMCC
Usually GETQE offers ASA-controlled data records unchanged. But you can request ASA to matching control conversion using this keyword. Then you get for every ASA data record two machine control records:

- a first one doing the forms control operation.
- a second one writing the actual data immediately.

JOBNAME *jobn*
jobn is the job name of the queue entry to retrieve.

CLASS *class*
class is the class of the queue entry to retrieve. If you do not specify the class, it defaults to A.

JOBNUM *jnum*
jnum is the number VSE/POWER has assigned to the queue entry you want to retrieve. If you omit this, REXX does not pass a number to VSE/POWER.

JOBSUF *jsuf*
jsuf specifies the segment number to be retrieved (provided you use VSE/POWER output segmentation for LST and PUN queue entries). *jsuf* is a number between 1 and 127. If JOBSUF is omitted, GETQE retrieves the first segment.

QNUM *qn*
qn specifies a POWER queue entry via its queue record number.

CRE
indicates that an in-creation entry is to be read.

OPTB *ostem*
ostem specifies the REXX stem to receive settings of user-defined output operands. The values are retrieved in the form: KEYWORDID={value|(value,...)} in *ostem.n*, for example, OPTBSTEM.1 = '001F=ABC'.

STARTREC *rpos*
rpos specifies the record number where retrieval is to start, if you want to have only part of the queue entry retrieved. If CTRLREC is not specified record numbering is based on logical records only. Otherwise record numbering is based on all records including the immediate control records.

RECORDS *rnum*
rnum specifies the number of records to be retrieved, if you want to have only part of a queue entry retrieved. Record numbering is depended on the existence of the CTRLREC operand. If CTRLREC has not been specified, record numbering is based on logical records only. Otherwise both logical records and immediate control records are counted.

STARTPAG *ppos*
ppos specifies the page number where retrieval is to start, if you want to have only part of a LST queue entry retrieved. If the LST queue entry does not start with a printer control record to start a new page, specify STARTPAG 0 to get the first records within this entry.

PAGES *pnum*
pnum specifies the number of pages to be retrieved, if you want to have only part of a LST queue entry retrieved.

STEM GETQE.
STEM *stem*
specifies the name of a stem. (A stem must end in a period.) GETQE stores lines into compound variables whose names begin with this stem. GETQE. is the default stem. The *stem* must be valid according to REXX rules for naming stems. (See "Stems" on page 21.) If a stem is not valid, the return code in the special variable RC is -22.

RECORDS *rnumvar*
rnumvar specifies a variable where you want to receive the number of records of a queue entry. With this request, you may not specify STARTREC or STEM. Only the number of records is returned, but queue entry data will not be copied. Record numbering is depended on the existence of the CTRLREC operand. If CTRLREC has not been specified, the number of the logical records is returned. Otherwise the number of logical records plus the number of immediate control records is returned.

PAGES *pnumvar*
pnumvar specifies a variable where you want to receive the number of pages of a LST queue entry. With this request, you may not specify STARTPAG or STEM. Only the number of pages is returned, but queue entry data will not be copied.

The stem is first dropped (as if the REXX instruction DROP *stem* had been used) before the retrieval. Then, when the entry is returned, each line of it is stored into the variable *stem.n*, where *n* is the record number of the entry. *Stem*.0 contains the number of lines in the entry. Error information is written to the current output file. ASSGN(STDOUT) returns the name of the current output file. If trapping is active, error information is also stored in the compound variables that the user specifies on OUTTRAP. The error information contains decimal numbers identifying the VSE/POWER spool-access services return and feedback codes describing the failure. REXX/VSE error message ARX0950E contains the return code from the VSE/POWER spool-access services interface. (See page "OUTTRAP" on page 94 for details about using OUTTRAP.)

The queue element disposition is unchanged because REXX uses a MODE of BROWSE. For the GETQE command the POWER access rules apply as described on "Accessing Entries in VSE/POWER Queues" on page 181. See the VSE/POWER Application Programming, for more details.

Security Considerations

GETQE allows retrieval of VSE/POWER queue elements for all user IDs set by the SETUID command. However, if the VSE system is secured (VSE IPL statement SYS SEC=YES), REXX introduces the following security checking for the GETQE command:

1. If you specified a VSE/POWER user ID and password via SETUID, then the password and user ID are passed to VSE/POWER for further security checking. If VSE/POWER refuses access, a RC=-13 with message ARX0950E occurs.
2. If you specified no VSE/POWER password via SETUID and the VSE security user ID of the executing job is authorized as administrator, then ADDRESS POWER GETQE is allowed to retrieve all queue elements.
3. If you specified no VSE/POWER password via SETUID and the VSE security user ID of the executing job is not authorized as administrator, then only the queue elements owned by the VSE security user ID can be retrieved, otherwise a security violation occurs with RC=-26. The REXX user ID set by SETUID must be equal to the VSE security user ID.

A VSE security ID can be set, for example, by the ID statement (see z/VSE System Control Statements) or by the SEC parameter of the VSE/POWER job statement (see VSE/POWER Application Programming)

Return Codes

When the VSE/POWER spool-access services interface encounters an error, the REXX special variable RC is set. Error information is written to the current output stream. You can use ASSGN(STDOUT) to return the name of the current output stream. The following table shows how GETQE sets the REXX special variable RC.

Return Code	Meaning
0	Successful processing.
-13	A severe XPCC or POWER error.
-14	General use storage could not be obtained.
-16	Storage problem occurred during set up to get connection to VSE/POWER spool-access services interface.
-17	Connection to VSE/POWER spool-access services failed.
-19	Incorrect input from parameter list.
-20	Error in STEM variable.
-22	The stem was not valid.
-26	Security violation.
-29	Invalid combination of operands defining the part of the VSE/POWER queue to be retrived, for example: • PAGES are specified, but POWER queue is not LST. • Starting record or page number is specified, but not the number of records or pages to be retrieved. • As well the starting record number as the starting page number are specified. • The number of records or pages is specified, but also STEM. • Some of the operands STARTREC, PAGES, or RECORDS are zero.
-32	Operand QNUM is invalid.

Return Code	Meaning
-33	Operand CRE can only be specified together with operand QNUM.
-36	Operand UCS is invalid.

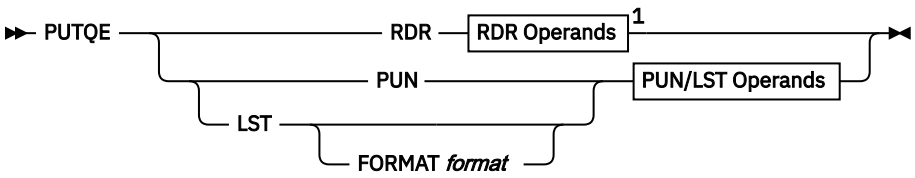
Note: OUTTRAP (page "OUTTRAP" on page 94) can trap error information from GETQE. If trapping is active, error information is also written to the stem that you specify on the OUTTRAP function.

Example

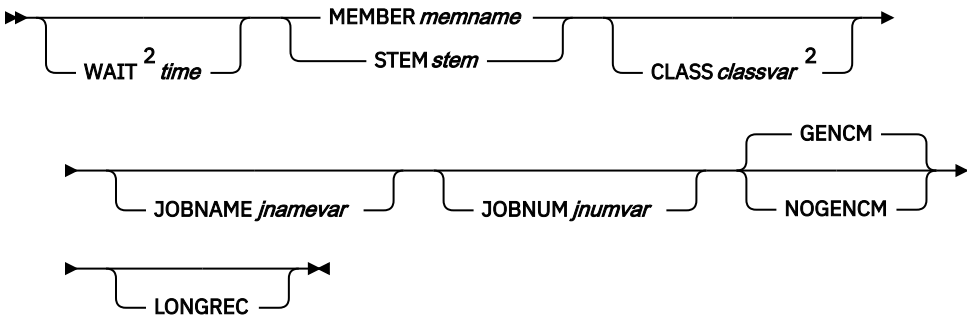
The following example retrieves the job with name MAKEJCL from the RDR queue. The job is class B and has job number 3450. GETQE stores the lines it retrieves in compound variables beginning with the stem FORJCL..

```
"GETQE RDR JOBNAME MAKEJCL CLASS B JOBNUM 3450 STEM FORJCL."
```

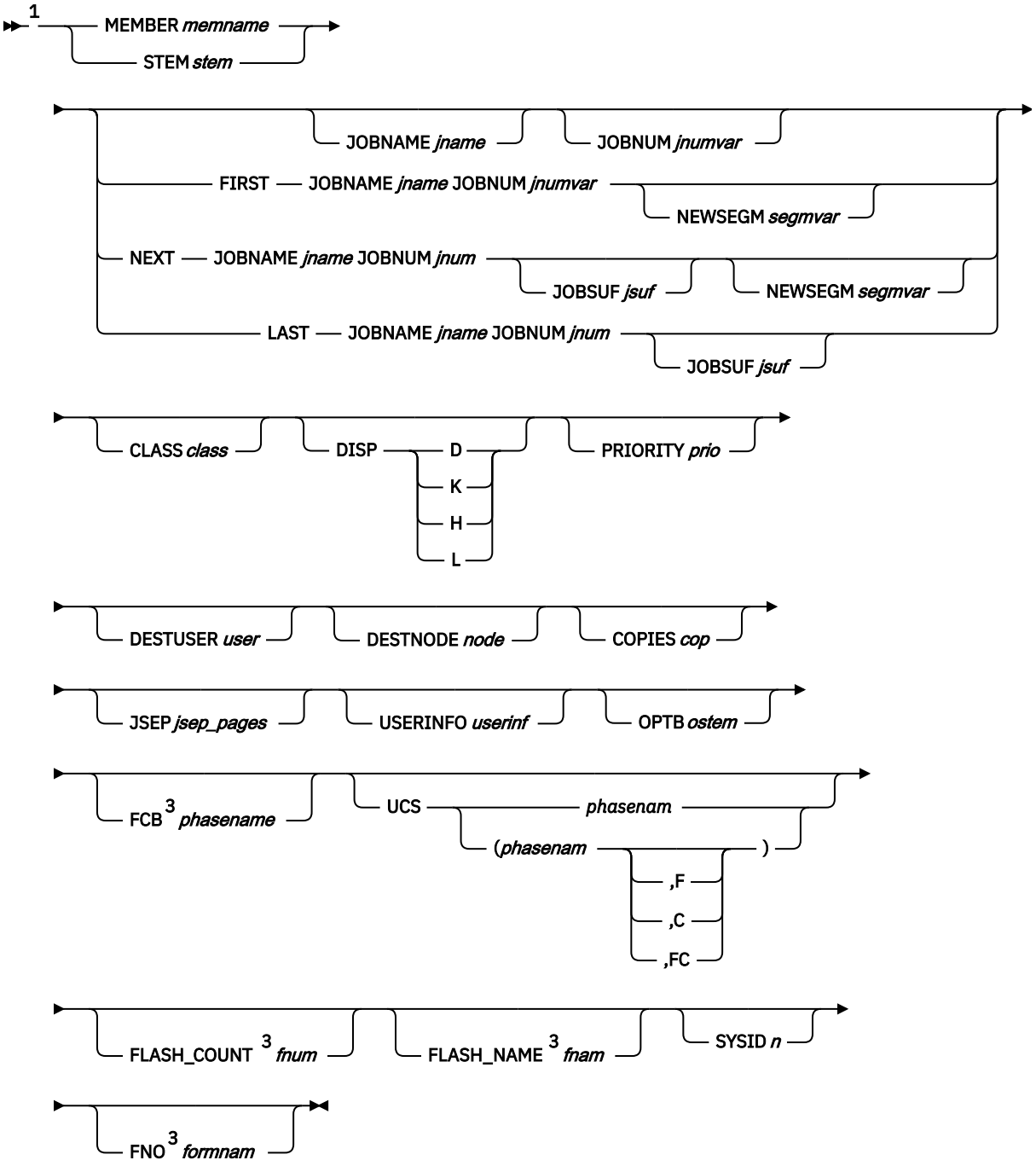
PUTQE



RDR Operands



PUN/LST Operands



Notes:

[1] You can include RDR operands and LST and PUN operands in any order.

[2] If you omit the WAIT operand or specify `WAIT 0`, the language processor ignores the CLASS operand. The contents of *classvar* is only considered if the WAIT option was specified.

[3] LST Queue only.

PUTQE places a job on a VSE/POWER queue (RDR, LST, or PUN). This job can be one that already exists in the VSE librarian or one the currently running REXX program created (through EXECIO) and stored into a stem.

Note:

1. For rules about naming jobs and classes, see VSE/POWER Application Programming, and VSE/POWER Administration and Operation.

2. No carriage control characters are processed if the FORMAT parameter has not been specified or FORMAT=NOCC.
3. If you specify PUN, the record format of the entry is Machine Control Character (MCC). For each supplied data record, the carriage control character X'01' is used for PUN queue elements.
4. If the characteristics of a VSE/POWER job change during the time interval for the PUTQE RDR WAIT option, the results are unpredicatble.
5. For PUTQE PUN and PUTQE LST a null line causes end of data.

Operands

RDR
the PUT RDR command generates a job completion message if the job submission was successful. This does not require that the WAIT option is specified. The job completion message is saved in a VSE/POWER message queue which can contain up to 99 messages for each user ID. If you retrieve a job completion message it is erased from the VSE/POWER message queue.

If the VSE/POWER message queue is full the latest message overwrites the oldest message. It is therefore possible that job completion messages generated by PUTQE RDR are lost. If a WAIT option was specified, PUTQE RDR waits until timeout occurs and returns on of the following return codes: -1, -2, -6, or -9. See the 'Examples' for job completion messages of a job transmitted to another VSE node.

LST
PUN
is the name of the VSE/POWER queue into which to place the data.

FORMAT *format*
specifies the type of the record format for all output records. You can specify *format* for the LST operand only. *format* is a value and used as input.

One of the following values is valid:

- MCC
- ASA
- NOCC
- CDPS

If FORMAT NOCC or no FORMAT operand has been specified, the input record consists of the data only. REXX adds the MCC carriage control character X'09'. If a FORMAT operand has been specified, the print record must have a one byte prefix containing the print control character followed by at least one byte containing the data. If the print record is not two bytes long, return code -25 occurs. There is no check if the one byte prefix contains the print control character.

WAIT *time*
specifies a number of seconds to wait for the execution of the job to complete. If *time* is `0`, no wait occurs and no job completion message is available.

Specifying this option causes the program to wait for the job to be executed so that the calling program can examine its results. A job completion message, ARX0970I, may be available through OUTTRAP.

The maximum return code from the job is placed in the REXX special variable RC. RC contains `0` if the job completes with a return code of `0` or if the completed job specifies no return code.

MEMBER *memname*
STEM *stem*
indicates where the file containing the job or output data resides. If you specify MEMBER *memname*, *memname* must be one of the following.

- A fully qualified name in the format library.sublibrary.member.type.

- A member name and member type in the format member.type. To use this form, the type on the LIBDEF statement must be either SOURCE or *. Otherwise, the member is not found, and you receive return code -23.
- The member name only. In this case, the default member type applies; this is PROC.

If you use STEM *stem*, the *stem* must be a valid REXX stem (it must end with a period). The *stem* indicates the compound variables containing the records for the queue entry. *stem*.0 contains a number indicating the number of records. The records are contained in *stem*.1 through *stem.n*. Data records are input to the queue; no checkpoints are taken.

CLASS *classvar*
is a variable to which you have previously assigned a string that is one or more characters representing VSE/POWER classes that exist in your VSE system. The class is whichever of the following is available first.

1. the class from the job stream
2. class A
3. the class you specify

Each successive class is tried until one in which the job can run is found. That class is stored in *classvar* as output. If you omit *classvar*, the job runs only in the class from the job stream. In this case if you have not specified the class in the job stream, the VSE/POWER default class A applies. See note "2" on page 188.

JOBNAME *jnamevar*
specifies the name of a REXX variable in which the job name for the job you created is returned. See note "1" on page 187. Example "5" on page 194 uses this parameter.

JOBNAME *jname*
is the name of the job for which to create data records.

JOBNUM *jnumvar*
specifies the name of a REXX variable. Into this variable, VSE/POWER stores the number of the job or of the output queue entry you created.

JOBNUM *jnum*
is the number VSE/POWER has assigned to the queue entry to which you want to add further data records.

GENCM
NOGENCM
GENCM (which is the default) requests that the VSE/POWER job that had been put into the RDR queue issues a job completion message after end-of-job. NOGENCM causes suppression of the VSE/POWER job completion messages.

LONGREC
indicates that the record length of a POWER RDR queue entry is 128 bytes. If not specified, the default length is 80 bytes.

FIRST
indicates that the given data is only the first portion of the entire data that will make up the output queue entry. More data will later be appended to this first portion. *jnumvar* of JOBNUM must be a variable that contains the job number.

The resulting disposition of the output queue entry is A.

NEXT
indicates that the given data is appended at the end of the specified output queue entry, and that more data can be appended later on. This option is only valid if the output queue entry was created by a previous PUTQE with option FIRST (or NEXT). The resulting disposition of the output queue entry is A.

If you specified the FORMAT operand for PUTQE FIRST, you must repeat it for PUTQE NEXT. Other operands describing output queue entry characteristics (such as USERINFO, COPIES, PRIORITY, DISP, DESTUSER, DESTNODE) are not allowed in this case.

LAST
indicates that the given data is appended at the end of the specified output queue entry, and that more data **cannot** be appended later on. This option is only valid if the output queue entry was created by a previous PUTQE with option FIRST (or NEXT).

If you specified the FORMAT operand for PUTQE FIRST and NEXT, you must repeat it for PUTQE LAST. Other operands describing output queue entry characteristics (such as USERINFO, COPIES, PRIORITY, DISP, DESTUSER, DESTNODE) are not allowed in this case.

JOBSUF *jsuf*
specifies the segment number of the POWER output queue entry where the given data is to be appended.

NEWSEGM *segmvar*
indicates that after appending of the given data to the specified output queue entry a new segment is to be started. *segmvar* specifies a variable used to return the number of the old segment where the given data was appended.

CLASS *class*
is a single character specifying the class of the LST or PUN queue element. If you specify more than one character, only the first character is used. If you omit this option, the VSE/POWER default is used. Operand CLASS is required together with option FIRST, NEXT, and LAST. See note “1” on page 187.

DISP
specifies the desired disposition of the output entry. Valid dispositions are D, K, H, and L.

PRIORITY *prio*
is the desired priority of the output queue entry. *prio* is a number between 0 and 9.

DESTUSER *user*
specifies the name of the destination user.

DESTNODE *node*
specifies the name of the destination node.

COPIES *cop*
is the number of desired copies. *cop* is a number between 1 and 125.

JSEP *jsep_pages*
is the number of desired job separator pages. *jsep_pages* is a number between 0 and 9.

USERINFO *userinf*
specifies user information appearing on job separator pages and in the list or punch account record for the job. *userinf* is a string of up to 16 characters.

Note:

1. As VSE/POWER uses the OR operation with an X'40'value for all characters (not only letters), to perform an uppercase translation, some non-letter characters may change to non-printable characters (see VSE/POWER Administration and Operation, for more details).
2. The USERINFO string is padded with blanks at the end. Use X'00' to specify blanks at the beginning or in the middle of the string. VSE/POWER converts X'00' to X'40'.

OPTB *ostem*
specifies the REXX stem which contains the keyword-value-pairs of the output operands defined by the user. *ostem* must end with a period to be a valid REXX stem. *ostem.0* contains a number indicating the number of keywords. *stem.1* through *stem.n* contain the keyword-value-pairs. To be able to use OPTB, the automatic startup of VSE/POWER requires the corresponding definitions of additional JECL output operands (see the description of the DEFINE statement in the *VSE/POWER Administration and Operation*, SC33-6733, manual). To pass a user keyword and its values to VSE/POWER, assign a string KEYWORD={value|(value,...)} to *ostem.n*, for example, OPTBSTEM.1 = 'PAGEDEF=HUGO'. VSE/POWER matches the received keyword with the specifications of the corresponding DEFINE autostart statement. If two or more keyword OPTBs specify the same user keyword, only the last specification becomes effective. It is also possible to specify the keyword id instead of the keyword for a user-defined output operand, for example OPTBSTEM.1 = '001F=HUGO'. In this case all following output operand specifications in *ostem* must use this keyword ID description and not the keyword.

FCB *phasename*
specifies the name of the FCB-image phase VSE/POWER is to use for printing the related job output.

UCS *phasenam*
UCS *(phasenam,<op>)*
specifies the name of the UCB-image phase loaded into the UCB (universal character set buffer) of the printer. The following options *<op>* are available:

F
to indicate that the UCB is to be loaded with the folding operation code causing lowercase letters to be printed in uppercase.

C
to prevent data checks from being generated because of print-line mismatches with the UCB.

FLASH_COUNT *fnum*
This operand applies to IBM 3800 only. *fnum* is a number from 0 to 255. It specifies the number of copies to be flashed with the overlay. If you specify a count without a FLASH_NAME, the forms-overlay frame loaded at the time of printing is used. If you specify a count of 0, then the operator is prompted to load the requested forms-overlay frame, but the overlay is not flashed.

FLASH_NAME *fnam*
This operand applies to IBM 3800 only. *fnam* is the one-to-four-character name of the forms-overlay frame to be used by the printer. If you specify an overlay name without a count, all copies are flashed.

SYSID *n*
This operand applies to shared spooling. *n* is either the character N or a number between 1 and 9. Specify SYSID N if the output is to be available on any of the sharing systems. Specify a digit between 1 and 9, if your output is to be available on a certain one of your sharing systems. For *n*, give the number with which the systems VSE/POWER was initialized (by SYSID=n in the VSE/POWER generation macro).

FNO *formnam*
specifies the form name. *formnam* consists of one to four alphameric characters.

When VSE/POWER is to place a job on the RDR queue or create an output queue entry, the user ID for this entry is determined according to the rules described for function USERID on page “USERID” on page 92. A password for output queue entries is the one last specified in a SETUID command (see hä“SETUID” on page 165).

When the VSE/POWER spool-access services interface encounters an error, the REXX special variable RC is set to -13. Error information is written to the current output stream. You can use ASSGN(STDOUT) to return the name of the current output stream.

If the return code from PUTQE is greater than -11, *classvar*, *jnamevar*, and *jnumvar* contain respectively the VSE/POWER class, job name, and job number of the job that was put into the reader queue.

If *classvar* includes incorrect VSE/POWER classes, for example, % or ' ', return code -13 is set.

The following table shows how PUTQE sets the REXX special variable RC. Return codes -1 through -12 and *n* occur only for the RDR queue.

Return Codes:

Return Code	Meaning
0	Successful processing.
-1	Job has been started. Timeout occurred; no job completion message has been retrieved.
-2	Job has been started. VSE/POWER Get Completion Message service (GCM service) is not active. No message can be retrieved.
-4	Job has been started but has been canceled.

Return Code	Meaning
-5	Job has been submitted but cannot be started. The job is in a hold state, that is, DISP=L or H, or the time event scheduling has not expired.
-6	A timeout occurred. The job has been submitted, but was not found in the RDR queue and a job completion message could not be retrieved. It is not known whether the job has been started or completed.
-7	Job has been submitted but not started because the specified class is busy. (This is returned only if you do not specify *class*.)
-8	Job has been submitted but not started because *class* is not defined or is disabled. (This is returned only if you do not specify *class*.)
-9	Job has been submitted. It was not found in the RDR queue and a job completion message could not be retrieved because VSE/POWER Get Completion Message (GCM) service is not active. It is not known whether the job has been started or completed.
-10	Job has been submitted but not started because of a scheduling error. Possible reasons are. • The class or classes are busy • The class or classes are disabled • The class or classes are not defined • A job entry is not found in the RDR queue. (This return code is issued only if *class* had at least one character.)
-11	An error occurred while submitting the job. Information messages are retrieved. Possible reason is: Power * $$ JOB JECL statements include invalid parameters (DISP, PRI, CLASS etc. values specified may be incorrect).
-12	An error occurred while submitting the job. No information message is available.
-13	A severe XPCC or VSE/POWER error or incorrect class specification.
-14	General use storage could not be obtained.
-15	No job statements are available.
-16	A storage problem occurred during the set up to get a connection to the VSE/POWER spool-access services interface.
-17	Connection to VSE/POWER spool-access services failed.
-18	Record is larger than 32K
-19	Error in an operand. Possible reasons are. • The job name you specified may be longer than a valid VSE/POWER job name • The job number may be larger than a valid VSE/POWER job number • A required operand is missing • A keyword is misspelled.
-20	Error in an internal call to ARXEXCOM to fetch or set a variable name. This could be because the variable name did not follow the rules for naming variables. (See “Tokens” on page 9 and “Compound Symbols” on page 20.)
-21	Error from MEMBER.
-22	The stem was not valid

Return Code	Meaning
-23	The member was not found
-24	The member name was not valid
-25	Invalid length of input record
-27	Operand NOGENCM is specified together with a WAIT time greater than zero.
-28	One of the append keywords FIRST, NEXT, or LAST is specified and at least one of the following conditions is true: • The POWER queue is neither LST nor PUN. • The operands JOBNAME, JOBNUM, or CLASS, which must be specified when appending DATA to an output queue, are missing. • Operand DESTNODE is specified. This is invalid as it is not possible to append anything to an entry in the POWER XMT queue. • The operands COPIES, PRIORITY, DISP, FORMAT, USERINFO, FCB, or DESTUSER are specified, which is invalid together with the kewords NEXT and LAST.
-29	No data is specified together with NEWSEGM. This does not work.
-30	The stem variable for ADDRESS POWER PUTQE RDR contains empty lines.
-31	FLASH_COUNT or FLASH_NAME invalid.
-34	Operand SYSID is invalid.
-35	Operand FNO is invalid.
-36	Operand UCS is invalid.
n	This is the maximum return code from the job.

Note: OUTTRAP (page "OUTTRAP" on page 94) can trap error information from PUTQE. If trapping is active, error information is also written to the stem that you specify on the OUTTRAP function. This error information can include VSE/POWER messages. See z/VSE Messages and Codes, for descriptions of VSE/POWER messages. ***Examples***

1. This example places the sublibrary member `MYJOB.PROC` on the RDR queue. (Note that REXX/VSE supplies the default member type, `PROC`.)

```
ADDRESS POWER "PUTQE RDR MEMBER MYJOB"
```

2. This example does the same, but the *memname* includes the library.sublibrary specification.

```
ADDRESS POWER "PUTQE RDR MEMBER MYLIB.MYSUB.MYJOB.PROC"
```

3. This example includes the CLASS option for a job you are putting on the RDR queue. Assume that the original class is B and that classes A and B are busy, disabled, or not defined but that class D can be used. First use an assignment statement to initialize a variable to the class or classes you want to specify on PUTQE. Then specify the name of this variable after the keyword CLASS.

```
myclass="AD9"
ADDRESS POWER "PUTQE RDR MEMBER MYPROG.PROC WAIT 5 CLASS myclass"
```

   REXX puts the job on the RDR queue with class D and returns D in the variable `myclass`.

4. This example includes the CLASS option for a job you are putting on a LST queue. After the keyword CLASS, you specify a single character that is the class of the LST queue entry.

```
ADDRESS POWER "PUTQE LST MEMBER MYPROG.PROC CLASS A"
```

5. This example shows the result when you use the *jnamevar*. Assume MYPROG.PROC contains the following JCL.

```
* $$ JOB JNM=GOODJOB,DISP=D,PRI=3,CLASS=B
// JOB LIZDIR EXECUTE PROGRAM LISTDIR
// LIBDEF PHASE,SEARCH=IJSYSRS.SYSLIB
// EXEC LIBR,SIZE=(AUTO,64K)
LISTDIR LIB=LIZH
/*
/&
* $$ EOJ
```

   If you use the following PUTQE command.

```
ADDRESS POWER "PUTQE RDR MEMBER MYPROG.PROC JOBNAME mine CLASS class"
```

   REXX puts GOODJOB into the variable `mine` and the class B into the variable `class`, and the entry on the RDR queue has the job name GOODJOB.

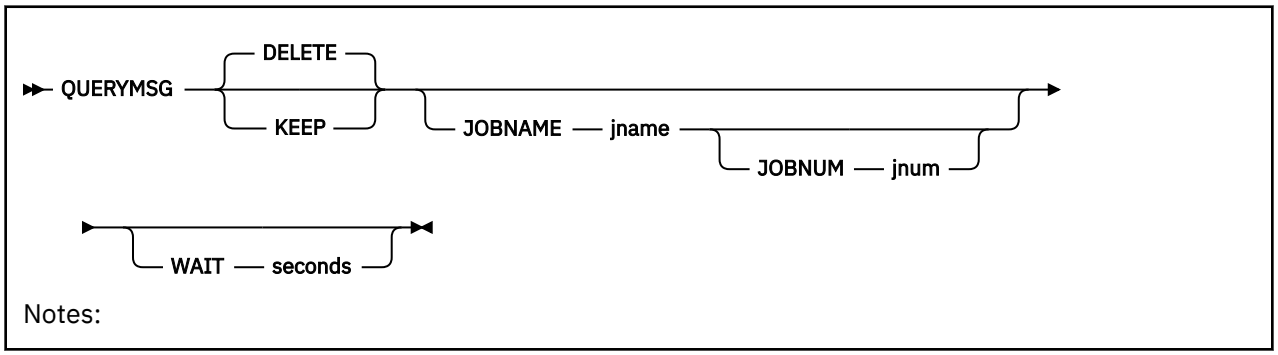
6. This example includes the OPTB option.

```
ostem.0 = 2
ostem.1 = 'PAGEDEF=HUGO'
ostem.2 = 'CICSDATA=xyz'
ostem.3 = '0021=(PPP,QQQ,RR,SS)'
ADDRESS POWER "PUTQE LST STEM ltfile. OPTB ostem."
```

7. This example creates a LST queue entry consisting of 2 segments in 3 steps.

```
ADDRESS POWER
  'PUTQE LST JOBNAME PUTQTEST JOBNUM jn FIRST' ,
  'CLASS Q STEM file1.'
  'PUTQE LST JOBNAME PUTQTEST JOBNUM' jn 'NEXT' ,
  'CLASS Q STEM file2. NEWSEGM segment_var'
  jobsuf_num = segment_var + 1
  /* start of new segment */
  'PUTQE LST JOBNAME PUTQTEST JOBNUM' jn ' LAST' ,
  'CLASS Q STEM file3. JOBSUF ' jobsuf_num
```

QUERYMSG



Notes:

QUERYMSG returns job completion message(s) into the stem specified by OUTTRAP. *Seconds*, *jname*, and *jnum* serve as input parameters only. The output reflects the specified search arguments. Search arguments can be Userid as given by the USERID() function, JOBNAME *jname*, and JOBNUM *jnum*.

Note:

1. QUERYMSG only retrieves job completion messages of jobs submitted to VSE/POWER via PUTQE RDR.
2. QUERYMSG retrieves all messages available at the time the command is issued. The maximum value of the OUTTRAP function does not limit the number of messages deleted by QUERYMSG DELETE. It only limits the number of messages stored into the OUTTRAP stem.

Operands

DELETE
specifies that the messages are deleted after retrieval. This is the default.

KEEP
the messages are kept in the VSE/POWER message queue and can be retrieved again.

JOBNAME *jname*
the name of the job you want completion messages from.

JOBNUM *jnum*
the number of the job you want completion messages from. If you specify JOBNUM, JOBNAME must be specified, too.

WAIT
specifies the maximum amount of time in seconds until the messages are returned.

The following table shows how QUERYMSG sets the REXX special variable RC.

Return Code	Meaning
0	Messages are available.
-1	Timeout occurred and/or no job completion message has been retrieved.
-2	VSE/POWER Get completion Message service (GCM) is not available.
-13	VSE/POWER error.
-19	Error in an operand.

QUERYMSG Examples:

1. This example returns all job completion messages for USERID() and deletes them from the VSE/POWER message queue.

```
QUERYMSG
```

2. This example returns all job completion messages for USERID() and leaves them in the VSE/POWER message queue.

```
QUERYMSG KEEP
```

3. This example returns all job completion messages for USERID() and JOBNAME *jname* and leaves them in the VSE/POWER message queue.

```
QUERYMSG KEEP JOBNAME jname
```

4. This example returns all job completion messages for USERID(), JOBNAME *jname*, and JOBNUM *jnum*, and leaves them in the VSE/POWER message queue.

```
QUERYMSG KEEP JOBNAME jname JOBNUM jnum
```

5. This example shows job completion messages of a job transmitted to another VSE node. For example, job TSTSUB3 was submitted to VSE node BOEVSE02 via PUTQE and the variable *jnumvar* returned the value '05097'.

```
ADDRESS POWER "PUTQE RDR MEMBER PRD2.REXX.TESTSUB3.Z" ,
              " WAIT 20 CLASS class JOBNAME jobname JOBNUM jobnum"
```

   Executing this job on the VSE node BOEVSE03 leads to the following job completion message:

```
ARX0970I JOB TESTSUB3 00346 EXECUTED NODE=BOEVSE03 DATE=08/12/94
         TIME=7:31:20 MAXRC=0008 LASTRC=0008 ORG=05097
```

ORG=05097 specifies the job number on the original node BOEVSE02 and 00346 is the job number on the executing node BOEVSE03.

However, if the job is executed on the same node BOEVSE02, it generates the following job completion message:

```
ARX0970I JOB TESTSUB3 05097 EXECUTED NODE=BOEVSE02 DATE=08/12/94
         TIME=7:31:20 MAXRC=0008 LASTRC=0008
```

Rules for Issuing Job Completion Messages

There are two cases where job completion messages are generated:

- ADDRESS POWER PRELEASE The message generated by this command is called the message for the releaser.
- ADDRESS POWER PUTQE RDR The message generated by this command is called the message for the submitter.

The following applies:

1. The releaser gets a completion message if the PRELEASE command has been successfully processed. If more than one PRELEASE command has been issued, the releaser gets the completion message of the last successfully processed command.
2. If the PRELEASE command does not achieve a successfully starting of the mentioned POWER job, message 1R88I NOTHING TO RELEASE is issued. There will be no completion message in this case.
3. If releaser and submitter are identical only one completion message is issued. Otherwise two completion messages are issued. The releaser and submitter are identical if they use the same partition ID, REXX user ID, node ID, and system ID.
4. There will be no job completion message if
   a. a job is being executed a second time without having been released by REXX. This can be a job which is scheduled to run repeatedly because of time scheduling operands (DUEDAY, for example).
   b. the POFFLOAD command has been used to write the job to tape
   c. a child job has been created by a parent job via the DISP=I operand within the * $$ PUN statement.
5. In a shared environment a completion message for the releaser is routed back to the system on which the PRELEASE had been issued.
6. In a network a completion message for the releaser is routed back to the node on which the PRELEASE had been issued.
7. The original jobnumber of a message for the submitter may not be same as the original jobnumber for the releaser, if the job has been submitted on a node A, sent to a node B, and has then been released on node B. The original jobnumber of a message for a submitter is the jobnumber on node A where the job has been submitted. The original jobnumber of a message for a releaser is the jobnumber on node B where the job has been released.

The *VSE/POWER Diagnosis Reference*, LY33-9163, describes how VSE/POWER generates completion messages caused by a PRELEASE command.

CTL

The CTL (control) service is part of the VSE/POWER access services. Through a CTL service request you can send commands to VSE/POWER. Use ADDRESS POWER to send the following commands:

PALTER
Alter attributes of a queue entry

[4] These commands are for authorized users only, for example a REXX procedure specifying the POWER master password in the SETUID command.

PBRDCST
Transmit a message

PCANCEL
Cancel a job that is being executed

PDELETE
Delete a reader or an output queue entry

PDISPLAY
Display status information about a reader or an output queue entry or a group of entries

PFLUSH DEV[4]
Cancel device

PGO[4]
Reactivate a task or partition

PHOLD
Place a reader or an output queue entry into the hold status

PINQUIRE
Display various kinds of job- and resource-status information, status information about queue entries on tape or network related information, status information of the active Dynamic Class Table.

PLOAD DYNC[4]
Load the Dynamic Class Table

PRELEASE
Release a job or an output queue entry. If you are not interested in POWER-generated job completion messages, add string **(NOGENCM** to your PRELEASE RDR command (for example: ADDRESS POWER "R RDR,MYJOB (NOGENCM")

PSEGMENT
Segment an output queue entry

PSETUP
Print the page layout of one or more pages

PSTART[4]
Start a device driving system like PSF, LANRES, or CICS Report Controller

PSTOP[4]
Stop a device driving system

PVARY DYNC[4]
Dis/enable dynamic classes.

PXMIT
Pass a command for processing by VSE/POWER to another node.

For the commands PALTER, PCANCEL, PDELETE, PHOLD, and PRELEASE the POWER access rules apply as described on “Accessing Entries in VSE/POWER Queues” on page 181.

The following table shows how a CTL service request sets the REXX special variable RC.

Return Code	Meaning
0	Successful processing.
-13	A severe XPCC or POWER error.
-14	General use storage could not be obtained.
-16	A storage problem occurred during the set up to get a connection to the VSE/POWER spool-access services interface.
-17	Connection to VSE/POWER spool-access services failed.

Note: OUTTRAP (page "OUTTRAP" on page 94) can trap error information from CTL. If trapping is active, error information is also written to the stem that you specify on the OUTTRAP function. This error information can include VSE/POWER messages. See z/VSE Messages and Codes, for descriptions of VSE/POWER messages.

The following shows an example of a CTL service request:

```
jobname = MYJOB
'SETUID PAUL'                           /* POWER FROM/TO user */
oldtrap = OUTTRAP(pwr_ret.)             /* POWER return info  */
ADDRESS POWER
'PDISPLAY LST,'|| jobname ||',CCLASS=X'  /* POWER command      */
If RC = 0 Then
 Do i=1 To pwr_ret.0
  Say pwr_ret.i
 End
```

Submitting and Controlling Power Jobs

REXX offers various types of job management:

1. **Submitting VSE/POWER jobs**: A job taken from a VSE library or REXX stem is placed into the VSE/POWER RDR queue via PUTQE RDR command. Here is an example:

   ```
   ADDRESS POWER "PUTQE RDR MEMBER MYJOB"
   ```

   PUTQE does not wait and REXX continues with the next instruction. PUTQE has no information if the job has started or completed. If the job has completed, VSE/POWER generates a job completion message .

2. **Waiting for synchronously running jobs**: A VSE/POWER job is scheduled in a different partition via PUTQE RDR WAIT. The REXX program waits and continues execution if one of the following occurs:

   - the job has completed
   - a scheduling error has occurred
   - the time has expired

   Here is an example:

   ```
   /*****************************************************************/
   /*                                                               */
   /*   The REXX program is submitting a job and awaiting its       */
   /*   execution.                                                  */
   /*   The job resides in the library member A.B.MYJOB.JCL         */
   /*   and runs an utility program. It's output is retrieved.      */
   /*   Subroutine CHECK_JOB_OUTPUT scans the job output for        */
   /*   the argument NAME. The class of the LST queue entry         */
   /*   will be the default class.                                  */
   /*                                                               */
   /*****************************************************************/
   ARG name .
   CALL OUTTRAP out.
   ADDRESS POWER
   class = 'ABC'
   'PUTQE RDR WAIT 60 MEMBER A.B.MYJOB.JCL',
             'JOBNAME jobname JOBNUM jobnum CLASS class'
   IF rc = 0
      THEN DO
         'GETQE LST STEM job_output.' ,
               'JOBNAME' jobname 'JOBNUM' jobnum
         IF Check_Job_Output(name)
               THEN SAY 'We found the' name
         END
      ELSE IF rc > 0
           THEN SAY 'Return code of MYJOB is:' rc
           ELSE SAY 'Job submission failed'
   EXIT
   /*  Check_Job_Output: Check each line of the job output for the */
   /*  blank delimited word passed via the first argument.         */
   /*  Return 1 if found,  otherwise return 0.                     */
   Check_Job_Output:
          arg looking_for
   ```

```
       do line = 1 to job_output.0
         if wordpos(looking_for,translate(job_output.line))¬=0
           then return 1
       end
       return 0
```

See also the demo program REXXWAIT described on page “REXXWAIT” on page 263.

3. **Managing asychchronously running jobs**: REXX is assumed to be running in partition F5. Several VSE/POWER jobs submitted or released by REXX are running in different partitions. After the jobs have ended, VSE/POWER generates job completion messages held in the message repositories F5.USER_1 and F5.USER_2. If at least one job completion message is available, QUERYMSG retrieves job completion messages.

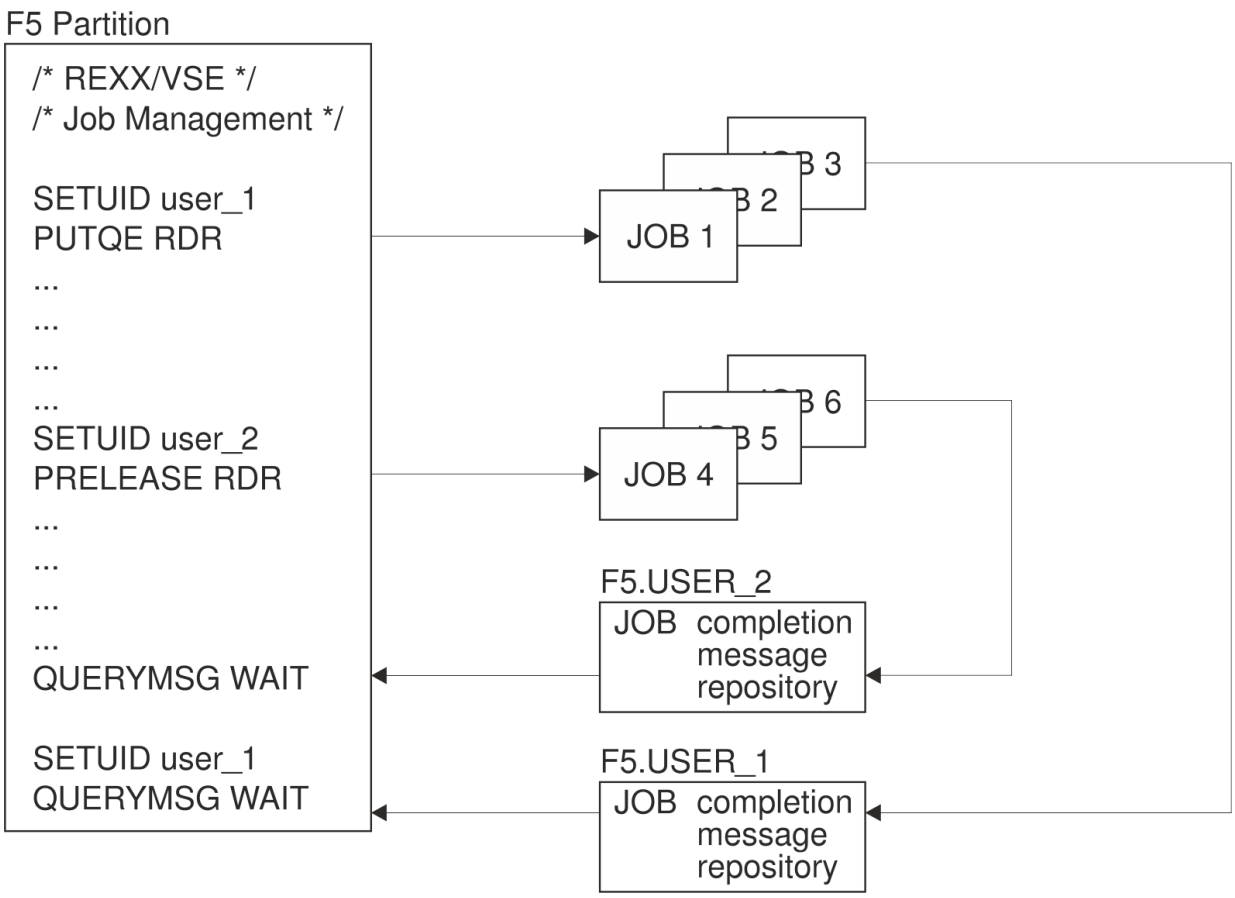


Figure 6. Job Management Using the QUERYMSG Function

Here is an example: Note that for simplicity only one job completion message repository and the job naming convention MYJOBnnn is assumed here.

```
/*****************************************************************/
/*   REXX code that shows the management of asynchronously       */
/*   running jobs.  VSE/POWER jobs with name MYJOBnnn,           */
/*   nnn=1,2,3,.., are submitted.                                */
/*****************************************************************/
ARG number_of_jobs
SETUID 'USER'
CALL OUTTRAP msg.,,NOCONCAT
ADDRESS POWER
DO i=1 TO number_of_jobs
   'PUTQE RDR MEMBER A.B.MYJOB'i'.JCL'
END
DO WHILE number_of_jobs ¬= 0
   CALL SLEEP 5              /* Wait  for messages */
   'QUERYMSG KEEP'        /* Query for messages */
   IF rc=0 THEN             /* Check completion  */
      DO i=1 TO msg.0 while number_of_jobs ¬= 0
```

```
         IF WORD(msg.i,1) = 'ARX0970I' THEN DO
         PARSE VAR msg.i . . jname jnumber . . . . maxrc .
         IF SUBSTR(jname,1,5) = 'MYJOB' THEN DO
            'QUERYMSG DELETE JOBNAME' jname ,
                           'JOBNUM' jnumber
            IF maxrc = 'MAXRC=0000'
               THEN SAY 'JOB' jname 'run successfully'
               ELSE SAY 'JOB' jname 'failed with' maxrc
            number_of_jobs = number_of_jobs -1
            END
         END
      END
END
```

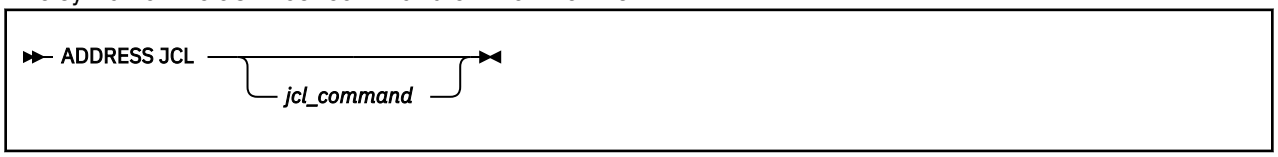
See also the demo program REXXJMGR described on page “REXXJMGR” on page 262.

Chapter 12. JCL Command Environment

The JCL Host Command Environment

The JCL command environment lets you isssue JCL commands via a REXX program.

The syntax of the JCL host command environment is



Note: Address JCL can only be used if the REXX program has been invoked by // EXEC REXX.

In case you only want to enter one JCL command, you can use this format:

```
ADDRESS JCL "jcl_command"
```

If you want to issue several JCL commands, you can use the following format:

```
ADDRESS JCL
"jcl_command_1"
"jcl_command_2"
:
"jcl_command_n"
```

In case a JCL command requires SYSIPT data as, for example, SET SDL, use the REXXIPT function. With REXXIPT you determine the name of the stem used to specify the SYSIPT input data.

The following example loads all required REXX/VSE phases into the SVA:

```
ADDRESS JCL
CALL REXXIPT input.

input.0  = 4
input.1  = 'USERPGM1,SVA'
input.2  = 'USERPGM2,SVA'
input.3  = 'USERPGM3,SVA'
input.4  = '/*'

'SET SDL'
```

You can keep the input_stem and at the same time control whether a JCL command reads data from REXXIPT:

```
input.0 = 0
'SET SDL'                   /* <= Does not read from input stem    */
                            /*    No phases are loaded into SVA    */
input.0 = 1
'SET SDL'                   /* <= Reads one record from input stem */
                            /*    Phase ARXINIT is loaded into SVA */
```

If you issue JCL commands which do not read data from SYSIPT, either do not specify a call to REXXIPT before the ADDRESS JCL command, or make sure that input stem.0 is set to 0.

The following example shows a REXX program which defines five DLBL/EXTENTs for VSAM files and writes data from the program stack on a file:

```
ADDRESS JCL
DO i=1 TO 5
"// DLBL RXTEST"|| i ||",'REXXVSE.TEST."|| i ||"',,VSAM"
"// EXTENT ,CTS220"
END
"/*"               /* Gives an indication for JCL that the       */
```

```
                    /* last label (RXTEST5) together with its     */
                    /* extents can be written into the label area */
```

Format of Address JCL Commands

ADDRESS JCL can issue a JCL command of up to 407 characters. REXX/VSE breaks the command string into appropriate pieces to match the JCL rules. ADDRESS JCL supports up to 6 continuation lines. No continuation character is required and the continuation line need not start at a specified column. The following example shows a long LIBDEF chain which does not fit into one JCL line:

```
ADDRESS JCL

user_lib   = 'USERLIB.SUB1,USERLIB.SUB2,USERLIB.SUB3'
system_lib = 'PRD2.PROD,PRD2.BASE,IJSYSRS.SYSLIB'

'LIBDEF *,SEARCH=('||user_lib||system_lib||')'
```

VSE JCL ON Conditions

VSE JCL deactivates ON $RC conditions set by job control during // EXEC REXX processing. After the REXX exec has finished, ON $RC is reset to active. Any non zero return code from the REXX exec sets the $RC condition to ON. The ON $ABEND and ON $CANCEL conditions remain active. These conditions cannot be set within the REXX exec. Options ACANCEL, JCANCEL, and SCANCEL are deactivated as long as REXX is running. When REXX has ended they are set to ACTIVE again.

Unsupported JCL Commands

ADDRESS JCL does not support the following JCL commands:

"/."
"/+"
"/&"
"ALLOC"
"EXEC"[5]
"GOTO"
"IF"
"UNBATCH"
"JOB"
"ON"
"OVEND"
"PAUSE"
"PROC"
"ROD"
"RSTRT"

Invoking these commands results in return code -7.

VSE JCL Output Trapping

VSE JCL traps all information and decision messages normally written to SYSLOG. Use the OUTTRAP function to retrieve these messages. Note that these messages are not shown at the operator's console. An exception are messages causing a job cancellation. The job is cancelled and JCL does not return to REXX.

Messages written to SYSLST and action messages written to SYSLOG are not trapped.

Return codes from the JCL Host Command Environment

[5] Instead EXEC PGM,PARM you may use ADDRESS LINK. See “The LINK Host Command Environment” on page 206

Return Code	Meaning
0	JCL command completes with return code 0 or with no return code (a JCL command returns either 0 or no return code).
-1	The REXXIPT input stem specifies an invalid number of input records.
-2	Error from ARXEXCOM while working with the REXXIPT input stem.
-3	JCL command not found.
-4	JCL command processing failed, an appropriate VSE error message may be retrieved via the OUTTRAP function.
-5	A Rexx program issues ADDRESS JCL but was not invoked by '// EXEC REXX=' or does not run under the main task.
-6	No more GETVIS storage available by ADDRESS JCL
-7	Restricted JCL command, not allowed by ADDRESS JCL.
-8	Error while ARXOUT was opened to accept OUTTRAP data records.

Chapter 13. Host Command Environments for Loading and Calling Programs

Host Commands

REXX/VSE provides the LINK and LINKPGM host command environments to let you load and call a phase from the active PHASE search chain.

To load and call a program, specify the name of the program followed by any parameters you want to pass to the program. For example:

```
ADDRESS LINKPGM "PROGRAM p1 p2 ... pn"
```

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

The LINK and LINKPGM environments all support programs of any AMODE or RMODE.

These environments differ in:

- the format of the parameter list that the program receives
- the capability of passing multiple parameters
- variable substitution for the parameters
- the ability of the called program to update the parameters.

The LINK environment offers an alternative to the job control statement EXEC PGM. The parameter list is the same as if you specify the PARM parameter in the EXEC PGM statement. For details see “The LINK Host Command Environment” on page 206 and “The LINKPGM Host Command Environment” on page 208

For the LINK environment, you can specify only a single character string to pass to the program. The LINK environment does not evaluate the character string and does not perform variable substitution. It simply passes the string to the called program. The program can use the character string it receives. However, the program cannot return an updated string to the REXX program.

For the LINKPGM environment, you can pass multiple parameters to the called program. The environment performs variable substitution on the parameters you specify. That is, the environment determines the value of each variable. When the environment calls a program, it passes the value of each variable to the program. The program can update the parameters it receives and return the updated values to the REXX program.

If you want to have a called program read input from REXX compound variables instead of reading from SYSIPT, use the REXXIPT external function. You specify a stem on the REXXIPT external function and call REXXIPT before calling the program. See “REXXIPT” on page 98 for details.

The table of authorized programs (ARXEOJTB) allows programs using the EOJ macro to return to REXX rather than terminating the jobstep.

After you load and call a program, the host command environment sets a return code in the REXX special variable RC. For the LINK and LINKPGM environments, the return code can be -3 if the host command environment could not locate the program you specified.

For the LINK and LINKPGM environments, the return code set in RC can also be -2. For the LINKPGM environment, this indicates unsuccessful processing of variables. This may have been because the host command environment could not:

- Perform variable substitution before loading and calling the program
- Update the variables after the program completed.

For the LINK environment, you can also receive an RC value of -2 if the length of the value of the parameter you pass is larger than the length that can be specified in the signed halfword length field in the parameter list. The maximum value of the halfword length field is 32,767. On exit from the called program, register 15 contains the return code from this program.

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

The following topics describe how to load and call programs using these host command environments.

The LINK Host Command Environment

The LINK environment lets you load and call a non-REXX program in the same partition under the same task where the REXX program is running. For the LINK environment, you can pass only a single character string to the program. The LINK host command environment calls programs with the same parameter list convention as the JCL parameter list:

```
// EXEC PGM=pgmname,PARM='character_string'
```

When you use the LINK environment, enclose the name of the program and the character string in single or double quotation marks. This prevents the language processor from performing variable substitution. Here are two examples:

```
ADDRESS LINK 'TESTPGMA varid'
ADDRESS LINK 'TESTMODA this is a parameter string'
```

If you want to pass the value of a variable, do not enclose it in quotation marks. In this case, the language processor performs the variable substitution before passing the string to the host command environment. The following excerpts from a REXX program would have the same results as the previous examples:

```
parm_value = 'varid'
ADDRESS LINK 'TESTPGMA' parm_value

parm_value = 'this is a parameter string'
ADDRESS LINK 'TESTMODA' parm_value
```

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

Figure 7 on page 207 shows how the LINK host command environment routine passes a character string to a program. Register 0 points to the ENVBLOCK under which the REXX program issuing the ADDRESS LINK is running. Register 1 points to a parameter consisting of:

- A length (of a character string). This is a halfword.
- A character string.

If you specify no parameters when loading and calling a program with ADDRESS LINK, then register 1 contains the same value as register 15. If you specify parameters, the high-order bit of the parameter register 1 points to is on. The halfword length field contains the length of the parameter you pass. The maximum value of the halfword length field is 32,767.

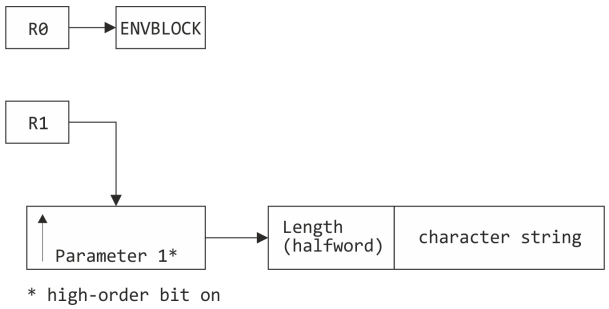


Figure 7. Parameters for the LINK Environment

For example, suppose you use the following instruction:

```
ADDRESS LINK 'TESTMODA numberid payid'
```

When the LINK host command environment routine loads and calls the TESTMODA 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` are REXX variables, the LINK host command environment performs no substitution.

You can use the LINK environment without specifying a character string. For example:

```
ADDRESS LINK "PROGA"
```

Return Codes from the LINK Environment

On return from the called program the contents of Register 15 is stored into the REXX special variable. The following table lists return codes from the LINK environment that are stored in the REXX special variable RC.

Return Code	Meaning
-1	The contents of *stem*.0 is not a positive number or 0. (Calling the REXXIPT function specifies a stem name; *stem*.0 specifies the number of data records available for reading, and this number must be positive or 0.)
-2	The parameter length exceeded 32,767. See “Host Commands” on page 205 for details.
-3	The host command environment could not find the program you specified. You may get message ARX0565I showing the failing return code of the LOAD or CDLOAD macro.
-4	The host command environment could not find the phase ARXEOJTB or the phase resides in the SVA.
-5	Usage of this program is restricted to the main task only.
-6	Not enough partition GETVIS storage available to successfully process the command.

Return Code	Meaning
-8	One of the following: • REXX tried to open ARXOUT to accept OUTTRAP data records before it gives control to the program specified in ADDRESS LINK. This open failed. • There are problems filling the OUTTRAP variable during an invocation of ADDRESS LINK LIBR. Check if your startup procedure contains the statement "// EXEC ARXLINK" and add it. This statement is contained in the startup procedure USRBG.PROC in library IJSYSRS.SYSLIB and in the skeleton SKUSERBG in the ICCF library 59. See also the description of SKUSERBG in the *VSE/ESA Planning* manual. • Insufficient storage.
-9	This occurs if you call LNKEDT for one of the following reasons: • SYSLNK was not opened. The invocation of LNKEDT is out of sequence because the VSE JCL statement CATAL or LINK did not precede the LNKEDT call. • Error during the attempt to write end-of-file marker on SYSLNK.
-10	The user program was not authorized to issue SVC 14 (EOJ macro) when it was called by ADDRESS LINK. See also message ARX0980E.
-11	Program does not fit into program area (SIZE too small).

The LINKPGM Host Command Environment

The LINKPGM environment lets you load and call a non-REXX program in the same partition under the same task where the REXX program is running. Using the LINKPGM environment, you can pass multiple parameters to the program. The parameters do not have a length field. Upon return from the called program, the value of the passed parameters are updated, and the length of each parameter is the same as when the parameter list was created. To use the LINKPGM environment, specify the name of the program followed by variable names for each of the parameters. Separate the variable names with 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 single or double quotation marks. When you use the quotation marks, the language processor does not evaluate any variables. It simply passes the expression to the host command environment for processing. The LINKPGM 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 environment routine evaluates the value of each variable, it builds a parameter list pointing to the values. The routine then loads and calls the program and passes the parameter list to the program.

Figure 8 on page 209 shows how the LINKPGM host command environment routine passes the parameters to the program. Register 0 points to the ENVBLOCK under which the REXX program issuing the ADDRESS LINKPGM 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 1 to indicate the end of the parameter list.

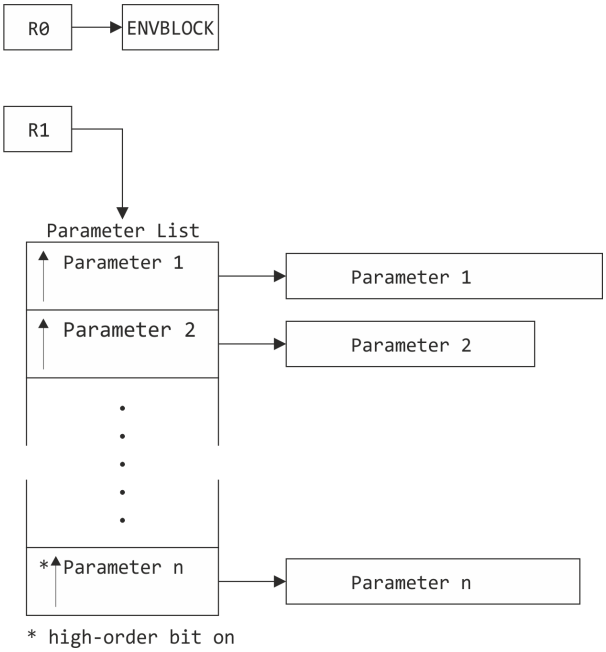


Figure 8. Parameters for the LINKPGM Environment

On output from the called 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 called 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 load and call a program.

The following example loads and calls the phase TESTMODA, passing the parameters 123456 and ABCdef. Before using ADDRESS LINKPGM, assign the values to variables. Suppose you expect the called program to pass back values with a length that is greater than 6 (for example, 20). You can pad the parameter with blanks on the right before passing it. One way to do this is to use the LEFT built-in function in the assignment statement.

```
var1=left('123456',20)
var2=left('ABCdef',20)

ADDRESS LINKPGM 'TESTMODA var1 var2'
```

Here is another example. Suppose you want to load and call to the RESLINE program, passing 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. Assign the value to *revcode* before using ADDRESS LINKPGM:

```
/*  REXX program that loads and calls RESLINE program    */
:
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 program, the RESLINE program could not return a value that is greater than 8 bytes. To let the called program return a larger value, you could pad the value of the original variable to the right with blanks. For example, in the REXX program you could add seven blanks

and assign the value "WK007816 " to the *revcode* variable. The length would then be 15 and the called program could return an updated value of up to 15 bytes.

You can use the LINKPGM environment and not specify any parameters. For example:

```
ADDRESS LINKPGM "MONBILL"
```

If you do not specify any parameters, then register 1 is equal to register 15.

Return Codes from the LINKPGM Environment

On return from called program the contents of Register 15 is stored into the REXX special variable. The following table lists return codes from the and LINKPGM environment that are stored in the REXX special variable RC.

Return Code	Meaning
-1	The contents of *stem*.0 is not a positive number or 0. (Calling the REXXIPT function specifies a stem name; *stem*.0 specifies the number of data records available for reading, and this number must be positive or 0.)
-2	Processing the variables of LINKPGM or REXXIPT was not successful. See “Host Commands” on page 205 for details.
-3	The host command environment could not find the program you specified. You may get message ARX0565I showing the failing return code of the LOAD or CDLOAD macro.
-4	The host command environment could not find the phase ARXEOJTB or the phase resides in the SVA.
-5	Usage of this program is restricted to the maintask only.
-6	Not enough partition GETVIS storage available to successfully process the command.
-8	REXX tried to open ARXOUT to accept OUTTRAP data records before it gives control to the program specified in ADDRESS LINKPGM. This open failed.
-9	This occurs if you call LNKEDT for one of the following reasons: • SYSLNK was not opened. The invocation of LNKEDT is out of sequence because the VSE JCL statement CATAL or LINK did not precede the LNKEDT call. • Error during the attempt to write end-of-file marker on SYSLNK.
-10	The user program was not authorized to issue SVC 14 (EOJ macro) when it was called by ADDRESS LINKPGM. See also message ARX0980E.
-11	Program does not fit into program area (SIZE too small)

Table of Authorized Programs

If you call a program using the EOJ macro by ADDRESS LINK or ADDRESS LINKPGM, you have to code an entry in the table of authorized programs ARXEOJTB if you want the EOJ macro to return to the calling REXX program. It also ensures that the programs having an entry in the ARXEOJTB table can only be invoked from a REXX program running under the main task.

If a user program uses the EOJ macro, and no entry exists in table ARXEOJTB, message ARX0980E is issued and the REXX program is terminated.

IMPORTANT

1. If an entry exists in table ARXEOJTB, the EOJ macro (SVC 14) does not end the jobstep and does not clean up any task related resources. It is your responsibility to make sure that your program does not hold any system resources, e.g. LOCK or GETVIS.
2. The completion of the // EXEC REXX jobstep cleans up task related resources.

PRD1.BASE contains a sample job named ARXEOJTB.Z. When adding or deleting entries from the ARXEOJTB table the ARXEOJTB_TOTAL and ARXEOJTB_USED fields must be updated. This entry specifies the name of the linked phase. In addition, you may specify if REXX is to load the user program into the program area via LOAD or into GETVIS storage via CDLOAD. If LOAD is to be used, you can choose between 2 options of space checking within the program area: Either all phases are to be considered that start with the first 4 bytes of the given phasename (option'A' or blank, for instance overlay program MSHP), or only the size of the single given phasename is checked (option 'S'). Also you may define an 8 byte synonym for the user program. For example, entry 1 in Figure 11 on page 213 specifies that issuing ADDRESS LINK LIBR actually calls phase ARXLIBR.

```
$$$$ JOB JNM=ARXEOJTB,CLASS=0,DISP=L
$$$$ LST DISP=D,CLASS=A
// JOB ARXEOJTB     ASSEMBLE & LINK
/. X THIS JOB ASSEMBLES AND LINKS THE REXX/VSE MODULE ARXEOJTB
/. X USING THE HIGH LEVEL ASSEMBLER.
/. X BEFORE YOU SUBMIT THE JOB UPDATE THE LIBDEF CHAIN TO YOUR NEE
// LIBDEF *,SEARCH=PRD1.BASE
// LIBDEF PHASE,CATALOG=PRD2.CONFIG
// OPTION ERRS,SYM,CATAL,NODECK,SXREF
   PHASE ARXEOJTB,*
// EXEC ASMA90,SIZE=(ASMA90,50K)
*/***START OF SPECIFICATIONS*****************************************/
*/*                                                                 */
*/*01*  MODULE-NAME = ARXEOJTB                                      */
*/*                                                                 */
*/*01*  DESCRIPTIVE-NAME = REXX EOJ Return Table                     */
*/*                                                                 */
*/*     ************************************************      */
*/*     *                                              *      */
*/*     *   LICENSED MATERIALS - PROPERTY OF IBM       *      */
*/*     *   THIS MODULE IS "RESTRICTED MATERIALS OF IBM"*      */
*/*     *   5686-066 (C) COPYRIGHT IBM CORP. 1995      *      */
*/*     *   SEE COPYRIGHT INSTRUCTIONS                 *      */
*/*     *                                              *      */
*/*     ************************************************      */
*/*                                                             */
*/*                                                             */
*/*                                                             */
*/*01*  FUNCTION = Declare the REXX EOJ Return table used by     */
*/*               LINKxxx Host Command Environments.            */
*/*                                                             */
*/*                                                             */
*/*     DESCRIPTION = Each entry of this table represents       */
*/*                   a phase to be loaded by ADDRESS LINK      */
*/*                   or ADDRESS LINKPGM.                       */
*/*                   You may assign a synonym under which      */
*/*                   the phase is to be invoked by the LINK    */
*/*                   and LINKPGM environment. You can also     */
*/*                   determin whether the phase can be CDLOADed */
*/*                   into the partition getvis or LOADed into the */
*/*                   program area of the partition.            */
*/*                                                             */
*/*                   An entry in ARXEOJTB allows a program     */
*/*                   called by ADDRESS LINK/LINKPGM to use the */
*/*                   EOJ macro (SVC 14) to pass control back   */
*/*                   to REXX.                                  */
*/*                                                             */
*/*                   An entry in ARXEOJTB ensures that a program */
*/*                   called by ADDRESS LINK/LINKPGM is processed */
*/*                   under the maintask only.                  */
*/*                                                             */
*/*                                                             */
*/*                                                             */
*/*   .-------------------------------------------------.       */
*/*   | NOTE: This phase must not be loaded into the SVA |      */
*/*   '-------------------------------------------------'       */
```

Figure 9. Table of Authorized Programs - Part 1 of 3

```
*/*                                                                   */
*/*                                                                   */
*/*02*    CHANGE-ACTIVITY =                                          */
*/*                                                                   */
*/*                                                                   */
*/***END OF SPECIFICATIONS*********************************************/
ARXEOJTB           CSECT
ARXEOJTB           AMODE 31
ARXEOJTB           RMODE ANY
         TITLE 'ARXEOJTB - REXX EOJ Return Table'
ARXEOJTB_HEADER    DS 0F             /* Set up the ARXEOJTB Header   */
                   DC CL8'ARXEOJTB'
*                                    /* Addr of first ARXEOJTB entry */
ARXEOJTB_FIRST     DC A(ARXEOJTB_ENTRIES)
ARXEOJTB_TOTAL     DC F'9'           /* Total # of entries           */
ARXEOJTB_USED      DC F'6'           /* # of entries used            */
ARXEOJTB_LENGTH    DC F'28'          /* Length of each entry         */
ARXEOJTB_FFFF DC X'FFFFFFFFFFFFFFFF' /* Set Header end marker        */
*                  /* START OF TABLE ENTRIES                        */
ARXEOJTB_ENTRIES   EQU *             /* Start of entries is here     */
*/*******************************************************************/
*/* Do not change the entries below                                 */
*/*******************************************************************/
ARXEOJTB_ENTRY_1   EQU *             /* ARXLIBR  Entry 1             */
                   DC CL8'LIBR    '  /* Synonym used in ADDRESS LINK */
                   DC CL8'ARXLIBR '  /* Name of phase                */
                   DC AL4(0)         /* Must be Zero                 */
                   DC CL1'NO'        /* Phase Loaded in program area */
                   DC CL7' '         /* Reserved                     */
ARXEOJTB_ENTRY_2   EQU *             /* ARXIDCMS Entry 2             */
                   DC CL8'IDCAMS  '  /* Synonym used in ADDRESS LINK */
                   DC CL8'ARXIDCAM'  /* Name of phase                */
                   DC AL4(0)         /* Must be Zero                 */
                   DC CL1'NO'        /* Phase Loaded in program area */
                   DC CL7' '         /* Reserved                     */
ARXEOJTB_ENTRY_3   EQU *             /* ARXEOJTB Entry 3             */
                   DC CL8'MSHP    '  /* Synonym used in ADDRESS LINK */
                   DC CL8'MSHP    '  /* Name of phase                */
                   DC AL4(0)         /* Must be Zero                 */
                   DC CL1'YES'       /* Phase Loaded in program area */
                   DC CL1'ALL'       /* Consider all phases @56301UB */
                   DC CL6' '         /* Reserved            @56301UB */
ARXEOJTB_ENTRY_4   EQU *             /* ARXEOJTB Entry 4             */
                   DC CL8'ASSEMBLY'  /* Synonym used in ADDRESS LINK */
                   DC CL8'ASMA90  '  /* Name of phase                */
                   DC AL4(0)         /* Must be Zero                 */
                   DC CL1'NO'        /* Phase Loaded in program area */
                   DC CL7' '         /* Reserved                     */
ARXEOJTB_ENTRY_5   EQU *             /* ARXEOJTB Entry 5             */
                   DC CL8'LNKEDT  '  /* Synonym used in ADDRESS LINK */
                   DC CL8'$LNKEDT '  /* Name of phase                */
                   DC AL4(0)         /* Must be Zero                 */
                   DC CL1'NO'        /* Phase Loaded in program area */
                   DC CL7' '         /* Reserved                     */
ARXEOJTB_ENTRY_6   EQU *             /* ARXEOJTB Entry 6             */
                   DC CL8'DITTO   '  /* Synonym used in ADDRESS LINK */
                   DC CL8'DITTO   '  /* Name of phase                */
                   DC AL4(0)         /* Must be Zero                 */
                   DC CL1'YES'       /* Phase Loaded in program area */
                   DC CL1'SINGLE'    /* Consider this phase @56301UB */
                   DC CL6' '         /* Reserved            @56301UB */
```

Figure 10. Table of Authorized Programs - Part 2 of 3

```
*/*****************************************************************/
*/* Do not change the entries above                               */
*/*****************************************************************/
ARXEOJTB_ENTRY_7   EQU *            /* ARXEOJTB Entry 7            */
                   DC CL8'        '  /* Synonym used in ADDRESS LINK */
                   DC CL8'        '  /* Name of phase               */
                   DC AL4(0)         /* Must be Zero                */
                   DC CL1'NO'        /* Phase Loaded in program area */
                   DC CL7' '         /* Reserved                    */
ARXEOJTB_ENTRY_8   EQU *            /* ARXEOJTB Entry 8            */
                   DC CL8'        '  /* Synonym used in ADDRESS LINK */
                   DC CL8'        '  /* Name of phase               */
                   DC AL4(0)         /* Must be Zero                */
                   DC CL1'NO'        /* Phase Loaded in program area */
                   DC CL7' '         /* Reserved                    */
ARXEOJTB_ENTRY_9   EQU *            /* ARXEOJTB Entry 9            */
                   DC CL8'        '  /* Synonym used in ADDRESS LINK */
                   DC CL8'        '  /* Name of phase               */
                   DC AL4(0)         /* Must be Zero                */
                   DC CL1'NO'        /* Phase Loaded in program area */
                   DC CL7' '         /* Reserved                    */
*
                   DC    C'PATCH AREA - ARXEOJTB'
                   DS    32F              Patch area
                   END   ARXEOJTB
$$/*
// EXEC LNKEDT,PARM='MSHP,AMODE=31,RMODE=ANY'
$$/&
$$$$ EOJ
```

Figure 11. Table of Authorized Programs - Part 3 of 3

Invoking VSE Utilities

You can use the ADDRESS LINK command environment to invoke VSE utilities such as LIBR, IDCAMS, Assembler, DITTO and MSHP. Note that only LIBR and IDCAMS are able to write output into OUTTRAP.

Invoking LIBR using ADDRESS LINK

```
ADDRESS LINK 'LIBR option_list'
```

option_list can only be MSHP.

Use SYSIPT or REXXIPT to supply input statements to ADDRESS LINK LIBR. Here is an example:

```
ARG sublib

ADDRESS LINK

CALL OUTTRAP libr_output.
CALL REXXIPT libr_input.

libr_input.0 = 2
libr_input.1 = 'ACC S='||sublib
libr_input.2 = 'LD ARX*.PHASE'

'LIBR'

IF word_found('ARXINIT')
   THEN SAY 'REXX/VSE was installed into' sublib
   ELSE SAY 'REXX/VSE is not installed into' sublib
EXIT

word_found:
ARG search_for
DO line = 1 to libr_output.0
IF WORDPOS(search_for,translate(libr_output.line))¬=0
   THEN RETURN 1
END
RETURN 0
```

See also demo program SETSDL described on page “REXXSSDL” on page 263.

Invoking IDCAMS using ADDRESS LINK

Invoke IDCAMS via

```
ADDRESS LINK 'IDCAMS option_list'
```

option_list corresponds to the options of the PARM command. See VSE/VSAM Commands for a description of the PARM command.

Use SYSIPT or REXXIPT to supply input statements to ADDRESS LINK IDCAMS. Here is an example:

```
ARG file_name
CALL OUTTRAP idcams_output.
CALL REXXIPT idcams_input.

idcams_input.0 = 1
idcams_input.1 = 'LISTCAT CLUSTER'

ADDRESS LINK 'IDCAMS MARGINS(1 80)'

IF rc = 0
   THEN CALL Print_Only_Lines_Including file_name
   ELSE SAY 'IDCAMS LISTCAT fails with RC='rc
EXIT

Print_Only_Lines_Including:
  ARG search_name
  DO line = 1 to idcams_output.0
    IF WORDPOS(search_name,translate(idcams_output.line))¬=0
      THEN SAY idcams_output.line
  END
  RETURN
```

Invoking ASSEMBLE and LNKEDT

This program assembles the source program, linkedits the phase and executes it.

```
CALL REXXIPT rexx_sysipt.
rexx_sysipt.0 = 5
rexx_sysipt.1 = "        PUNCH ' PHASE EXAMPLE,* '"
rexx_sysipt.2 = 'EXAMPLE START   0'
rexx_sysipt.3 = '        SR 15,15'
rexx_sysipt.4 = '        BR 14'
rexx_sysipt.5 = '        END ,'


ADDRESS JCL '// OPTION CATAL'

ADDRESS LINK

  'ASSEMBLY'

IF rc = 0 THEN
  DO
    ADDRESS JCL
    '// LIBDEF PHASE,CATALOG=DEVLIB.TEST'

    ADDRESS LINK

    'LNKEDT MSHP,AMODE=31,RMODE=24'

    IF rc = 0 THEN
      DO
        ADDRESS JCL
        '// LIBDEF PHASE,SEARCH=DEVLIB.TEST'
        ADDRESS LINK
        'EXAMPLE'
      END
  END
EXIT rc
```

See also demo program REXXASM described on page “REXXASM” on page 263.

Invoking DITTO

```
/* This exec copies tapes and compares them                          */
/* Parameters:                                                       */
/*     in: cuu of input tape unit                                    */
/*    out: cuu of output tape unit                                   */
/* nfiles: number of files to be copied                              */

ARG in out nfiles .

input.1 = '$$DITTO TT INPUT='in',OUTPUT='out',NFILES='nfiles
input.2 = '$$DITTO REW OUTPUT='in
input.3 = '$$DITTO REW OUTPUT='out
input.4 = '$$DITTO TTC INPUT='in',OUTPUT='out',NFILES='nfiles
input.5 = '$$DITTO REW OUTPUT='in
input.6 = '$$DITTO REW OUTPUT='out
input.7 = '$$DITTO RUN OUTPUT='out
input.8 = '$$DITTO EOJ'
input.0 = 8

CALL REXXIPT input.
ADDRESS JCL '// UPSI 1'
ADDRESS LINK 'DITTO'
EXIT
```

Chapter 14. REXX/VSE Console Automation

Benefits of a Programmable REXX Console

REXX/VSE Console Automation enables you to automate and make more productive the operation of your VSE/ESA console.

REXX/VSE Console Automation is centered around a REXX VSE/ESA programmable console. It provides an easy-to-use VSE/ESA console command environment that allows to activate and deactivate one or more VSE/ESA console sessions. VSE/ESA console commands may be imbedded into a REXX program. A GETMSG function receives command responses and console messages.

There is also a rich set of REXX external functions that make it easy to write REXX console applications. Thereby, a REXX program can retrieve and process console commands and react on events.

VSE/ESA Console Automation: By having a REXX VSE/ESA programmable console you can

- Write a REXX program that issues VSE console commands and retrieves the command responses. These commands include
  - VSE AR commands
  - Console redisplay commands
  - VSE/POWER, VSE/ICCF, CICS, VTAM, SQL commands.
- Program VSE console operator (inter)actions
- Monitor programs and subsystems running in VSE/ESA partitions and react on messages by giving appropriate replies.
- Control batch job processing.

A Look at VSE/ESA's Console Support

REXX Console Automation has to do with the data flow from and to consoles. The following text gives an overview of VSE/ESA's handling of console traffic and how REXX/VSE console automation fits in.

The entire console traffic is managed by the **Console Router**:

- **A message**

  written by a program is queued in the Console Router and then routed to the appropriate console(s).
- **A command**

  entered at a console is queued and routed to the responsible command processor.
- **A command response**

  issued by a command processor is queued and routed to the console that issued the command.
- **A reply**

  entered at a console is queued and routed to the program that is waiting for the reply.

The following figure serves as an illustration. Please keep in mind that "consoles" (on the right hand side) does not necessarily mean physical consoles, but rather **console programs**. For example, behind every CICS terminal there is a console program. A REXX procedure that uses the console command environment is also is a console program.

The figure shows the names of macros (such as WTO, MCSOPER) that are associated with a given activity. These macros are important elements of REXX/VSE Console Automation. They are discussed after the figure.

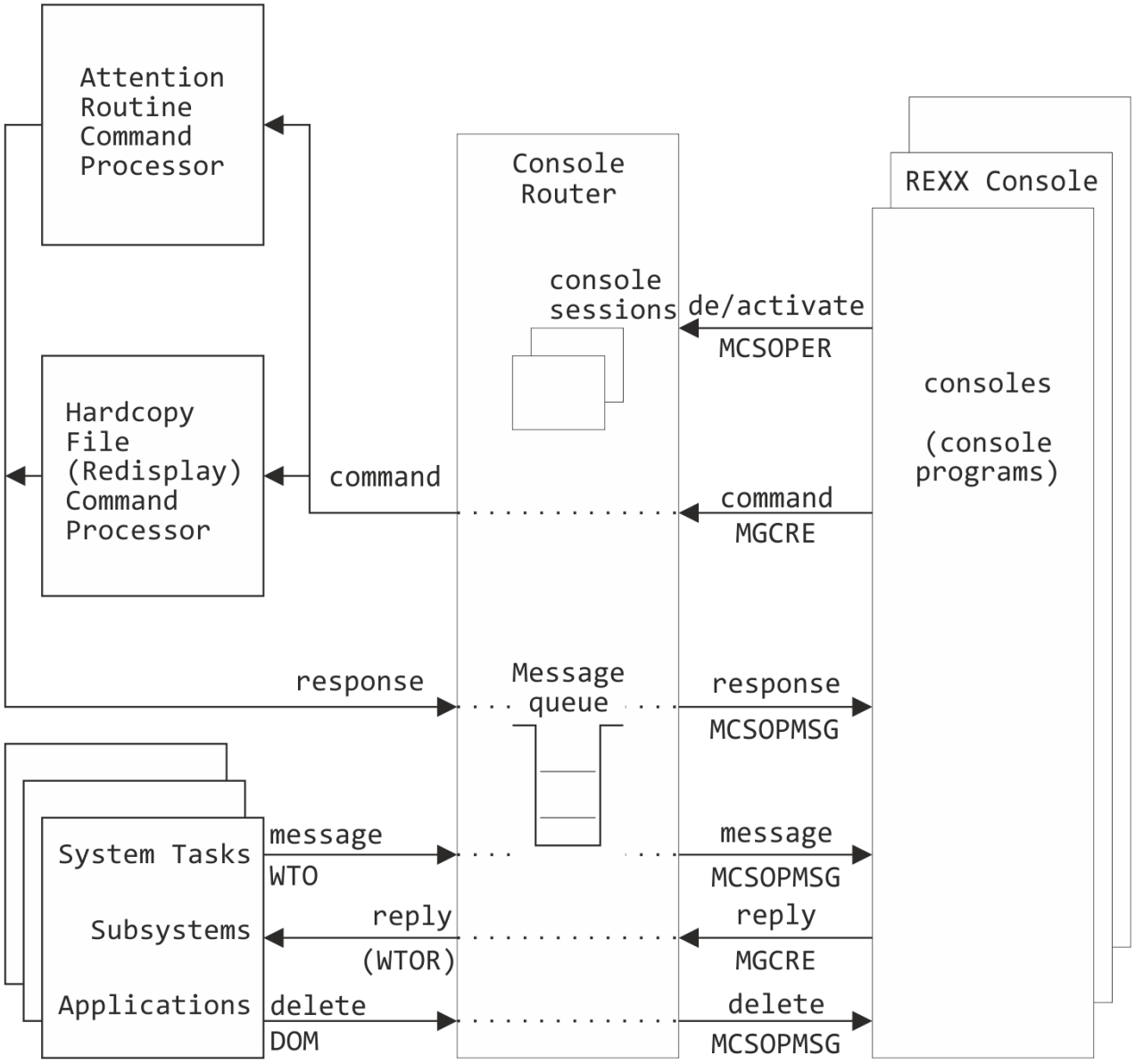


Figure 12. Console Data Flow

On each side of the Console Router there is a set of interfaces:

- I/O interfaces for the system and for applications to communicate with consoles.
- General-use console interfaces for the console programs to interface with the system.

The interfaces are briefly described in the following. Together with each individual element, a reference to an equivalent *REXX Console Automation* function is given.

Console I/O Interfaces

Any program can perform I/O to and from a console: it can write a message to a console, or read the input that was entered at the console. It can also delete a message from the console.

For these purposes three macros are available which are presented in the following. For detailed descriptions refer to the IBM manual *VSE/ESA System Macros Reference*.

WTO - Write to Operator

The WTO macro is used to write a (single or multi-line) message to one or more consoles.

In *REXX Console Automation*, the SENDMSG function uses the WTO macro.

Because *REXX Console Automation* is able to maintain one or more console sessions of its own, the SENDMSG function allows for console-to-console communication.

WTOR - Write to Operator with Reply

This macro works much like the WTO macro, but additionally requests a reply from the operator.

In REXX, PULL from SYSLOG uses the WTOR macro.

DOM - Delete Operator Message

Some messages are displayed with **highlighting**. They do not disappear from the screen, rather remain in **HOLD** state. For example, a message written via the WTOR macro is displayed in highlighted form to permanently draw attention to the fact that a reply is needed.

When a program recognizes that the reason for the highlighting does no longer exist, it issues a DOM macro. This removes the highlighting, and the message usually disappears from the screen.

Likewise, the operator may request "deletion" (dehighlighting, to be exact) of a highlighted message. Again, this causes the DOM macro to be activated.

In *REXX Console Automation* this task is performed by the DELMSG function.

General-Use Console Interfaces

These privileged macro interfaces are **intended for system programs**. A portion of their functionality is **generally available** to REXX programs through *REXX Console Automation*.

To understand the concept and possible error messages, a brief overview of the general-use console interfaces follows.

MCSOPER - Activate Console

This macro is used to activate (or deactivate) a console session. A console session is required for communicating with the system. Communication means retrieving messages from the system and/or passing input like commands or replies to the system.

In *REXX Console Automation* a console session is established through the ADDRESS CONSOLE 'ACTIVATE...' command. Along with this command, a console profile can be specified which determines the subset of message traffic to be handled.

A console session is terminated through the ADDRESS CONSOLE 'DEACTIVATE...' command.

MCSOPMSG - Retrieve Message

This macro is used to retrieve a message from the system.

In *REXX Console Automation*, the GETMSG function and, for retrieval by special search criteria, the FINDMSG function use the MCSOPMSG macro.

A **command response** that the system sends to the console is a special type of message. The GETMSG allows to select only the command responses from the entire collection of console messages.

MGCRE - Create Command or Reply

This macro is used to pass input like a command or a reply to the system.

In *REXX Console Automation*,

- the command ADDRESS CONSOLE 'command_or_reply_string' and
- the SENDCMD function

use the MGCRE macro.

The MGCRE macro allows to pass a command and response correlation token (CART). The CART can later be used to select, from a heterogeneous collection of console messages coming from all kinds of sources, just the right command response.

REXX Console Automation has a special console command (the CART command) that establishes a CART for the current console session. This CART is then attached to any command that is issued from the current console session.

Master Console versus User Console

Master console and user console are distinguished by the level of **command authorization**.

Master Console

A master console has unrestricted authorization to reply to all outstanding messages and to issue any kind of system command. One or more master consoles can be active at the same time. The **system console** is one such master console, mainly used to IPL the system or as backup when no other master console is active.

User Console

A user console can only issue a restricted set of system commands, just enough to perform operation tasks within its own scope and without impacting system wide operation. It can only respond to messages that are directed to it.

A user console receives only those messages that are specifically directed to it. They are messages that relate to its own activities. For example,

- The system response to the REPLID command is sent only to the console that issued the command.
- Job-related messages are sent to the console that is designated as the recipient of those messages, very often the console that submitted the job.

Redisplay from the hardcopy file is only possible for messages that were originally routed to this user console.

Message routing to user consoles can be controlled by the ECHO/ECHOU=user-id option of the VSE/POWER JOB statement. When this option is included in a submitted job, all messages related to the execution of that job are routed to the console of the ECHO/ECHOU user-id.

Routing Codes

There are two sides to the routing code. On the one side, the **originator** of a message indicates, in terms of routing code(s), the **console** type where the message is to be delivered. On the other side is the console which has routing codes defined to indicate which messages it wants to have delivered. When both routing codes match, a message is delivered at that console.

Examples of routing codes are:

2
: The message indicates a change of the system status that requires action by an operator with master authority.

7
: A message gives information to the unit record pool about a unit record device, for example a request to mount a printer train.

11
The message is intended for the problem programmer and is to be routed to the console identified by an ECHO/ECHOU option or to a terminal of the Interactive Interface.

In *REXX Console Automation*, routing codes of a console are defined in its **console profile** (see "Activating a Console Session" on page 222). For example,

REXALLRC
means "receive all routing codes," or in other words: a master console that receives all messages.

REXNORC
means "receive no routing codes," or in other words: a master console that does not want to see any messages. This is useful when your REXX program is primarily concerned with issuing commands and reacting on the corresponding command responses.

Service Offerings

Console Command Environment

REXX Console Automation allows you to establish (*activate*) one or more VSE console sessions from your REXX program. After you have activated a console session, you can issue VSE system and subsystem commands and retrieve the corresponding responses.

You can only work with **one console** at a time. By default, this is the console that has been activated as the most recent one. This is referred to as the *current console*. You can switch from one active console to another which becomes the new *current console*.

The VSE system and subsystem commands that you may issue during a console session depend on the authorization of the userid associated with the job that starts your REXX program. Please refer also to section "Security Considerations" on page 223.

Console Commands

At the REXX console, you can issue either

- REXX console commands, or
- VSE console commands.

REXX Console Commands

The REXX console command environment provides unique REXX console commands. Using ADDRESS CONSOLE, you may issue one of the following commands:

- ACTIVATE
- CART
- CONSTATE
- CONSWITCH
- DEACTIVATE

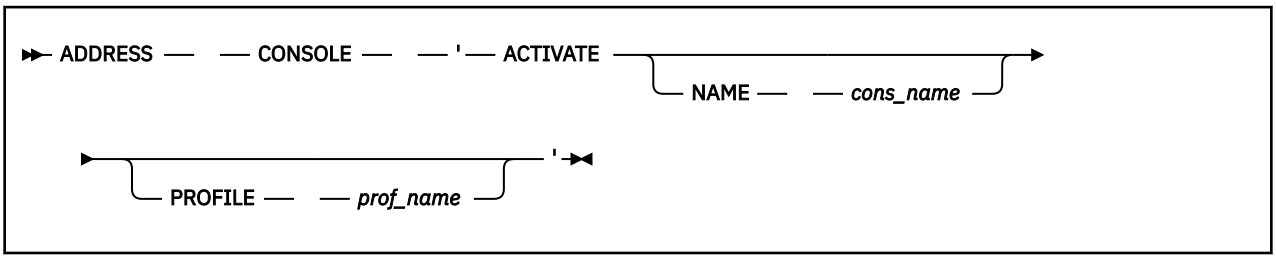
VSE Console Commands

Commands that are not REXX console commands get passed to the VSE console command processor. This is also true for replies to messages.

Note: To be able to issue VSE console commands, you need to have a *current console*.

Activating a Console Session

You activate a VSE console session via the command:



where:

cons_name
Specifies the name of the console that you want to activate. The name is an alphanumeric character string of between 4 and 8 bytes. You choose the name.

If the NAME parameter is not specified, *cons_name* defaults to REXX.

prof_name
Specifies the profile name you want to use for the console that you are about to activate. You must name one out of a set of predefined REXX profiles (tailored to the master or the user console). A set of predefined console profiles is shipped as object code. Their names and purpose are shown in the table below.

If the PROFILE parameter is not specified, *prof_name* defaults to REXX which is one of the predefined console profiles.

Name	Master/User	Description
REXX	User	Receives only messages that are specifically routed to this console. These are: 1. Messages issued from a job that names this console in the VSE/POWER JECL parameter ECHO. 2. Messages directed to this console from the WTO macro. 3. Responses to commands that were issued from this console.
REXALLRC	Master	Receives command responses **and all** routing codes.
REXNORC	Master	Receives command responses **but no** routing codes.
REXAUTO	Master	Receives messages from an automated message handling program such as VSE/OCCF.

Example: As an example, you activate a VSE console session with console name ***myecho*** via

```
ADDRESS CONSOLE 'ACTIVATE NAME myecho'
```

or via

```
ADDRESS CONSOLE
'ACTIVATE NAME myecho'
```

Because the default profile name REXX applies, this command establishes a user console and, therefore, gives only limited command authority.

The console name (***myecho*** in this example) is significant for routing messages issued by a VSE/POWER job to this REXX console. This is illustrated in “Routing Messages From and Replies To a Specific Partition” on page 227.

Return Codes: For return codes of the ACTIVATE command and their explanations, please refer to “ARXCONAD Return Codes” on page 228.

Security Considerations

To what extent you can establish a master console or a user console depends on whether security is active in your system.

If your system runs with security not active, there are no limitations. But be aware that your system resources may not be adequately protected against unauthorized accessing.

If your system runs with security active (IPLed with SEC=YES in the SYS command), a **VSE security user-id** must be supplied in the job that calls the REXX program. It is supplied either in the job control statement

```
// ID USER=user-id,...
```

or in the VSE/POWER JECL statement

```
* $$ JOB... SEC=user-id,...
```

If no VSE security user-id is given, the ACTIVATE command fails with return code -12 (security violation).

The VSE security user-id is checked against a user profile in the Access Control Table (DTSECTAB). The profile's parameters MCONS and AUTH are of significance, as shown in the diagram below.

MCONS=YES gives *master console* authorization. AUTH=YES indicates that the user is the security administrator (this implies master console authorization, or MCONS=YES).

As shown in the following table, the security administrator is free to choose any cons_name. Other users must specify the VSE security user-id in order to get access to the desired console.

DTSECTAB Parameters	cons_name of REXX Master Console	cons_name of REXX User Console
AUTH=YES	any	any
MCONS=YES,AUTH=NO	VSE security user-id	VSE security user-id
MCONS=NO,AUTH=NO	none	VSE security user-id

Receiving Messages from VSE/OCCF

You may have the VSE/ESA optional program VSE/OCCF (Operator Communication and Control Facility) installed. VSE/OCCF is an operator automation program by itself and is in no way required for using *REXX Console Automation*.

Through VSE/OCCF, messages can automatically be routed to a NetView console. In *REXX Console Automation* you may request that these messages are routed to a REXX console, instead of to a NetView console.

Rerouting to the REXX console is achieved by going through the following steps.

1. Install VSE/OCCF

   VSE/OCCF is a VSE/ESA optional program. To install it, you are encouraged to use the *Install Programs* dialog of the VSE/ESA Interactive Interface.

2. Build a VSE/OCCF Message Automation Table

   Library 59 contains job skeleton SKOCCF that builds the VSE/OCCF message automation table for the VSE/ESA Unattended Node Support. You can use it as a model for your own coding.

3. Start VSE/OCCF

   To start VSE/OCCF issue the command

   ```
   QSTART matab
   ```

where *matab* is the name of the message automation table.

4. Request rerouting by VSE/OCCF

   Issue the REXX command SYSDEF

   ```
   rc = SYSDEF('CONNECT OCCF')
   ```

   This causes VSE/OCCF to route messages that are designated to be routed to NetView to the REXX console, instead. The command is described in section “SYSDEF” on page 240.

5. Activate a REXX console with profile REXAUTO.

   Issue the command

   ```
   ACTIVATE CONSOLE ... PROFILE REXAUTO
   ```

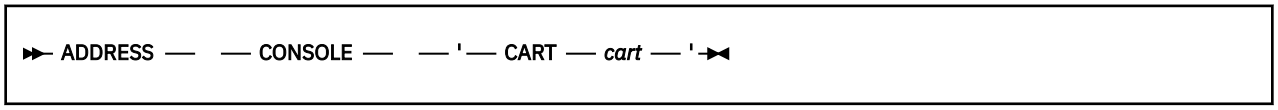
   This defines the REXX console as a type that receives messages from an automated message handling program such as VSE/OCCF.

When your REXX program has finished its work, it should reset the above functions through the following commands:

QSTOP (to suspend VSE/OCCF functions)
rc = SYSDEF('DISCONNECT OCCF')
QEND (to terminate VSE/OCCF processing)

Creating a Command and Response Correlation Token (CART)

You define a CART for the current console via the command



The operand specification is as follows:

cart
Specifies the value of the CART as a string of up to 8 bytes (a larger string will be truncated down to 8 bytes). You are free to choose any value. The string must not contain any blank.

The CART is associated with every command that is issued from the current console. The CART serves to distinguish between heterogeneous command responses, each coming from a different command. The GETMSG function will then not pick up any command output that has accumulated, but rather in a selective manner.

This is illustrated in the following example. (The GETMSG function is described in “GETMSG” on page 232.)

```
mask1= 'FF0000000000000'X         /* compare CARTs on first byte      */
mask2= 'FFFFFF000000000'X         /* compare CARTs on first 3 bytes   */
zero = '0000000000000000'X        /* no checking                      */
'CART AttenRtn'
'MAP'                             /* VSE AR command                   */
'CART RED'
'RED 10L,F5'                      /* REDisplay command                */
'CART' zero
'REPLID'                          /* Console Router command           */
rc = GETMSG('MSG.','RESP','A',mask1,5)   /* get MAP output            */
rc = GETMSG('MSG.','RESP','RED',mask2,5) /* get REDisplay output      */
rc = GETMSG('MSG.','RESP',,,5)           /* get REPLID output         */
```

Note: When having multiple commands processed at the same time, their output could very well be intermixed. Always use CARTs to get the outputs separated from each other.

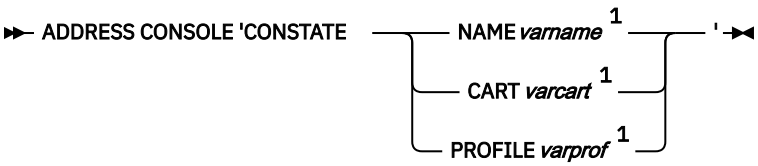
The direct succession of the three commands shown above is only possible because the commands are each handled by a different command processor. Outstanding responses from one VSE command (processor) must first be retrieved before you can start another command belonging to the same

command processor. This subject is discussed further in section “Having Command Responses Outstanding in Parallel” on page 226.

Return Codes: For return codes of the CART command and their explanations, please refer to “ARXCONAD Return Codes” on page 228.

Querying the Current Console Setting

You can enquire about the current console settings via the CONSTATE command. The command has the following format:



Notes:

[1] The variables may be entered in any order, but at least one of them must be specified.

where

varname
Is the name of a variable that returns the name of the current console.

varcart
Is the name of a variable that returns the setting of the current cart.

varprof
Is the name of a variable that returns the name of the profile that is associated with the current console.

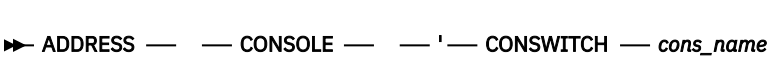
Return Codes: For return codes of the CONSTATE command and their explanations, please refer to “ARXCONAD Return Codes” on page 228.

Switching to a Console Session

You can only work with **one console** at a time. By default, this is the console that has been activated as the most recent one. This is referred to as the *current console*.

Via the CONSWITCH command you can switch to another console and make it the *current console*. The CONSWITCH command resets the CART.

The command has the following format:



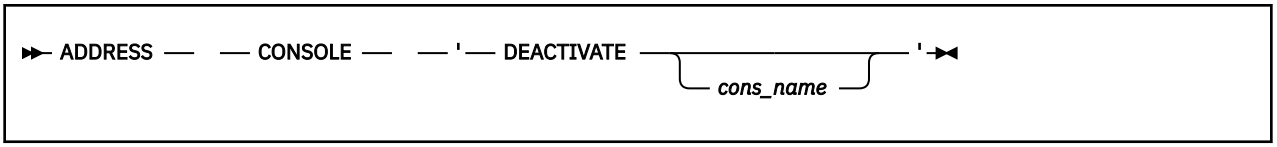
where

cons_name
Specifies the name of the console you want to switch to.

Return Codes: For return codes of the CONSWITCH command and their explanations, please refer to “ARXCONAD Return Codes” on page 228.

Deactivating a Console Session

You deactivate a VSE console session via



where:

cons_name
Specifies the name of the console to be deactivated. It is the name of the console that you activated with an ACTIVATE command. The default name is REXX.

Return Codes: For return codes of the DEACTIVATE command and their explanations, please refer to "ARXCONAD Return Codes" on page 228.

Note: If you deactivate the current console, no console is current. To make another console the current console, you have to use the CONSWITCH command.

Temporarily Shutting off an (Unknown) Console

It can sometimes happen that you want to open a REXX console but you don't know from where your program was called and whether another console is active out there.

The following example shows how you could proceed (it is assumed that there is already a *current console*).

```
ADDRESS CONSOLE
'CONSTATE NAME oldcons CART oldcart' /* save state of current console*/

'ACTIVATE MYCONS'                    /* activate my own console      */
                                     /* which now becomes the        */
                                     /* current console
 ... doing my thing ...

                                                                     */
'DEACTIVATE MYCONS'                  /* deactivate my own console    */
'CONSWITCH' oldcons 'CART' oldcart   /* make the previously current  */
                                     /* console current again        */
```

Examples of REXX and VSE Console Commands

```
ADDRESS CONSOLE
'ACTIVATE NAME master PROFILE rexnorc'  /* Activate master console. No */
                                        /* routing codes are received  */

'ACTIVATE NAME rexx'                    /* Activate user console       */

'CONSWITCH master'                      /* Switch to the master console*/

'CART syscmd'                           /* Specify a user defined cart */

'D NET,APPLS'                           /* Issue VTAM command          */

'DEACTIVATE master'                     /* Deactivate console          */
'CONSWITCH rexx'                        /* Switch to the user console  */

'CONSTATE NAME consname',               /* Query for the current       */
         'CART cart PROFILE consprof'   /* console settings            */

'MAP'                                   /* Issue AR command            */

'DEACTIVATE rexx'                       /* Deactivate console          */
```

Having Command Responses Outstanding in Parallel

The VSE/ESA console support requires that you retrieve the outstanding responses for one VSE system command before you send the next VSE system command. Therefore, only one VSE system command **per console** can be sent at a time.

After sending a VSE system command to the console, you need to **wait** until the response is complete. You must retrieve the response before you send the next VSE system command. This is illustrated in the following scenario:

```
ADDRESS CONSOLE

"system_cmd_1"
fc = GETMSG(msg.,'RESP',,,30)
 ...
 ...
"system_cmd_2"
fc = GETMSG(msg.,'RESP',,,30)
```

You may get return code -10 if you send a second VSE system command before you retrieve the response via the GETMSG function.

There are different "types" of VSE console command processors (each of which is able to process one command in parallel to the others):

1. AR commands

   This includes VSE/POWER, CICS and VTAM commands.
2. VSE Redisplay commands
3. VSE Console Router commands, such as REPLID or CANCEL AR.

Although you may have for each of these console command processors one command response outstanding in parallel, it is strongly recommended that you retrieve the response for one command before you issue the next command.

Routing Messages From and Replies To a Specific Partition

The ECHO/ECHOU parameter of the VSE/POWER JECL statement * $$ JOB relates to the NAME keyword of the ACTIVATE CONSOLE command. Consider, for example, the following VSE job.

```
* $$ JOB JNM=MYJOB ECHO=(ALL,MYECHO)
// JOB MYJOB
 ....
 ....
/&
* $$ EOJ
```

All messages issued by job MYJOB are routed to the REXX console which was activated under the name 'MYECHO' by a REXX program. In order to retrieve the message, the REXX program issues the GETMSG function. The REXX program replies to the outstanding reply by just enclosing the reply in quotes:

```
ADDRESS CONSOLE 'ACTIVATE NAME myecho'
 .....
fc = GETMSG(msg.,'MSG',,,5)               /* retrieve next message  */
                                          /* and wait max. 5 seconds*/
 .....

GetReplyId(msg.1) '...reply text...'      /* reply to message       */

 .....



GetReplyId:
 ARG message
 position = POS('-',WORD(message,1))   /* did we get something like:  */
 IF position > 0                       /*     R1-47 // PAUSE         */
    THEN                               /* extract from message...    */
     replyid = SUBSTR(WORD(message,1),position+1)    /* reply ID     */
    ELSE
     replyid= WORD(message,2)          /* reply ID                   */
RETURN replyid
END
```

The above example shows how to handle console messages that are "fresh," that is, messages that **have just been delivered at the console**. Another example (see “Scan the Hardcopy File” on page 264) shows how to retrieve messages that had **appeared earlier** and are now stored on the hardcopy file.

Tracking of Operator Communication

Responses to operator-given commands are only routed to the console the operator is using for his communication, not to other defined master consoles. Thus it is not possible to track ongoing operator communication via a REXX console program. Since operator dialogs are logged in the hardcopy file, they can be scanned using the REDISPLAY command.

Console Host Command Replaceable Routine

Module ARXCONAD processes all host commands requested by ADDRESS CONSOLE. It uses the REXX/VSE interface for host command environment routines as described elsewhere in this manual, beginning in section “Host Commands and Host Command Environments.” on page 24.

Entry for ARXCONAD in Table SUBCOMTB

The Host Command Environment Table (SUBCOMTB) contains an entry for the CONSOLE host command environment. Via this table entry, you can control whether or not your REXX programmers have access to console automation functions. To invalidate the access, make a copy of ARXPARMS, blank out the constants NAME and ROUTINE, and catalog the copy into a separate sublibrary. Through different LIBDEF chains, you make the console automation functions available to one group of programmers and deny them to another group.

ARXCONAD Return Codes

ARXCONAD may issue the following return codes:

-1
Console has already been activated.

-2
Either the console name is not found in the list of activated consoles, or there is no current console.

-5
Syntax error, for example invalid token, or invalid profile.

-6
Error in ARXEXCOM.

-7
Console profile table ARXCPROF.PHASE not found.

-8
Console profile not found in table ARXCPROF.PHASE.

-9
No more storage.

-10
A VSE system macro failed. REXX message ARX0565I shows the macro name. You can call the SYSVAR function to receive the return and reason code. Refer to “Return and Reason Codes” on page 267 for explanations of those codes.

-11
Error in console table service routine.

-12
Security violation.

Console-related REXX Functions

REXX Console Automation has several REXX functions that allow a REXX program to work with the REXX console.

These functions are described in the following sections. They are presented in alphabetical order.

DELMSG
FINDMSG
GETMSG
LOCKMGR
MERGE
OPERMSG
SENDCMD
SENDMSG
SYSDEF

Some functions return a **function code**. Function codes are listed and explained together with the function descriptions.

Many error conditions lead to messages

REXX syntax error 40 - invalid call to routine... plus

```
ARX0960E ERROR Running Function xxxxxxxx, RC=nn
```

You find explanations for each error code RC in section “Error Codes of Failing Functions” on page 242.

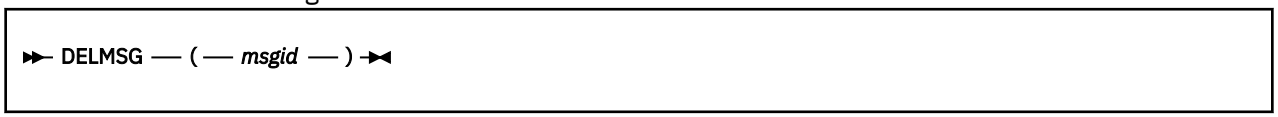
Some error conditions are detected by VSE console support macros. These macros give return and reason codes which you can retrieve by calling the SYSVAR function. The return and reason codes are explained in “Return and Reason Codes” on page 267.

DELMSG

The DELMSG function removes the HOLD state from a message. On a real console screen, the HOLD state is visible from the highlighted display toward the top of the screen to indicate that an action or reply is pending. DELMSG does not truly *delete* a message, rather resets the intensity attribute from *high* to *normal* and positions the message at its proper place within the entire set of console messages.

The DELMSG function comes into play when the condition that caused a message to be displayed does not exist anymore, for example when a device became ready.

DELMSG has the following format:



where

msgid
Is the ID of the message to be deleted. This ID can be obtained from the Message Data Block (MDB) variable MDBGMID that the GETMSG function returns (see “Message Data Block (MDB) Variables” on page 234).

A REXX procedure (see “REXXDOM” on page 261) is included in your system for demonstration purposes. It uses the DELMSG function and shows how, under *REXX Console Automation*, you can change the physical message presentation.

FINDMSG

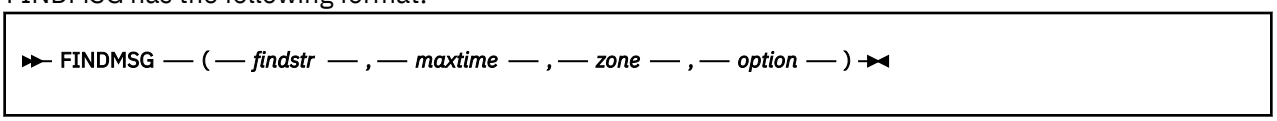
This function retrieves VSE console messages until the find criteria is fulfilled. It can be used to monitor applications that issue console messages.

FINDMSG searches for 'findstr' in all VSE console messages that have accumulated at the REXX console since the console had been activated (plus earlier messages that are in HOLD state). If successful, FINDMSG returns the **first** matching console message, otherwise a null string.

The FINDMSG function sets special REXX variables as output. If FINDMSG cannot find a matching 'findstr', these variables are set to null strings.

Name	Description
MDBCPNUM	5-digit job number of the VSE/POWER job that issued the message.
MDBCRET	Command processor return and reason codeWORD(mdbcret,1) is the 4-digit return code. WORD(mdbcret,2) is the 4-digit reason code. Please refer to “Command Processor Return and Reason Codes” on page 270.
MDBGDOM	One character with a value of '1' or '0'. A '1' indicates that a message whose ID is stored in MDBGMID is to be deleted, for example due to a preceding DOM (Delete Operator Message) macro or a delete request from the operator.
MDBGHOLD	One character indicating whether the message is a **highlighted** message to be held on the console ('1') or not ('0'). If the message is a response to the REDISPLAY command issued with the HOLD option, MDBGHOLD has a value of '0'.
MDBGJBNM	8-character job name of the VSE/POWER job that issued the message.
MDBGMID	Message ID as an 8-character representation for a 4-byte field of 8 hexadecimal values. If the message is a response to the REDISPLAY command and is not a **highlighted** ("HOLD") message, MDBGHOLD contains a null string; a valid MDBGMID is only returned if that message is a HOLD message.
MDBGDSTP	Year concatenated by day of the year, for example 1995131
MDBGTIME	Time of the day, for example 14:07:26.65
SYSTBLENTRY	Matching table entry if findstr specifies a table.

FINDMSG has the following format:



Arguments are

findstr
character string to be found as substring within a console message.

findstr can be specified in the format *lib.sublib.mn.mt*. The specified library member is then considered as a table. Its entries, from the second word through position 71, serve as search argument. Assuming that your message action table contains three entries

```
AnyWord    Enter Y to bypass verification
another new data
and_a_3rd     Enter
```

and among the console messages the following message appears

```
BG 0000 Enter new data set name
```

then the second entry would yield a match (also the third if the second entry were not there). In case of a match, the corresponding table entry is returned in its entirety in the REXX variable SYSTBLENTRY. In the above example, SYSTBLENTRY would contain the string

```
another new data
```

A later section shows the practical use of the table (see Figure 13 on page 250). There the first word of a table entry indicates an action to be taken in case of a match. The table is therefore called **REXX message action table**.

If you do not specify the 'findstr' parameter, FINDMSG uses the REXX message action table previously loaded into main storage. If not available, message ARX0960E is raised together with error code 48.

Note: If the search string happens to be of the format *lib.sublib.mn.mt*, you must use the "table approach," that is, place the string into a table entry. Placed directly into the function call, it would be interpreted as name of a message action table instead of a search string.

When using the "table approach," be aware that I/O takes place when the message action table is being read. In the example below, the first coding sequence would yield a better performance than the second because only one read operation is needed.

```
call FINDMSG 'prd1.base.rexxtabl.z',,,'LOADACTN' /* Load message
                                        action table into main storage */
do forever
 ....
 message = FINDMSG(,10)
 ....
end



do forever
 ....
 message = FINDMSG('prd1.base.rexxtabl.z',10)
 ....
end
```

maxtime
maximum number of seconds to run. Default is 5 seconds.

zone
specifies, after the keyword ZONE, a beginning position and an ending position within the message where a match with 'findstr' is searched for: 'ZONE mm nn' If not specified, the entire message is searched.

option
is either a concatenation option with keywords CONCAT or NOCONCAT (NOCONCAT is the default), or a load-table option with keyword LOADACTN.

CONCAT requests that multiple text lines of a message are treated as one long message string for the purpose of searching. See also the section below, “Handling of Multi-line Messages” on page 232.

LOADACTN requests that a message action table gets loaded into main storage in a format that can be used by the FINDMSG function. As an example of the layout of the message action table, you may use the table contained in member REXXTABL.Z. Please refer to section Figure 13 on page 250.

With the LOADACTN keyword, only the first parameter of the function call is relevant. It is the member name of the source, that is, the message action table in the format *lib.sublib.mn.mt*.

Note: The non-matching console messages are no longer available after FINDMSG has looked through them. In other words, a subsequent function call of FINDMSG would not be able to access those messages once more.

Examples of Functions Calls

```
message = FINDMSG('ENTER DATA',60)
message = FINDMSG('PRD2.PROD.MSG.TABLE',60,'ZONE 1 20')
```

Handling of Multi-line Messages

Messages that are longer than 80 characters appear in the REXX console as text lines of up to 80 bytes. When option NOCONCAT is set, the FINDMSG function searches, with a given 'findstr,' against every text line as if this were a complete message by itself. A ZONE specification would be valid for every text line. A string that continues from one text line into the next will not yield a match with 'findstr.'

When the concatenation option is set to CONCAT, FINDMSG treats all text lines as one long message for the purpose of searching. In this situation it could be cumbersome to determine the proper ZONE value.

If a match comes true, FINDMSG connects the individual text lines into one long string and returns the string as its value. It returns the matching table row in variable SYSTBLENTRY. This is valid for both options, CONCAT or NOCONCAT.

Note that when 'findstr' comes from a table, the concatenation option can lead to different matches. Be aware that with option NOCONCAT each text line (beginning with the first, then the second, and so on) is checked against the **entire table**. Consider the following example:

Your REXX console program issues the VSE/POWER command **d rdr,rexx*** In this case VSE/POWER sends a multi-line message.

```
F1 0001 1R46I  READER QUEUE   P D C S  CARDS
F1 0001 1R46I  REXXTRY  08089 3 L 0        6 FROM=(REXXLOAD)
F1 0001 1R46I  REXXASM  08090 8 L Y        6 FROM=(REXXLOAD)
F1 0001 1R46I  REXXWAIT 08070 3 L Y        6 FROM=(REXXLOAD)
```

Assume that a message action table had been loaded via FINDMSG(.....,'LOADACTN')

```
Action_1    REXXWAIT
Action_2    REXXASM
Action_3    REXXTRY
```

FINDMSG(,10,,'NOCONCAT') leads to Action_3 FINDMSG(,10,,'CONCAT') leads to Action_1

This result is due to the particular sequence of table entries. Both options would lead to Action_1 if the table had been arranged like so:

```
Action_1    REXXTRY
Action_2    REXXASM
Action_3    REXXWAIT
```

Error codes of FINDMSG are listed under “Error Codes of Failing Functions” on page 242.

GETMSG

The GETMSG function retrieves a message (which can be a command response) depending on a msgtype parameter, as shown in the function format, below. It returns a function code that replaces the function call.

Msgtype 'RESP':

The GETMSG function, with msgtype 'RESP' specified, retrieves, in variables, all command responses that have accumulated for the console session. The command response consists of one or more *messages*.

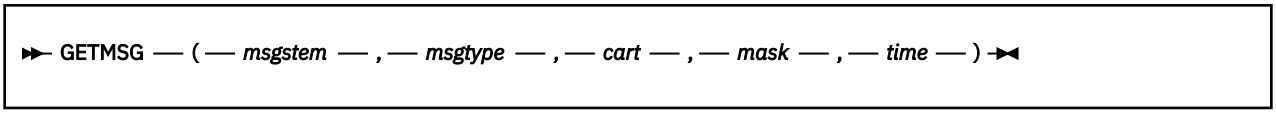
The GETMSG function retrieves the response(s) and stores the text into successive variables until the last message of the response is found and no more responses are available, or until a timeout occurs.

Msgtype 'MSG':

The GETMSG function, with msgtype 'MSG' specified, retrieves only one message at a time. The message itself may have more than one line. Each line of the message text is stored into successive variables.

Function Format

GETMSG has the following format:



where

msgstem
is the stem into which GETMSG places the message text. For example, if 'msg.' is specified as msgstem and GETMSG retrieves three lines of message text, GETMSG places these lines into the stem consisting of msg.1, msg.2 and msg.3. GETMSG stores the number of lines retrieved into the variable msg.0.

msgtype
is the type of message to be retrieved

MSG
requests retrieval of any next message from the console. Messages are retrieved on first-in-first-out basis. If you do not specify any msgtype, 'MSG' is the default.

RESP
requests retrieval of all command responses that have accumulated for the console session.

In addition to the message itself, a block of **MDB variables** (for msgtype MSG) or an **array of MDB variables** (for msgtype RESP) is returned; see below under "Message Data Block (MDB) Variables" on page 234.

cart
applies only to msgtype RESP: *command and response correlation token* (CART) specifies which command responses are to be retrieved. A command may have a CART associated with it (see also "Creating a Command and Response Correlation Token (CART)" on page 224). By matching this CART against the CART specified here in GETMSG, only specific command responses can be selected for retrieval.

The cart is a value of up to 8 bytes. If you do not specify a cart, then the default cart of 8 bytes with hexadecimal zeroes (XL8'0') is used.

mask
acts as a filter for the comparison on the two **CART** values. Only the bit position where the mask contains a '1' takes part in the comparison.

The mask is specified as a value of 16 hexadecimal digits, enclosed in quotes and terminated by an X. A value of 'FFFF000000000000'X, for example, requests that only the first two bytes of the cart are to be compared. If you do not specify a mask, the default of '0000000000000000'X is used, in other words: no comparison takes place and GETMSG retrieves all available responses.

time
is the maximum number of seconds the REXX program waits for the message or the response to arrive. The default is one second.

GETMSG Function Codes

The GETMSG function issues the following function codes:

0
GETMSG processing was successful. GETMSG retrieves a message (if msgtype is 'MSG') or the complete response (if MSGTYPE is 'RESP').

4
GETMSG processing was successful. However, GETMSG did not retrieve a message (when msgtype is 'MSG') or a response (when msgtype is 'RESP'). GETMSG returns function code 4 if one of the following occurs:

- No message available
- Search criteria did not match

- Time limit expired.

5
Applies only to msgtype 'RESP': search criteria did not match but there is at least one other message queued for the console.

8
GETMSG function failed due to the failure of a VSE macro. REXX message ARX0565I shows the macro name. You can call the SYSVAR function to receive the return and reason code. Refer to "Return and Reason Codes" on page 267 for explanations of those codes.

12
GETMSG processing failed. A console session is not active.

16
GETMSG processing failed. A console session was being deactivated during GETMSG processing.

20
GETMSG processing failed due to ARXEXCOM error or stem count error.

Message Data Block (MDB) Variables

The GETMSG function sets the following REXX variables, either as one block (for msgtype MSG) or (for msgtype RESP) as multiple blocks thus giving an array of variables.

Name	Description
MDBCPNUM	5-digit job number of the VSE/POWER job that issued the message.
MDBCRET	Command processor return and reason code. WORD(mdbcret,1) is the 4-digit return code. WORD(mdbcret,2) is the 4-digit reason code. Please refer to "Command Processor Return and Reason Codes" on page 270.
MDBGDOM	One character with a value of '1' or '0'. A '1' indicates that a message whose ID is stored in MDBGMID is to be deleted, for example due to a preceding DOM (Delete Operator Message) macro or a delete request from the operator.
MDBGHOLD	One character indicating whether the message is a **highlighted** message to be held on the console ('1') or not ('0'). If the message is a response to the REDISPLAY command issued with the HOLD option, MDBGHOLD has a value of '0'.
MDBGJBNM	8-character job name of the VSE/POWER job that issued the message.
MDBGMID	Message ID as an 8-character representation for a 4-byte field of 8 hexadecimal values. If the message is a response to the REDISPLAY command and is not a **highlighted** ("HOLD") message, MDBGHOLD contains a null string; a valid MDBGMID is only returned if that message is a HOLD message.
MDBGDSTP	Year concatenated by day of the year, for example 1995131
MDBGTIME	Time of the day, for example 14:07:26.65

There are situations where the MDB variables are reset to null values. Most often, this happens when variable msgstem.0 has a value of zero.

Examples:

1. Assume you want to find out whether CICS is up and running and, if not, restart it using REXX console automation. Key to this task are retrieval and analysis of the responses to the VTAM command

D NET,APPLS

You also want to wait up to 30 seconds for the complete response.

```
ADDRESS CONSOLE                        /* Establish Console Env        */
```

```
'ACTIVATE NAME REXX PROFILE REXNORC ' /* Activate Console session     */
                                     /* Must be master console to be */
                                     /* able to issue the following  */
                                     /* VTAM command                 */
                                     /*                              */
'D NET,APPLS'                        /* VTAM command shows APPLIDs   */

IF GETMSG(vtam_msg.,'RESP',,,30) = 0  /* Wait for VTAM messages       */

 THEN                                /* Analyse the VTAM messages    */
 DO i=1 TO vtam_msg.0
    pos = INDEX(vtam_msg.i,'PRODCICS')

    IF pos > zero THEN

       IF WORD(SUBSTR(vtam_msg.i,pos),2) ¬= 'ACTIV'  /* CICS down ? */

          THEN 'R RDR,PRODCICS'                    /* Release CICS*/
 END

'DEACTIVATE REXX'                    /* Deactivate console session */
```

2. This example shows how you access an array of MDB variables. Notice that one and the same index is used for the msgstem and for the MDB variables.

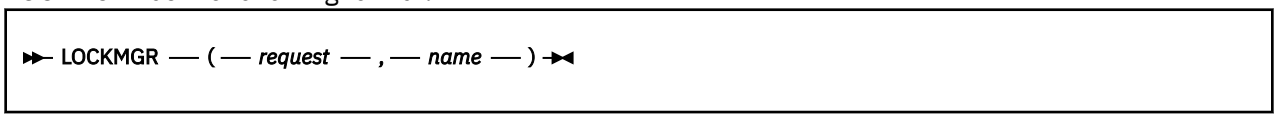
```
 ...
'map'
fc = GETMSG(msg.,'RESP')
DO i=1 TO msg.0
   SAY msg.i
   SAY mdbgtime.i
END
 ...
```

LOCKMGR

The LOCKMGR function allows to serialize a REXX program (or part of a REXX program) by means of a lock/unlock mechanism. Serialization is system wide at task level.

If several REXX programs running under different tasks want to execute the same REXX program, but only one task is allowed at the same time, then you must lock a resource via LOCKMGR('LOCK',name). This causes all other REXX programs that also issue a lock-resource to be set into the wait state until the REXX program unlocks the resource via LOCKMGR('UNLOCK',name).

LOCKMGR has the following format:



request
'LOCK' or 'UNLOCK'

name
8-byte character string that defines the name of the resource to be locked or unlocked. A name that is longer than 8 characters will be truncated to 8 characters. A name that is shorter than 8 characters will be filled with blanks.

Function Codes: The LOCKMGR function returns a function code.

0 Resource was successfully locked/unlocked.
2 The resource is locked already.
4 Resource not available.
8 Some error during lock/unlock. Also, message ARX0565E will be issued.

Invalid input or any other error causes REXX syntax error 40.

Example:

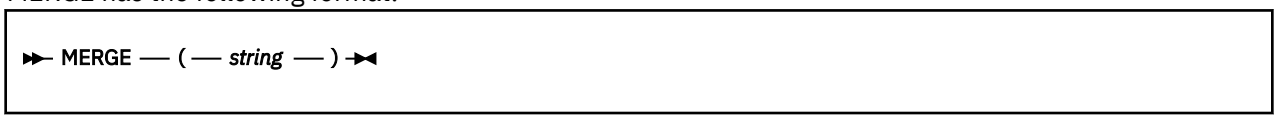
```
rc = LOCKMGR('LOCK',name)   /* Obtain a lock or wait until somebody   */
if rc <= 2 then             /*        unlocks the resource name.      */
```

```
do                          /* We locked the resource and can be sure */
.....                       /* that the DO block of this REXX program */
.....                       /* is executed ONCE in the VSE system.    */
.....
call LOCKMGR 'UNLOCK',name/* Unlock the resource and wake up all    */
end                         /* REXX programs waiting for the          */
                            /* resource to be unlocked                */
```

MERGE

The MERGE function creates a new library member using a skeleton and input variables. The skeleton is a template and contains user-defined variables (placeholders) that are to be filled in with actual data. The MERGE function returns the new library member name.

MERGE has the following format:



where

string
Is a character string that consists of blank-delimited tokens.

```
string = 'token_1 token_2 ... token_n' ['DATA=YES']
```

Each token represents the assignment of a variable. A token is of type varname=value. There are user-defined variables and system variables.

- A value must not contain blanks.
- A user variable can have any variable name.
- System variables have reserved variable names.

System Variables:

INNAME
Library member of type *lib.sublib.mn.mt* that is to be used as template to create a new job (skeleton).

OUTNAME
Library member of type *lib.sublib.mn.mt* that will contain the newly created job.

DATA=YES
Library member specified with OUTNAME, which contains SYSIPT data.

User-Defined Variables: They are identified by two leading and two trailing hyphens. The format is --varName-- When specifying the variable name for the MERGE function, do not include the hyphens (just say varName=).

If a user-defined variable does not receive a value, it will be dropped. This is useful for defining optional operands in JCL.

The MERGE function recognizes continuation characters at position 72 of the skeleton. Continuation lines have to follow JCL conventions.

If the merged text is longer than 71 bytes, the next line begins at position 16. Output lines are thus formatted according to the rules of job control (JCL). However, continued lines may, for formatting reasons, loose blanks in the merge process.

Example: Assume your library DEVLIB.EXEC contains in member SKJOB.Z the following job skeleton.

```
$$$$ JOB JNM=--var001--,CLASS=Y,PRI=8,DISP=D
$$$$ LST CLASS=Q,DISP=D
// JOB --var001--  Job for REXX Console Event Processing
// LIBDEF *,SEARCH=--var002--
// EXEC REXX=--var001--,PARM='REXXCOF5,00780,--var003--,--msg--'
$$/&
$$$$ EOJ
```

After your REXX program has processed the following two statements

```
string = "INNAME=devlib.exec.skjob.z  OUTNAME=devlib.exec.myjob.z",
         "var001=eventest var002=prd1.base var003=prd1.base.skrcjcl.z",
         "msg=HALT_EXIT_REACHED,PRESS_ENTER_TO_END"

CALL MERGE string
```

four variables will have been merged into the skeleton, and library member MYJOB.Z in library DEVLIB.EXEC will contain this job:

```
* $$ JOB JNM=EVENTEST,CLASS=Y,PRI=8,DISP=D
* $$ LST CLASS=Q,DISP=D
// JOB EVENTEST   Job for REXX Console Event Processing
// LIBDEF *,SEARCH=PRD1.BASE
// EXEC REXX=EVENTEST,PARM='REXXCOF5,00780,PRD1.BASE.SKRXJCL.Z,HALT_EXIX
               T_REACHED,PRESS_ENTER_TO_END'
/&
* $$ EOJ
```

Notice that the conventions of the VSE utility DTRIINIT have been used to mask leading characters

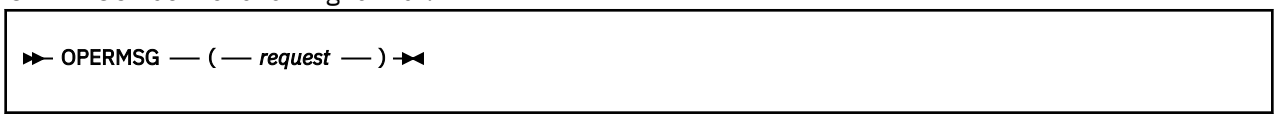
* $$
/*
/&

Error Codes: MERGE does not issue any error codes. Invalid input or any other error causes REXX syntax error 40.

OPERMSG

This function adds or removes an operator communication exit. This exit allows the VSE operator to communicate asynchronously with the REXX program by entering the VSE attention routine command MSG.

OPERMSG has the following format:



The request can be one of the following:

ON
Returns the current state and then activates support for the operator communication exit (OC exit) that is used by a VSE attention routine command

```
MSG nn,DATA=msgdata
```

The following 2-character values of 'msgdata' have a special significance for the processing of the OPERMSG function. When activated, the exit checks for the occurrence of a command **MSG nn,DATA=xx** where

HI
Halt Interpretation

DATA=HI is interpreted as an external interrupt. By preceding the OPERMSG command with the SIGNAL ON HALT command, the REXX program can provide a label to branch to when the HI request is detected.

This setup is useful for controlling a program that is at risk of going into a loop.

TS
Trace Start

DATA=TS simulates the REXX immediate command TS. TS puts the REXX program into interactive debug. This is helpful if a program is looping.

TE
Trace End

HT
Halt Typing

DATA=HT simulates the REXX immediate command HT. This command suppresses output that a program generates.

RT
Resume Typing

For detailed information on the above facilities, please refer to Chapter 10, “REXX/VSE Commands,” on page 143

OFF
Returns the current state and then removes the operator communication exit.

()
Returns the current state (ON or OFF).

MSGDATA
Returns the last **msgdata** which the operator entered via the MSG command. OPERMSG returns the null string if no OC exit request was entered at the VSE console. OPERMSG returns one blank if the "MSG nn" was entered without any message data. Any other string that OPERMSG returns is the message data.

Examples

```
SAY OPERMSG('ON')            /* returns 'OFF' first time called */
                             /* and sets the OC exit support ON */
SAY OPERMSG('MSGDATA')       /* returns the message data        */
SAY OPERMSG()                /* returns 'ON' and keeps it as ON */
SAY OPERMSG('OFF')           /* returns 'ON' and switches to OFF*/
SAY OPERMSG()                /* returns 'OFF'                   */
```

PAUSEMSG

This function issues a console message and waits for an operator reply. If successful, PAUSEMSG returns the operator reply, otherwise a null string.

PAUSEMSG has the following format:

message
Any string of 1 to 122 bytes. A string longer than that will be truncated to 122 bytes.

Examples

```
reply = PAUSEMSG(message)
SAY PAUSEMSG('Please enter your name')
```

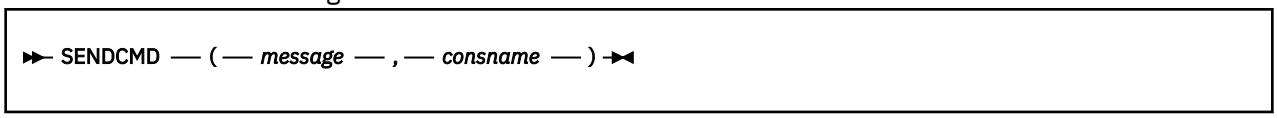
SENDCMD

This function complements the standard VSE console command processing. It allows to issue a VSE console command or a reply. SENDCMD can be used to give a single reply, for example the reply to a // PAUSE statement.

The SENDCMD function does not require an active console session. On the other hand, the target console with the specified (or default) consname must not be active when the SENDCMD function is issued.

Command output is sent to all master consoles that are defined to receive **all messages** ("all routing codes"). There is one exception, however: if the job that started the REXX program specifies an ECHOU user ID, the command output is sent only to the console of that user ID.

SENDCMD has the following format:



message
Any string of 1 to 125 bytes. A string longer than that will be truncated to 125 bytes.

consname
VSE console name.

The default name is 'REXREPLY'.

Note that the *consname* parameter does not determine which console would receive command output. It is only used to identify the command in the hardcopy file and, consequently, in any redisplay output. If your system runs with security active (IPLed with SEC=YES in the SYS command), SENDCMD uses as consname the **VSE security user-id**. Any other consname is ignored.

Function Codes: The SENDCMD issues the following function codes:

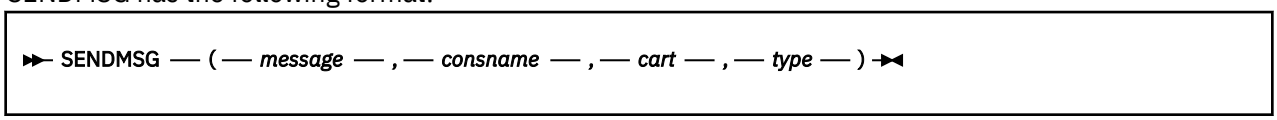
0
SENDCMD processing was successful.

8
SENDCMD function failed due to the failure of a VSE macro. REXX message ARX0565I shows the macro name. Call the SYSVAR function to receive the return and reason code. Refer to "Return and Reason Codes" on page 267 for explanation of these codes.

SENDMSG

This function sends a message to a specific VSE console. No other VSE console will receive the message. Exception: If you specify 'ALL' as *consname*, the message is sent to all active master consoles.

SENDMSG has the following format:



message
Any string of 1 to 125 bytes. A string longer than that will be truncated to 125 bytes.

consname
VSE console name to be used as destination for the message.

The default name is 'REXX'. Specify 'ALL' to send the message to all master consoles active in your VSE/ESA system.

cart
Command and response correlation token. Default is XL8'0'.

The Command And Response Token (CART) provides a selection criteria for the GETMSG function to retrieve one or more particular messages from the set of console messages. Please refer also back to section "Creating a Command and Response Correlation Token (CART)" on page 224.

The 'cart' specification in SENDMSG is only applicable when the message to be sent is a **command response**. It has no meaning for an "ordinary" message such as a notification message. Sending a command response comes into play when you write a command processor in REXX.

type
If HIGH, the message is displayed in highlighted form. If omitted, the message is displayed in non-highlighted form.

Examples: In the following example, the SENDMSG function serves to find out whether the other partition is already active.

```
do while SENDMSG('Hello World','REXX')
    call SLEEP 5
end
```

Sending a highlighted message to all active master consoles is done in the following example:

```
fc = SENDMSG('Hello','ALL',,'HIGH')
```

Function Codes: The SENDMSG function returns a function code:

0
SENDMSG processing was successful.

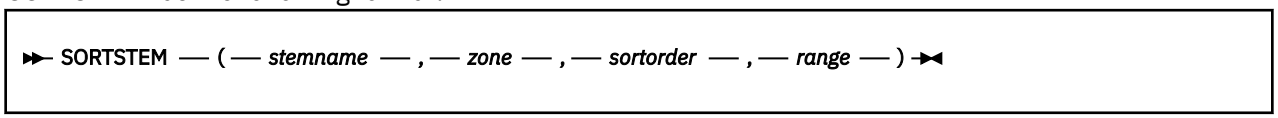
1
The console that was intended as receiver of the message is not active.

Invalid input or any other error causes REXX syntax error 40.

SORTSTEM

This function sorts a stem variable. Sort criteria is the standard "<" -function applied to EBCDIC-strings. SORTSTEM returns function code 0 that replaces the function call.

SORTSTEM has the following format:



stemname
is the stem to be sorted. For example, if *'svar.'* is specified as stemname, variable *svar.0* has to contain the number n of strings to be sorted, and *svar.1*, *svar.2*, ... *svar.n* have to contain the strings to be sorted.

zone
specifies, after the keyword ZONE, a beginning and an ending position within the strings to be sorted identifying the sort criteria. Due to the implemented sort algorithm ("heapsort") the order of "equal" strings may be changed. If not specified, the entire strings are used during comparison. If string lengths do not match, the shorter string is padded with blanks.

sortorder
defines the order of sorting

ASCENDING
means sorting in ascending order. This is the default.

DESCENDING
means sorting in descending order.

range
specifies, after the keyword RANGE, a beginning and an ending index, identifying the part of the stem to be sorted.

Examples of Function Calls

```
fc = SORTSTEM('input.')
fc = SORTSTEM('svar.','ZONE 9 14','DESCENDING','RANGE 11 24')
```

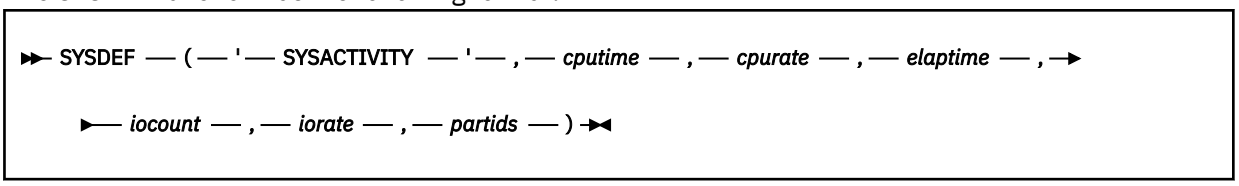
See “Error Codes of Failing Functions” on page 242 for error codes returned by the SORTSTEM function.

SYSDEF

This function is used for two different purposes.

1. It tells the VSE/REXX CPU Monitor which elements of system activity are to be checked. It also sets the limits which, when having been exceeded, lead to a console message.

   The SYSDEF function has the following format:



   where the 5 variables set the limits:

   cputime
   CPU time, in 1/100 seconds, that has accumulated in a partition. A value of 1000, for example, indicates that a partition after having used up 10 seconds of CPU time has reached its limit.

   cpurate
   CPU time as a percentage of elapsed time of the specified interval

   elaptime
   Elapsed time, in seconds, that has accumulated in a partition

   iocount
   number of I/Os that have accumulated in a partition

   iorate
   number of I/Os per second, during the specified interval

   partids
   a string of 2-character partition IDs. It indicates those partitions that are to be excluded from monitoring. Generic notation is allowed: by specifying an '*' in the second position you exclude the entire dynamic partition class from being monitored. For example,

   ```
   'F1F2F3Y*'
   ```

   says that partitions F1, F2, F3, and all partitions of class Y should not be monitored.

   A value of zero for any of the numeric parameters indicates that the corresponding system activity should not be checked.

   If a parameter is specified as nullstring or not specified at all, the corresponding value remains unchanged.

2. Request rerouting from NetView to the REXX console

   This function is of primary interest in the context of *REXX Console Automation*.

   Through the console profile REXAUTO, a REXX console can be defined as receiver of messages that are sent from an automated message handling program such as VSE/OCCF. The SYSDEF function requests that VSE/OCCF reroutes messages that are designated (in the VSE/OCCF message automation table) to be routed to NetView to the REXX console, instead.

   Issue the SYSDEF command as follows:

   ```
   rc = SYSDEF('CONNECT OCCF')
   ```

   To reset the above definition, you have to *disconnect* the REXX console from VSE/OCCF:

   ```
   rc = SYSDEF('DISCONNECT OCCF')
   ```

If your system runs with security active (IPLed with SEC=YES in the SYS command), you can use the SYSDEF function only if the user-id of the **VSE security administrator** was supplied in the job that called the REXX program.

The SYSDEF function returns either of the following:

0
if no values have been set so far and you invoked the function with only the first keyword ('SYSACTIVITY') specified.

values
a string consisting of 6 words. These words contain the values (limits and partition IDs) that were input to the SYSDEF function:

1. CPU time
2. Elapsed time
3. I/O count
4. I/O rate
5. CPU rate
6. String of partition IDs.

2
Applicable only to CONNECT: VSE/OCCF was already connected. This could be a connection to NetView **but remember that you should not use REXX Console Automation and NetView at the same time**.

4
VSE/OCCF has not been started.

If a parameter was excluded in the function call, the corresponding word contains a null value.

SYSVAR

The SYSVAR function described on page "SYSVAR" on page 102 has an argument, SYSERRCODES, that specifically relates to the console environment. SYSERRCODES contains the return and reason code of the VSE system macro (such as MGCRE, MCSOPER, or WTO) which is used to issue a VSE console command.

The following example demonstrates that you cannot call for the service of a command processor while the response to the preceding command has not been retrieved.

```
ADDRESS CONSOLE
'ACTIVATE NAME REXX PROFILE REXNORC'
'RED'
'RED'                              /* REDISPLAY command a second time */
CALL SYSVAR 'syserrcodes'
IF syserrcodes = '0008 0001'
   THEN SAY 'Command not accepted because command processor is busy'
```

Return and reason codes are returned as decimal values.

Please refer to "Return and Reason Codes" on page 267 for explanations of all return and reason codes issued from VSE/ESA system macros.

Error Codes of Failing Functions

The following message indicates an error

```
ARX0960E ERROR Running Function xxxxxxxx, RC=nn
```

RC=nn points to a specific error. Possible error codes, together with an explanation, are shown in the following table.

Function Name	Error Code RC	Explanation
DELMSG	04	Invalid number of arguments, or nullstring as argument
	12	Invalid parameter list

Function Name	Error Code RC	Explanation
FINDMSG	08	Failure of VSE macro. REXX message ARX0565I shows the macro name. Call SYSVAR to determine return and reason code. Refer to “Return and Reason Codes” on page 267.
	12	No active console session.
	16	Console session deactivated during FINDMSG processing
	20	ARXEXCOM error or stem count error
	24	Unused
	28	Invalid library member in connection with LOADACTN option
	29	Unused
	32	Invalid parameter list
	36	Invalid TIME
	40	Invalid ZONE
	44	Unused
	48	Message action table not loaded
	52	Problem loading library member
	56	Library member not found
	60	Storage problem
	64	Invalid OPTION
	68	Format error within message action table
GETMSG	04	Invalid number of arguments
	12	Invalid parameter list
LOCKMGR	04	Invalid number of arguments
	12	Invalid parameter list
	20	EXECIO error
MERGE	24	Empty input string
	28	EXECIO DISKR error
	30	Invalid parameter list (invalid filename for INNAME or OUTNAME)
	32	Invalid parameter list (either no parameter or more than one parameter)
	34	Invalid token
	36	EXECIO DISKW open error
	40	EXECIO DISKW error (unable to write)
	44	EXECIO DISKW close error
OPERMSG	04	Invalid number of arguments
	08	STXIT macro failed
	12	Invalid parameter list

Function Name	Error Code RC	Explanation
	16	Storage problem
	24	CDLOAD failed
	28	STXIT OC not activated
PAUSEMSG	04	Invalid number of arguments
	08	WTOR error
	12	nullstring as argument
SENDCMD	04	Invalid number of arguments
	12	Invalid parameter list
SENDMSG	04	Invalid number of arguments
	08	WTO macro failed
	12	Invalid parameter list
SORTSTEM	04	Invalid number of arguments.
	08	Invalid ZONE.
	12	Invalid STEM variable.
	16	Storage problem.
	20	Stem handling problem.
	24	Invalid STEM.0 setting.
	28	Invalid sort order
	32	Invalid range
SYSDEF	04	Invalid number of arguments
	12	Invalid parameter list
	16	REXX not initialized
	20	Security violation

It may happen that module ARXEFPLX issues an RC=4. Most likely, this indicates a shortage of GETVIS storage. In rare cases, the setting of variable pools failed.

For error return codes issued by VSE system macros, please refer to

- Section “Return and Reason Codes” on page 267 -- for macros MCSOPER, MCSOPMSG, MGCRE,
- z/VSE System Macros Reference -- for other macros such as CDLOAD or WTO.

Always Keep in Mind...

This section contains important hints and tips that you should always be aware of when working with *REXX Console Automation*.

The entire message traffic between programs and consoles goes through the **Console Router queue**. When a message enters the Console Router queue, the target console(s) are posted to retrieve the message. Only after all target consoles have retrieved the message, the queue space occupied by this message is eligible to be reused for a new message.

Space for the Console Router queue is not unlimited. Therefore, **if messages are not retrieved at the target consoles**, the queue can become flooded. In the worst case, this could lead to a complete

standstill of the system. To take precautions against this, a console is suspended if no message is retrieved within the next 15 seconds after arrival of a message at this console. If a console is suspended, no more messages are routed to this console. A REXX console procedure would have to DEACTIVATE and re-ACTIVATE the console in this case, but incoming messages during suspend are lost.

By observing the following rules, you will be able to avoid such critical situation.

Note that the CORCMD debugging command is available for situations where your console functions do not work as they are supposed to. Please refer to “CORCMD Command for Problem Solving” on page 271.

Make Frequent Use of the GETMSG Function

A prominent task of a console program is to retrieve the messages that are sent to the console. The console program can be a console presentation program such as the system console, or it can be a programmed operator.

In *REXX Console Automation*, you retrieve messages by using the GETMSG function. You are urged to frequently call this function. This helps to have message items disappear from the Console Router queue.

Do not Send Messages to "Yourself"

After your REXX program has activated a REXX console with profile REXALLRC ("receive all routing codes"), it should not send messages to this console. Specifically, it should not issue

SAY to SYSLOG
REXX TRACE to SYSLOG

A sequence like the following is deadly.

```
ADDRESS CONSOLE 'ACTIVATE ... PROFILE REXALLRC'
  DO FOREVER

    GETMSG...
    SAY...

    SAY...

  END
```

Your program is not able to retrieve messages as fast as it sends them.

Redirect Some Output to SYSLST

A way to avoid the above situation is to direct SAY output and trace output to SYSLST:

```
CALL ASSGN 'STDOUT','SYSLST'
```

Direct Messages to Only One Console (ECHOU Option)

Another way of directing high-volume message traffic away from the REXX console is to make sure that it is **not routed to all consoles**. This is accomplished by including the VSE/POWER option ECHOU in the job that calls your REXX program.

The ECHOU option prevents the job's console output from being delivered to all master consoles (one of them being the REXX console which had been activated with the REXALLRC profile). Instead, the output is sent to the console that is associated with the ECHOU user-id. Your REXX console is thus freed from debug and tracing messages; it receives only its own, very specific traffic of commands and command responses.

An effect similar to the ECHOU option is achieved by starting your REXX program at a master console via a

```
r rdr,pausexx
```

job. An ECHOU option for this master console has been made effective automatically by the VSE system.

Remember the REXNORC Profile

The discussion so far had assumed that the REXX console is activated with profile REXALLRC ("receive all routing codes"). This profile is useful for global systemwide monitoring.

You should ask yourself whether the REXX console, at a particular place in your REXX program, would do better with the REXNORC ("receive no routing codes") profile. A console which needs to receive only its command responses should be defined with that profile. For example, in a scenario like the following

```
ACTIVATE ... PROFILE REXNORC
...
   REDISPLAY...

DEACTIVATE ...
```

the REXX console is reserved for receiving only the REDISPLAY output. If instead it is activated with profile REXALLRC, a lot of other messages can pile up at this console while the REDISPLAY command processor is still searching through the hardcopy file.

Split off a Time-consuming Task into a Separate Job

Ask yourself how much synchronous processing you can afford between two incidents of message retrieval.

If there is a good chance that this processing may last a long time (for example waiting for a certain event to happen, or a fair amount of I/O operations), you might be better off if you let a separate job perform **asynchronously** that piece of processing.

You either use the PUTQE command directly. Or you use an approach that is shown in one of the demo applications (see “REXXSPCE” on page 256) where the REXXCO application triggers a job via the **REXX Message Action Table**. In doing so, REXXCO also issues the PUTQE command.

Finish All Preparatory Work Prior to ACTIVATE CONSOLE

This is a similar line of thought as in the preceding section. Your preparatory work might involve time-consuming tasks such as reading and processing of files.

Place these tasks ahead of ACTIVATEing your REXX console. If, instead, these tasks are started after ACTIVATE and take a long time to finish, they might hold up your REXX program and prevent it from retrieving all other messages that keep accumulating.

Similarly, DEACTIVATE your REXX console before you start time-consuming cleanup tasks.

Handle One Command at a Time

It is advisable to finish processing of one command prior to issuing the next command. You should thus adhere to the following sequence:

```
. . .
issue command
retrieve and process response
issue command
retrieve and process response
. . .
```

If you fail to do that, you run the risk of getting a 'COMMAND PROCESSOR BUSY' condition. This is indicated by a return code of -10 from the REXX host command routine plus return and reason codes 8 / 1 from the MGCRE macro.

Start Testing on a Small Scale

This involves two rules:

1. Start testing your REXX console automation program in a test environment before you release it to your production system.
2. At the earlier stages of testing, let a REXX master console run with profile REXNORC ("receive no routing codes"). This helps to isolate the automated handling of commands and associated responses.

The Most Important Rule...

At the end of this section, the two most important rules are summarized for you to remember:

- Let your REXX program **retrieve the messages** that were delivered at the REXX console.
- **Deactivate a REXX console** that is no longer being used.

Also check the priority of the partition running your REXX console program. Keep in mind that the Console Router program, whose queue space may soon be exhausted, could bring the entire system to a standstill.

REXX/VSE CPU Monitor

The VSE/REXX CPU Monitor is a CICS program that checks for critical performance values in VSE partitions. It issues a console message when it detects that user-defined limits have been exceeded. The user sets these limits using the **SYSDEF** function described on “SYSDEF” on page 240.

The CPU Monitor is called as a user exit from **CICS transaction IEXM**, a CICS background transaction. This is described in the IBM CICS Transaction Server for z/VSE Enhancements Guide. The CPU monitor analyzes the data that the CICS transaction recorded up to the last measurement interval. It checks whether in any partition any of the limits set by function SYSDEF have been exceeded. If this is the case, the program sends message ARX0998I to the console, for example

```
ARX0998I PID Y1 JOB TEST EXCEEDS THE LIMITS: CPUTIME=10.15
```

Note that the message may show more than one critical value. This depends on how many of the five system activity items exceeded their limits.

The entire message can appear more than once within a measurement interval: for each partition that went over its limit you get one such message.

Using the REXX Console Automation function, you can provide automated actions in response to the above message. For example, you may cancel a critical partition that is using up system resources. The REXX Console Automation function includes an application example that provides a mechanism for capturing an event as indicated by the above message. You find statements for handling the event in member REXXEVNT.Z at label FLUSHCPU.

The REXXCO application framework is described in the following section “REXX Console Application Framework” on page 247.

REXX program REXXCPUM is an example of how to make best use of transaction IEXM in conjunction with the SYSDEF function and the CPU Monitor. See “REXXCPUM” on page 259 for a description of REXXCPUM.

REXX Console Application Framework

An example application framework (its name is REXXCO) is available to demonstrate how you can exploit *REXX Console Automation* to your best advantage. First some typical operation scenarios are described which are addressed with this framework. Then its concept is briefly discussed. The examples are presented in detail further below, in section “Automated Operation Demos (Examples)” on page 253.

Operation Scenarios

- Suppose the operator has to flush jobs that misbehave, for example jobs that produce too much output. The operator would notice this through VSE/POWER message

  1Q52I OUTPUT LIMIT EXCEEDED...

  This task can be automated. Job REXXFLSH demonstrates this; see “REXXFLSH” on page 255.

- Suppose the operator has to conduct a defined dialog with an application. For example the operator enters some data via the operator communication facility or answers an outstanding reply.

  This task can be automated. Job REXXCXIT demonstrates this; see “REXXCXIT” on page 255.
- Suppose the operator has to schedule a job sequence according to clues taken from console messages. A more complex example is given with REXXSPCE; see “REXXSPCE” on page 256. Jobs request work file space via messages to the console. These requests are met in an automated way through the REXXCO application framework.

Concept

The REXXCO application framework maintains an active REXX console. It is designed to monitor all messages that are available at this console.

Jobs issue messages for various purposes. In a given installation, some of these messages may trigger a request to perform a certain function. These messages are recognizable by given character strings within the message text. This can be the message ID up front, but also any substring inside the message text.

The REXXCO application framework is a programmed operator who finds those messages and automatically initiates a predefined action.

The application is designed to be highly tailorable so that it can be adapted to various needs. The tailoring is done on two levels:

1. Definition of message - action pairs

   A message is identified by a search string, the action by a name. This definition has the form of a table and is called **Message Action Table**. You can build your own table, but you may also work with the IBM-supplied table which you find in your system under the name REXXTABL.Z.
2. The actions can be one of three programmed responses:
   - Internal REXX program
   - External REXX program
   - VSE job.

   This is described below, in section “Actions” on page 249.

Message Action Table Entries

An entry in this table contains the following information:

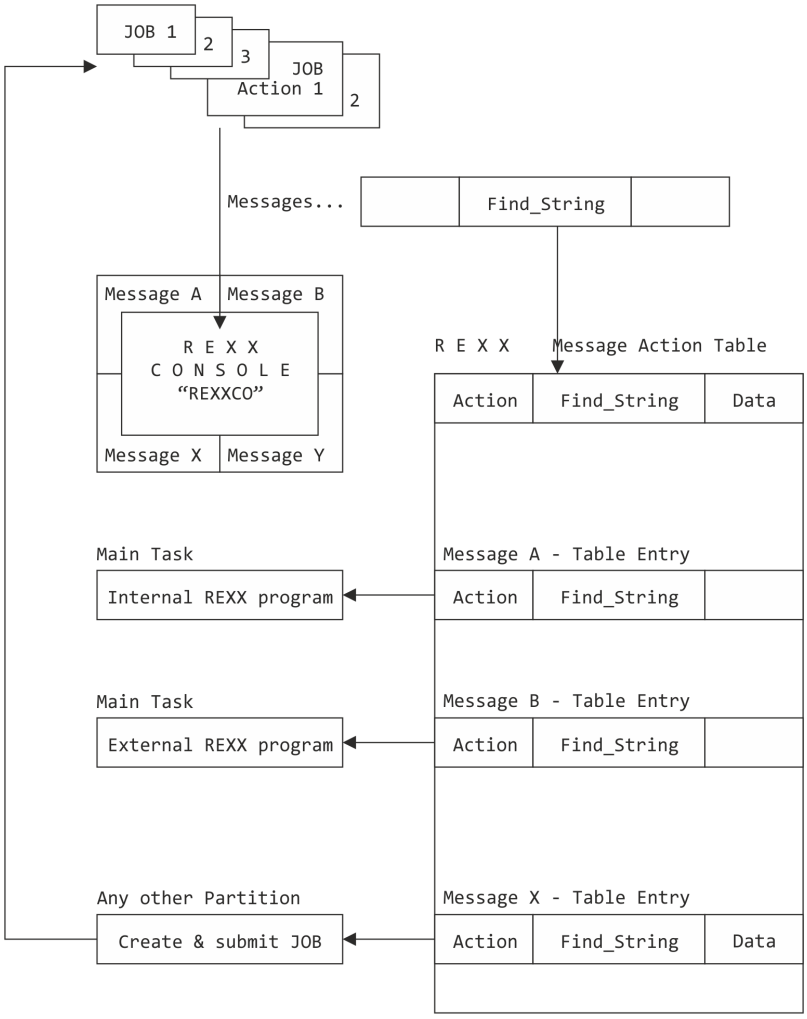
Find_String
: This string extends from the second word through position 71 (trailing blanks are not considered). It serves as search argument to select from the set of console messages those that represent a **functional request**.

Action
: This is written in the first word of an entry. It may be used to initiate an action (*function*) to be taken after a match of Find_String has been found in a console message.

Data
: This can be any character string and is (optional) additional information for the function that performs *Action*. It is recognizable in the Message Action Table as **continuation line** (the preceding line has a continuation character in position 72).

Note: In this context, a match between an entry in the Message Action Table and a console message is called an **event**.

The interplay between messages coming in at the console and entries in the Message Action Table is illustrated in the following figure.



Actions

As you can see from the above figure, in the REXXCO application **Action** can be performed by three kinds of units:

- Internal REXX Program

  This is one out of a set of REXX programs (also called *REXX procedures*) each of which is capable of providing the appropriate action for an event.

  None of these programs is a complete self-contained program. Instead, this type of program consists of just a sequence of REXX statements. These sequences (*procedures*) are collected together into a larger REXX program. The procedure is uniquely identified by a label. The label corresponds to the first word (Action) in a Message Action Table entry.

- External REXX program

  In the Message Action Table, this is indicated by an ampersand (&) in position 1. Behind the &, beginning in column 2, is the name of the program. It will be called as (external) function:

  ```
  rc = progname(..parameters..)
  ```

- A VSE job

  This is indicated by just an asterisk (*) in position 1, without any name following. The REXXCO application creates a new batch job using a given job skeleton and submits this job to the VSE/POWER reader queue.

To create the job, the application uses the MERGE function (see “MERGE” on page 94). The input string for MERGE consists of continuation lines (also called **data lines**) in the table entry. These lines must at least supply values for the system variables INNAME and OUTNAME of the MERGE function. In addition, these data lines can assign values to user-supplied variables in the skeleton.

For an illustration of the three types of action specification, you find below the Message Action Table that is supplied by default in your system.

```
/* REXXTABL REXX/VSE Message-Action-Table for Demonstration Purposes */
/* Action   FindString(2nd word..pos 71)  Data,in continuation lines */
FLUSH1Q52I  1Q52I
FLUSHCPU    ARX0998I
REPLY       HIT ENTER TO CLOSE THE FILE
REPLY       HALT EXIT REACHED, PRESS ENTER TO END
&REXXSTOP   THIS LOOP CAN BE STOPPED
*           REXXCO<< VSAM WORKFILE:                                    X
            INNAME=PRD1.BASE.SKRXVSAM.Z OUTNAME=PRD1.BASE.REXXSPCE.JOB X
            VAR001=REXXSPCE VAR002=PRD1.BASE VAR003=PRD1.BASE.SKRXJCL.Z
*           0D20E                                                      X
            INNAME=PRD1.BASE.SKRXPTLG.Z OUTNAME=PRD1.BASE.REXXPTLG.JOB X
            VAR001=REXXPTLG
```

Figure 13. Example of a Message Action Table

How to Use the REXX Console Application REXXCO

Loading

The REXXCO application must first be loaded as a member of type PROC. You do this by using the REXXLOAD program. This program is described in section “REXXLOAD” on page 253.

Invocation

You start the application from a job by supplying the statement

// EXEC REXX=REXXCO,PARM='p1 p2...pn'

Your system includes job REXXCONS which activates a console session and then starts the REXXCO application.

The PARM operand allows you to override several names that are used by the REXXCO application (there is no checking for valid names). PARM has the following parameters. Everyone of them is coded as p=value.

MSGTABLE=
Fully qualified library member name (lib.sublib.mn.mt) of the Message Action Table. The REXXCO application uses PRD1.BASE.REXXTABL.Z as default.

EVENTPROC=
Name of a REXX program that is called when Action is provided by an **internal REXX program** (Action in the Message Action Table has neither an '&' nor an '*' in postion 1, but rather the label of that internal REXX program).

The REXX program named by EVENTPROC has one or more of internal REXX programs bundled together. Each of them is uniquely identified by a label, and this label corresponds to an Action in the Message Action Table.

The REXXCO application uses PRD1.BASE.REXXEVNT.PROC as default. Please refer also to “User-Supplied REXX Action Program” on page 251.

CONSNAME=
The 8-byte unique console name. If not specified it defaults to REXXCOnn, where nn is the partition id.

PROFNAME=
The 8-byte console profile name. If not specified it defaults to REXALLRC.

LOCKID=
An 8-byte name to be used by the LOCKMGR function. It will be passed to an Action job or to an event-handling REXX program. If not specified it defaults to REXXCOnn, where nn is the partition id.

Note that you may call REXXCO in multiple partitions. This would allow you to supply different tables, event-handling Action programs etc.

Termination

You terminate the REXXCO application by entering at the console the command

MSG part_ID,DATA=EXIT or **MSG part_ID,DATA=HI**

Event

An event takes place if a console message retrieved by the REXXCO application contains the Find_String (as substring) specified in a REXX Message Action Table.

User-Supplied REXX Action Program

As was described earlier, the REXXCO application calls a REXX program to perform an action (unless Action in the Message Action Table shows an '*'). The program is either a complete REXX program, as indicated by an '&' in the first position in the table. Or it is the REXX program you had specified in the parameter EVENTPROC. This is a set of internal REXX programs each of them uniquely identified by a label.

You might access REXX statements at the given label as shown below (an excerpt from member REXXEVNT.PROC).

The first statement, ARG, serves as interface between REXXCO and the REXX program defined in the EVENTPROC parameter. Also, any external Action program has to apply this interface.

```
ARG lockid,message,tblentry
/*
lockid    is the lockid you should use for the LOCKMGR function.
message   is the VSE console message
tblentry  is the matching table entry of REXX console message table

Your REXX program may proceed as follows:
*/
 ...
action  = WORD(tblentry,1)        /* Extract Action from table entry */
SIGNAL action
 ...
 ...
LABEL1:  /* This code will get control if action='LABEL1' */
 ...     /* Here you process the VSE console message      */
RETURN
```

User-supplied Job Skeletons

You can create your own job skeleton. This is a VSE library member whose name you supply in the INNAME= parameter of the message action table. It may contain variables to be resolved.

Here is an example of a job skeleton that contains variables to be resolved by the MERGE function.

```
$$$$ JOB JNM=--var001--,CLASS=Y,PRI=8,DISP=D
$$$$ LST CLASS=Q,DISP=D
// JOB --var001--  Job for REXX Console Event Processing
// LIBDEF *,SEARCH=--var002--
// EXEC REXX=--var001--,PARM='--rexxlock--,--rexxjnum--,--var003--,--reX
              xxmsg--'
$$/&
$$$$ EOJ
/+
```

Figure 14. Example of a Job Skeleton

Variable Resolution within Job Skeletons

Values for user-supplied variables in a job skeleton are represented as

```
--varNNN--
```

In order to have the variables resolved by the REXXCO application, you have to assign data to the variables. You do this via the continuation lines (*data lines*) in the REXX Message Action Table entry. For example,

```
  *     any substring of a console message to be found                X
        INNAME=PRD1.BASE.SKRXVSAM.Z OUTNAME=PRD1.BASE.NEWJOB.Z        X
        VAR001=EVENTEST VAR002=PRD1.BASE VAR003=PRD1.BASE.SKRXJCL.Z
```

The INNAME parameter specifies the fully qualified name of the job skeleton. The OUTNAME parameter specifies the fully qualified name of the output library member. After the variables are merged into the skeleton, the output library member PRD1.BASE.NEWJOB.Z looks as shown below.

```
* $$ JOB JNM=EVENTEST,CLASS=Y,PRI=8,DISP=D
* $$ LST CLASS=Q,DISP=D
// JOB EVENTEST   Job for REXX Console Event Processing
// LIBDEF *,SEARCH=PRD1.BASE
// EXEC REXX=EVENTEST,PARM='REXXCOF5,00780,PRD1.BASE.SKRXJCL.Z,F5 0005 X
               * REXXCO<< VSAM WORKFILE: TRACKS=5'
/&
* $$ EOJ
```

Please note that the following variables are generated by the REXXCO application and are concatenated implicitly to the data line.

Reserved Variables	Examples
REXXPID	F5
REXXRPLY	0005
REXXLOCK	REXXCOF5
REXXJNUM	00780
REXXMSG	F5 0005 * REXXCO<< VSAM WORKFILE: TRACKS=5 (the first 76 characters)

Error Handling

The REXXCO application provides only limited error handling. Two execution phases must be distinguished:

1. Initialization

   This phase lasts until a console is activated. Error conditions during this phases are reported via error codes 12, 16, 24 and 28 (see below) and cause the program to be terminated.

2. Message Monitoring

   If an Action program fails and returns a nonzero error code, REXXCO goes into a controlled termination with error code 8.

   Syntax errors are not trapped. Any syntax error causes immediate termination at the point where the syntax error occurred, for example when SENDCMD gives a reply where there is no outstanding reply ("REXX syntax error 40 - invalid call to routine... ").

   Any other error conditions are not handled. Proper error handling can only come from the Action program itself.

REXXCO returns the following

Error Codes:

8
An (internal or external) Action terminated with a nonzero return code.

12
Invalid message ID.

16
Invalid parameter format.

24
SYSVAR failed.

28
ADDRESS CONSOLE failed.

Automated Operation Demos (Examples)

REXX Console Automation, as delivered to you, has several ready-to-run jobs and REXX programs that demonstrate the versatility and simplicity of implementing an automated, or programmed, VSE/ESA console.

Three of these demo programs

REXXFLSH
REXXCXIT
REXXSPCE

are practical applications of the concept presented in the preceding sections. They are described below with some detail. The other programs address a great variety of operation tasks:

REXXCPUM
Shows how to make best use of transaction IEXM in conjunction with the SYSDEF function and the CPU monitor

REXXDOM
Shows the manipulation of messages

REXXTRY
Provides a dialog with the operator to issue REXX commands

REXXJMGR
(this and the following) Manages VSE/POWER jobs

REXXWAIT
(this and the preceding one) Manages VSE/POWER jobs

REXXASM
(this and the following) Controls the interplay between JCL, Assembler, Linkage Editor and Librarian

SETSDL
(this and the preceding one) Controls the interplay between JCL, Assembler, Linkage Editor and Librarian

This group of programs will be presented more or less in an overview fashion. For detailed information you are advised to look at the actual source code which has ample commentary on the use and functionality of the particular program.

Before you start using any of the demo units, you have to load them into a predefined place using the REXXLOAD program. This is described in the following section.

REXXLOAD

This REXX program phase copies library members that are shipped to you into

- A member of type PROC - if the library member contains a REXX procedure

- The VSE/POWER reader queue - if the library member contains a job.

The program is driven by a special member which specifies the source library member as first word of each record. The second word of the record specifies the target where the library member should be copied to: either

- The keyword RDRCLASS=x to indicate that the library member contains a job to be transferred into the VSE/POWER reader queue, or
- The name of a PROC type member.

The special member might contain records like the following:

```
PRD1.BASE.REXXCXIT.Z      RDRCLASS=Y
PRD1.BASE.REXXFLSH.Z      RDRCLASS=Y
PRD1.BASE.REXXDOM.Z       PRD1.BASE.DOM.PROC
PRD1.BASE.SETSDL.Z        PRD1.BASE.SETSDL.PROC
PRD1.BASE.REXXSAA.Z       PRD1.BASE.REXXTRY.PROC
PRD1.BASE.REXXCO.Z        PRD1.BASE.REXXCO.PROC
```

Your system already has such a member ready to be processed, its name is PRD1.BASE.REXXZBK.Z. You may create your own member. If you give it a different name, you have to pass this name as a PARM argument when you call REXXLOAD.

Invocation

You start the program from a PAUSExx job, for example PAUSEBG. The partition must have 1MB of GETVIS storage available. The GETVIS requirement can even be higher if very large members are to be copied.

```
r rdr,pausebg
```

Then enter at the console

```
0 // LIBDEF *,SEARCH=PRD1.BASE
0 // EXEC REXXLOAD[,PARM='l.s.mn.mt']
```

The program now copies the library members one by one to their targets, according to the specifications given in the special member. After each successful copy operation, a message is issued telling which member has just been processed.

Error Conditions

All error conditions are handled via routine ARXERROR which issues message ARX0960E. The following error codes may show up:

100+RC
where RC was returned from EXECIO

200+ABS(RC)
where RC was returned from PUTQE (can be negative)

7
Invalid member specification in PARM.

8
* $$ JOB statement is missing in source VSE/POWER job.

9
Incorrect member specification in special member.

If REXX has not been initialized, REXXLOAD fails with RC 4094.

Note: Running REXXLOAD multiple times results in duplicate job entries in the VSE/POWER reader queue. You might want to delete the superfluous jobs.

REXXFLSH

This demo addresses the first operation scenario that was presented in section “Operation Scenarios” on page 247. It contains the implementation of an **Action Job** and an **internal Action** program.

Scenario

Suppose the operator has to flush jobs that misbehave, for example jobs that produce too much output. The operator would notice this through VSE/POWER message

1Q52I OUTPUT LIMIT EXCEEDED...

This task can be automated. Job REXXFLSH demonstrates this.

Running the Demo

(Before you start make sure that you loaded the necessary jobs and program modules using REXXLOAD; see “REXXLOAD” on page 253.)

The first step is to start the REXXCO application which activates a REXX console. You do this by entering the VSE/POWER command

```
r rdr,REXXCONS
```

The job REXXCONS calls the REXXCO application by using an EXEC statement as described in section “Invocation” on page 250. Now REXXCO starts watching out for message 1Q52I.

As a second step you start the job REXXFLSH

```
r rdr,rexxflsh
```

REXXFLSH is a job that issues (simulates the occurrence of) the VSE/POWER messages

```
0D20E HARDCOPY FILE SHOULD BE PRINTED
1Q52I OUTPUT LIMIT EXCEEDED FOR REXXFLSH 00987 xx, 180
```

A PRINTLOG is started automatically. The partition xx where the job runs is then PFLUSHed by the console application.

Background Information

The Message Action Table has entries

```
FLUSH1Q52I  1Q52I
*           0D20E                                                    X
            INNAME=PRD1.BASE.SKRXPTLG.Z OUTNAME=PRD1.BASE.REXXPTLG.JOB X
            VAR001=REXXPTLG
```

When REXXFLSH issues its message 0D20E, REXXCO builds job REXXPRTL by using job skeleton SKRXPTLG (please refer to the INNAME parameter in the Message Action Table). The job name REXXPRTL is taken from variable VAR001 in the Message Action Table. REXXCO submits job REXXPRTL to the VSE/POWER reader queue to invoke the PRINTLOG utility. When REXXFLSH issues its message 1Q52I, REXXCO calls the REXX program REXXEVNT.PROC which has an internal REXX procedure at label FLUSH1Q52I. This procedure initiates the PFLUSH command.

REXXCXIT

This demo addresses the second operation scenario that was presented in section “Operation Scenarios” on page 247. It contains the implementation of an **external Action** program.

Scenario

Suppose the operator has to lead a defined dialog with an application. For example the operator enters some data via the operator communication facility or answers an outstanding reply.

This task can be automated as demonstrated by job REXXCXIT.

Running the Demo

(Before you start make sure that you loaded the necessary jobs and program modules using REXXLOAD; see “REXXLOAD” on page 253.)

The first step is to start the REXXCO application which activates a REXX console. You do this by entering the VSE/POWER command

```
r rdr,REXXCONS
```

The job REXXCONS calls the REXXCO application by using an EXEC statement as described in section “Invocation” on page 250. Now REXXCO starts watching out for, among several other strings, for the string 'THIS LOOP CAN BE STOPPED'.

As a second step you start the job REXXCXIT

```
r rdr,rexxcxit
```

The job REXXCXIT issues the message

```
THIS LOOP CAN BE STOPPED BY: MSG 'SYSPID', DATA=HI
```

The message indicates that the REXX program is in a loop and that the loop can be interrupted by an operator communication exit.

The string 'THIS LOOP CAN BE STOPPED' triggers the REXXCO application to stop the loop. REXXCXIT then issues a message with an open reply

```
***** HALT EXIT REACHED, PRESS ENTER TO END
```

The REXXCO application gives the reply.

Background information

REXXCXIT starts the operator communication exit via OPERMSG(ON). It defines a trap for the HALT condition (SIGNAL ON HALT). The REXXCO application recognizes the message

```
THIS LOOP CAN BE STOPPED BY: MSG 'SYSPID', DATA=HI
```

in the Message Action Table and calls an *external* REXX Action program. This program issues the operator command

```
MSG xx,DATA=HI
```

HI ("Halt Interpretation") causes REXXCXIT to end its loop and to issue the open reply message. Again, the REXXCO application recognizes the message. Via the Message Action Table, an *internal* REXX program at label REPLY in REXXEVNT.PROC gets control and gives a reply (any reply will do). This leads to a normal EXIT from REXXCXIT.

REXXSPCE

This is another, more elaborate demo of what the REXXCO console application is able to achieve. It contains the implementation of an **Action Job**.

Scenario

In this demo, the REXXCO application monitors incoming messages of three jobs: REXXVSM1, REXXVSM2 and REXXVSM3. Each job attempts to write 5 tracks worth of data into VSAM work files. The jobs do not know if and where this workfile space is available, and they don't care. All they know is the **amount** of space they need. They communicate this need by way of a message.

A real-life operator normally would be able to determine how much, and where workfile space is available, and then enter the necessary JCL information. The REXXSPCE application shows how this task can be automated.

Before Starting...

Remember that before you invoke the program you should have loaded the necessary jobs and program modules using the REXX program REXXLOAD (see “REXXLOAD” on page 253). Also, in order to run the demo with REXXVSM1/2/3, you need at least 7 dynamic partitions of class Y. The class Y needs a minimum of 1M allocation space. Approximately 800K of this allocation are needed as GETVIS storage.

Running the Application

The first step in starting the REXXCO application is to activate a REXX console. You do this by entering the VSE/POWER command

```
r rdr,REXXCONS
```

The job REXXCONS calls the REXXCO application by using an EXEC statement as described in section “Invocation” on page 250. Now REXXCO starts watching out for specific messages issued by REXXVSMn jobs.

You now start these REXXVSMn jobs by entering

```
r rdr,rexxvsm*
```

Make sure that you have only one copy of each job REXXVMS1/2/3 in the reader queue. Delete duplicate job entries if there are any. Otherwise the demo would not function properly because very likely partition class Y has not enough partitions available.

Please note that you could start only one of the REXXVSMn jobs and still get the benefit of the demo.

Handling REXXVSMn Messages

Any one of the REXXVSMn jobs calls the REXX program REXXVSAM. This program stops itself by issuing the statement

```
Y2-0046 // PAUSE REXXCO<< VSAM WORKFILE: WRQST=ALLOC TRACKS=5
```

(Partition ID and reply ID 'Y2-0046' are arbitrary example values.) By looking at the Message Action Table shown in Figure 13 on page 250 you will find the substring within the above message

```
REXXCO<< VSAM WORKFILE:
```

as a Find_String in the table. The corresponding Action column shows an *. This indicates that a job is to be built and submitted to VSE/POWER.

REXXCO does just that: it builds job REXXSPCE by using the job skeleton SKRXVSAM (please refer to the INNAME parameter in the Message Action Table). The job name REXXSPCE and the name of the program to be invoked, again REXXSPCE, are taken from variable VAR001 in the Message Action Table.

REXXCO submits job REXXSPCE to the VSE/POWER reader queue. When the job starts running, it calls program REXXSPCE.

Creating DLBL/EXTENT Statements

The REXXSPCE program finds out whether a workfile of required size is available. It uses information from the given message (WRQST=ALLOC and TRACKS=5) and also from member REXXCNTL.Z.

For each system resource used by the job(s), for example a work file, there is an entry in member REXXCNTL.Z. It records system-dependent information like VSAM master catalog or volume and maximum number of tracks. The master catalog is assumed to reside on the indicated volume.

Here is an example entry:

```
/* ------------------------------------------------------------- */
/* Resouce Id   Used   System related info                       */
/* ------------------------------------------------------------- */
VSAM_WORKFILE   *      CAT=REXX.VSAM.CAT VOL=SYSWK1 MAXTRACKS=7
```

Assuming for example that at the beginning 7 tracks are available (MAXTRACKS=7), the request of job REXXVMS1 can be satisfied.

Note: This demo uses an oversimplified method of computing and bookkeeping of available workfile space. It is there for demo purposes only and should not be considered as applicable for real-life production systems.

If not enough work space is available, the job has to wait until a free resource is available. Otherwise the REXXSPCE program builds appropriate JCL statements by using skeleton SKRXJCL.Z as a base. The skeleton's name was supplied via VAR003 in the Message Action Table (see Figure 13 on page 250) and passed as a parameter (see Figure 14 on page 251). Program REXXSPCE also uses information from member REXXCNTL.Z to merge into the skeleton.

As a result, REXXSPCE creates a JCL procedure that contains the necessary DLBL and EXTENT information. The name of the procedure is ALLOCnn where nn is the reply ID of the waiting job, for example ALLOC46. The JCL procedure is started by sending a (programmed) reply to the REXXVSMn job:

```
46 // EXEC PROC=ALLOC46
```

Writing into and Freeing up Work Space

Job REXXVSM1 ends its pause and writes something into the work file. When done, it issues the message

```
Y2 0046  REXXVSM1  HIT ENTER TO CLOSE THE FILE
```

By looking at the Message Action Table shown in Figure 13 on page 250 you will find the substring within the above message

```
HIT ENTER TO CLOSE THE FILE
```

as a Find_String in the table. The corresponding Action column says REPLY which points to a REXX procedure within member REXXEVNT.Z. Job REXXVSM1 continues and requests that the workfile space should be freed:

```
Y2 0046 * REXXCO<<VSAM WORKFILE: WRQST=FREE TRACKS=5
```

Again, a new REXXSPCE job is built and submitted to VSE/POWER. The job frees up 5 tracks of workfile space. It also computes and records the new value of available tracks.

The flow of events that was shown for one job should really be thought as a process of multiple jobs running or waiting. Assuming for example that the original amount of available tracks is 7, job REXXVSM2 has to wait until REXXVSM1 frees its 5 tracks.

Summary Listing of Demo Parts

The demo consists of the following parts:

-- Demo Jobs --

REXXVSMx.Z, x=1,2,3
"Customer's" batch job. Calls REXXVSAM.PROC. This is the source of what will be loaded into the VSE/POWER reader queue by REXXLOAD.

REXXCONS.Z
Starts the REXX console application REXXCO.PROC. This is the source of what will be loaded into the VSE/POWER reader queue by REXXLOAD.

-- Demo REXX Programs --

REXXSPCE.PROC
REXX program that executes the ALLOC and FREE work requests.

REXXEVNT.PROC
REXX program that contains *internal* REXX Action programs.

REXXVSAM.PROC
REXX program that is the batch application accessing VSAM files.

-- Demo Library Members --

REXXCNTL.Z
Job Resource Control File that contains VSAM file information

Please be aware that this member has a reference to volume SYSWK1. Also, the master catalog is assumed to reside on SYSWK1. If you use different volumes you have to adjust the reference.

REXXTABL.Z
Message Action Table that contains strings of message text for which the REXX console is sensitive in order to schedule the workfile request.

SKRXJCL.Z
JCL PROC skeleton that supplies JCL statements for REXXVSMx. Will be used by REXXSPCE.PROC.

SKRXVSAM.Z
Skeleton to create REXXSPCE job which is used by the REXXCO application framework.

REXXCPUM

This function is an example of how you can make optimum use of the measurement facilities described in "REXX/VSE CPU Monitor" on page 247.

The function has three main parts

1. Verifying that CICS is active
2. Setting a system activity limit
3. Starting CICS transaction IEXM.

It also allows to terminate the CPU monitor function.

If the REXXCO application is active in another partition, it can monitor the ARX0998I messages generated by transaction IEXM in combination with user exit ARXITCPU.

Scenario

Suppose the operator has to flush jobs that consume too much CPU time. Using the REXX CPU Monitor, the operator would notice this through message

```
ARX0998I PID Y1 JOB TEST EXCEEDS THE LIMITS: CPUTIME=10.15
```

This task can be automated. REXX procedure REXXCPUM demonstrates this.

Invocation

The REXXCPUM function is called according to the following format:

CALL REXXCPUM CICS_prompt,CPU_Time_Limit,Elapsed_Time_Limit,
I/O_Count_Limit,I/O_Rate_Limit,CPU_Rate_Limit,Interval,runMonitor

where

CICS_prompt
Identifies the CICS partition, for example 'F2-0002' (which is also the default).

CPU_Time_Limit
Limit of accumulated CPU time, in 1/100 seconds, that should not be exceeded. The default is 0.

A value of 0 causes REXXCPUM not to set the CPU time limit, that is, the current limit remains unchanged.

This parameter can have a special value: STOP. This tells REXXCPUM to terminate CPU monitoring; message ARX0998I will no longer appear. You code the REXXCPUM call as follows:

CALL REXXCPUM ,'STOP'

Elapsed_Time_Limit
Limit of elapsed time a job is running, in seconds, that should not be exceeded. The default is 0.

A value of 0 causes REXXCPUM not to set the elapsed time limit, that is, the current limit remains unchanged.

I/O_Count_Limit
Limit of job I/Os, as an absolute number, that should not be exceeded. The default is 0.

A value of 0 causes REXXCPUM not to set the I/O count limit, that is, the current limit remains unchanged.

I/O_Rate_Limit
Limit of job I/Os, in numbers per second, that should not be exceeded within a measurement interval. The default is 0.

A value of 0 causes REXXCPUM not to set the I/O rate limit, that is, the current limit remains unchanged.

CPU_Rate_Limit
Limit of job CPU time, in percent of total CPU time, that should not be exceeded within a measurement interval. The default is 0.

A value of 0 causes REXXCPUM not to set the CPU rate limit, that is, the current limit remains unchanged.

Interval
The time between two measurement activities in the format hhmmss. For example '000015' is 15 seconds '001230' is 12 minutes and 30 seconds.

The smallest value is 10 seconds, the default is 15 seconds.

runMonitor
The duration of how long the CPU monitor is to run, starting from the current time. The format is 'hh:mm:ss' The program uses this value to compute the STOptime which it passes to CICS transaction IEXM. The default is 12:00:00, i.e. 12 hours.

The REXXCPUM function is intended to be called as subroutine or as function from another REXX program, for example

```
CALL REXXCPUM 'F2,1000'    or   x = (REXXCPUM 'F2')
```

If you want to invoke REXXCPUM via

```
// EXEC REXX=REXXCPUM...    or   EXEC REXXCPUM...
```

within a REXX program (for example REXXTRY), then you must remove the comment around the PARSE ARG instruction and put the following ARG instruction inside a comment. This is also described in the source code.

Also note that you may want to modify the first instruction

```
partIds = 'F1F2F3'        /*<==== Modify here to your needs */
```

which sets some default values for partitions to be excluded from monitoring.

Error Codes

REXXCPUM issues the following error codes:

40
CICS not active

48
Activate console failed

52
ADDRESS CONSOLE cons_cmd failed

56
Invalid runMonitor argument

60
Invalid cpu_time_limit argument

64
Invalid elapsed_time_limit argument

68
Invalid io_count_limit argument

72
Invalid io_rate_limit argument

76
Invalid cpu_rate_limit argument

80
Invalid interval argument

REXXDOM

This demo shows at a real console how under *REXX Console Automation* the presentation characteristics (highlighting, in this case) of messages can be changed. REXXDOM is an example for scanning the hardcopy file from a REXX program.

Running the Demo

It takes three steps to run the demo:

1. Generate a phase with name REXXWTO. This program writes highlighted messages. You generate the phase by submitting the following JCL statement:

   ```
   // EXEC REXX=REXXDOM,SIZE=(ASMA90,50K),PARM='LINK'
   ```

2. Create 10 highlighted messages by invoking the above program, either
   - from a PAUSE job:

     ```
     // EXEC REXXWTO
     ```

     or
   - in a REXX procedure:

     ```
     ADDRESS LINK REXXWTO
     ```

3. Invoke REXXDOM again, this time to remove one or more highlighted messages from the HOLD state (that is: dehighlight it). Submit the instruction

   ```
   EXEC REXXDOM n
   ```

where n is the number of messages (between 1 and 10) to be dehighlighted. You place this statement either in a REXX procedure, or you enter it from a REXXTRY prompt (REXXTRY is presented in the following section).

When you run the demo again, you can start with step 2 and leave out the first step.

Background Information

REXXDOM uses the command

REDISPLAY ...,HOLD

to retrieve from the hardcopy file messages that are in HOLD state. It counts these messages (excluding messages about outstanding replies which will not be deleted). It then uses the MDBGMID variable(s) to initiate the DELMSG function as often as requested by the user.

Other Examples (Not Related to Console Functions)

REXXTRY

Before you start using any of the demo units, you have to load them into a predefined place using the REXXLOAD program. This is described in section “REXXLOAD” on page 253.

This job calls REXX program REXXTRY which prompts you to interactively try REXX statements, possibly including REXX commands that you developed yourself.

If you run REXXTRY with no parameter, or with a question mark as a parameter, it will briefly describe itself. You may also enter a REXX statement directly on the command line for immediate execution, for example

```
4 call sysdef 'connect occf'
```

or, to utilize REXXTRY as an interactive execution environment,

```
4 exec rexxasm
```

or one of your own REXX programs.

REXXJMGR

This program implements a simple job manager. VSE/POWER jobs with name MYJOBnnn, nnn=1,2,3,.., are submitted. When starting REXXJMGR, you specify in PARM how many jobs are to be submitted (the default is 2).

Asynchronous execution will be controlled. This subject is discussed at some detail in section “Submitting and Controlling Power Jobs” on page 198.

REXXJMGR writes a report, an example is shown below:

```
0 exec rexxjmgr
Y3 0001 1Q47I   Y3 MYJOB2 08172 FROM BOEVSE16(REXXJMGR) , TIME=12:10:56
Y3 0047 // JOB MYJOB2
        DATE 08/30/95,CLOCK 12/10/56
Y2 0001 1Q47I   Y2 MYJOB1 08171 FROM BOEVSE16(REXXJMGR) , TIME=12:10:56
Y2 0046 // JOB MYJOB1
        DATE 08/30/95,CLOCK 12/10/56
Y3 0047 EOJ MYJOB2    MAX.RETURN CODE=0004
        DATE 08/30/95,CLOCK 12/10/58,DURATION   00/00/01
Y2 0046 EOJ MYJOB1    MAX.RETURN CODE=0000
        DATE 08/30/95,CLOCK 12/10/58,DURATION   00/00/01
BG 0000 ****** JOB REPORT **************************
BG 0000 ****** MYJOB2 failed MAXRC=0004 TIME=12:10:58
BG 0000 ****** MYJOB1 run successfully  TIME=12:10:58
BG 0000 ****** END OF REPORT ************************
BG 0000   rc = 0 ..................................... REXXTRY on VSE
BG-0000
```

REXXWAIT

This program submits a job and awaits its execution.

The job runs a utility program in another partition. It issues a List Directory command. Its output is retrieved and scanned for a user-supplied string (the default is ARXINIT).

An example of console output is shown below.

```
45 exec rexxwait
Y2 0001 1Q47I   Y2 MYJOB 04584 FROM BOEVSE16(REXX) , TIME=12:05:18
Y2 0046 // JOB MYJOB
        DATE 07/13/95,CLOCK 12/05/18
Y2 0046 EOJ MYJOB     MAX.RETURN CODE=0000
        DATE 07/13/95,CLOCK 12/05/19,DURATION   00/00/00
Y1 0045 *-----------------------------------------------*
Y1 0045  Job output was analysed. Match found at line 40
Y1 0045 *-----------------------------------------------*
Y1 0045 ARXINIT  PHASE    95-05-09 95-07-11 303032 B     307  NO YES  31 ANY
Y1 0045   rc = 0 ..................................... REXXTRY on VSE
```

REXXASM

This program works with VSE JCL and VSE utilities. A small assembler program will be compiled, linked and executed. Its name can be supplied in PARM (the default is DEMOSVA).

The size of the program will be shown via a LIBR LD SDL excerpt.

You may invoke the program from a REXXTRY prompt. An example of console output is shown below.

```
45 exec rexxasm
Y1 0045 *-------------------------------------------------------------------*
Y1 0045 (1) Module DEMOSVA assembled
Y1 0045 (2) Module DEMOSVA linked
Y1 0045 (3) Module DEMOSVA executed: RC= 0
Y1 0045 (4) Module DEMOSVA found in library PRD1.BASE:
Y1 0045 DEMOSVA  PHASE    95-07-03 95-07-13      6 B       1 YES YES  31  24
Y1 0045 *-------------------------------------------------------------------*
Y1 0045   rc = 0 ..................................... REXXTRY on VSE
Y1-0045
```

REXXSSDL

This program communicates with the VSE Librarian to either

- load a phase into the SVA (using the JCL function SET SDL) and to show the phase's address in the SVA, or to
- just show the address. In this case, you invoke the program via

```
exec rexxssdl demosva (noload
```

The phase name can be supplied as a parameter (the default is DEMOSVA).

An example of a REXXSSDL output is shown below.

```
0 exec rexxssdl demosva
BG 0000 ---------------------------------------------------------------------
BG 0000  M E M B E R    ORIGIN SVA/MOVE  LOADED  PHASE  ADDRESS  ENTRY POINT
BG 0000 NAME     TYPE   SYSLIB   MODE   INTO SVA SIZE   IN SVA   IN SVA
BG 0000 ---------------------------------------------------------------------
BG 0000 DEMOSVA  PHASE     NO    YES      YES        6  001E68F8 001E68F8
BG 0000   rc = 0 ..................................... REXXTRY on VSE
BG-0000
```

Miscellaneous Examples of Using the REXX Console

Retrieve Messages using Filter and Timestamp

```
ADDRESS CONSOLE
'ACTIVATE NAME REXX PROFILE REXNORC'  /* Activate console session   */

'RED 10L,'0S03I'                  /* VSE console REDisplay command  */
                                  /* to get message(s) 0S03I        */
'RED E'                           /* End redisplay                  */

rc = GETMSG(consmsg.,'RESP',,,30) /*Retrieve response of redisplay  */
 .....
 .....
'DEACTIVATE REXX'                /* Deactivate console session      */
EXIT
```

Scan the Hardcopy File

The VSE system command REDISPLAY allows to retrieve messages from a specific partition.

```
ADDRESS CONSOLE
'ACTIVATE NAME REXX PROFILE REXNORC'  /* Activate console session   */

'RED 34L,F5,E'                   /* VSE console REDisplay command    */
                                 /* to get message from partition F5 */

rc = GETMSG(consmsg.,'RESP',,,30) /*Retrieve response of redisplay  */
 .....
 .....
'DEACTIVATE REXX'                /* Deactivate console session      */
EXIT
```

Just like the VSE utility PRINTLOG, the REDISPLAY command allows to view records from the hardcopy file in a selective manner. You can specify a filter so you will see only messages from a particular partition, or messages with a particular message ID.

Scan Job Messages for One Partition

REXX program REXXSCAN, by utilizing the VSE command REDISPLAY and the GETMSG function of *REXX Console Automation*, creates a report covering all jobs that have been running in a given partition since a given date. The partition ID and the date are passed as PARM information when invoking the program:

```
// EXEC REXX=REXXSCAN,PARM='partid date'
```

```
/*********************************************************************/
/***** REXXSCAN:                                                 ******/
/***** Scan Job Messages REXX program for REXX Console Automation    */
/***** Demo Purposes.                                                */
/*****                                                               */
/***** Test VSE console REDISPLAY command together with the REXX     */
/***** Console GETMSG function. Create a job report of all jobs      */
/***** that have been running in a given partition since a given     */
/***** date. Job report is contained in the POWER LST queue and      */
/***** displays POWER jobname, POWER jobnumber, POWER starttime,     */
/***** VSE jobname, startdate, starttime, endtime, enddate,          */
/***** duration, and the maximum return code of a VSE job.           */
/*****                                                               */
/***** INVOCATION: // EXEC REXX=REXXSCAN,PARM='partid date'          */
/*****             Scans all messages written for partition or class */
/*****             'partid' starting with date                       */
/*****   partid    BG,F1...F9,FA,FB,partition ids of dynamic classes */
/*****   date      mm/dd/yy                                          */
/*****                                                               */
/*****   Example:  // EXEC REXX=REXXSCAN,PARM='BG 08/31/95'          */
/*****                                                               */
/***** RETURN CODES of the REXXSCAN program:                         */
/*****                                                               */
/*****   0         All messages for partition scanned                */
/*****   999       Invalid parameters                                */
/*****   1000+x    Problems with GETMSG                              */
/*****                                                               */
/*********************************************************************/

PARSE ARG partid date .   /* Pass as argument partition id and date  */
/*********************************************************************/
/* check partid                                                      */
/*********************************************************************/
 .....
 .....

/*********************************************************************/
/* check date                                                        */
/*********************************************************************/
 .....
 .....

ADDRESS CONSOLE
CALL SYSVAR(syspid)      /* Get the Partition the PROC is running in */
retcode = 0              /* initialize return code                   */
                         /* initialize end of redisplay condition    */
end_line = '----------------------- E N D  O F  F I L E ---'

/*********************************************************************/
/* Put heading line to output stream                                 */
/*********************************************************************/
SAY 'PWR_jobn jobnr PWRstart VSE_jobn startdate starttim enddate' ||,
    '    endtime  duration   rc'
```

Figure 15. Job Message Scanner REXXSCAN - Part 1 of 3

```
SAY '============================================================' ||,
    '======================'

/*********************************************************************/
/* Activate console session                                          */
/*********************************************************************/
ACTIVATE 'name REXX'SYSPID 'profile REXNORC'

/*********************************************************************/
/* Start REDISPLAY for partition partid                              */
/*********************************************************************/
'RED F,' || date || ',' || partid   /* start redisplay               */
IF RC > 0 THEN
DO
  retcode = RC
  SIGNAL end_label
END

/*********************************************************************/
/* Analyze messages from REDISPLAY                                   */
/*********************************************************************/
DO FOREVER                         /* Forever - until end of messages */
 rc=GETMSG(t.,'msg',,,10)          /* Retrieve response of redisplay  */
 IF rc>0 THEN                      /* show problems with MSG retrieval*/
 DO
   retcode = 1000+rc
   LEAVE
 END
 IF (SUBSTR(t.1,1,LENGTH(end_line)) = end_line) THEN
 DO                                /* no more messages              */
   LEAVE
 END

 message = SUBSTR(t.1,1,48)        /* Find the msg ID               */
 first = WORD(message,3)           /* first word of message         */
 second = WORD(message,4)          /* second word of message        */

 SELECT
 WHEN (first='1Q47I' |,            /* start of POWER JOB found      */
       first='//' & second='JOB' |, /* start of VSE JOB found       */
       first='EOJ') THEN           /* end of VSE JOB found          */
   DO                              /* concatenate message parts     */
     total_msg = ''
     DO I=1 TO t.0
       total_msg = total_msg || t.I
     END
   END
 OTHERWISE                         /* do nothing                    */
 END /* select */
```

Figure 16. Job Message Scanner REXXSCAN - Part 2 of 3

```
 SELECT                             /* do message analysis           */
 WHEN(first='1Q47I') THEN           /* start of POWER job found      */
   DO
     /* save POWER jobname job_number and starttime */
     PARSE VAR total_msg,
       . . . . power_jobname job_number nil1 ', TIME=' power_starttime nil2
   END
 WHEN(first='//' & second='JOB') THEN /* start of VSE job found      */
   DO
     /* save VSE jobname startdate and starttime */
     PARSE VAR total_msg,
       . . . . VSE_jobname nil ' DATE ' start_date ', CLOCK ' start_time
   END
 WHEN(first='EOJ') THEN             /* end of VSE job found          */
   DO
     /* save VSE jobname returncode enddate endtime and duration */
     PARSE VAR total_msg,
       . . . VSE_jobname2 nil1 ' MAX.RETURN CODE=' job_rc nil2,
       ' DATE ' end_date ',CLOCK ' end_time ',DURATION ' job_time nil
     /* put info into the current output stream, i.e. LST output */
     SAY SUBSTR(power_jobname,1,8) job_number,
         RIGHT(power_starttime,8),
         SUBSTR(VSE_jobname,1,8),
         start_date SUBSTR(start_time,1,8) end_date end_time,
         RIGHT(STRIP(job_time),9) job_rc
   END
 OTHERWISE                          /* accepted messages             */
 END /* select */

'RED'                               /* continue redisplay            */
 IF RC > 0 THEN
 DO
   retcode = RC
   SIGNAL stop_label
 END
END /* do forever */

stop_label:
/*********************************************************************/
/* Stop REDISPLAY for partition partid                               */
/*********************************************************************/
'RED END'                           /* stop redisplay mode           */
IF RC > 0 THEN
DO
  retcode = RC
  SIGNAL end_label
END

end_label:
/*********************************************************************/
/* Deactivate console session                                        */
/*********************************************************************/
DEACTIVATE 'REXX'SYSPID

EXIT retcode
```

Figure 17. Job Message Scanner REXXSCAN - Part 3 of 3

Return and Reason Codes

This section lists return and reason codes, together with explanations, as issued by the

MCSOPER macro
MCSOPMSG macro
MCSCRE macro
Command processors.

Note: Return and reason codes are given in decimal format.

MCSOPER Macro

00 00
Successful completion.

04 00
Console with specified name is already active (ACTIVATE) or not active (DEACTIVATE).

16 00
Invalid input: The address of the parameter list or of an input parameter is invalid.

16 02
Invalid input: The specified console was not activated by this task (DEACTIVATE).

16 04
Invalid input: The requested function is invalid (not ACTIVATE nor DEACTIVATE).

16 08
Invalid input: The specified name contains invalid characters, or is none of the predefined values nor a valid VSE/ESA userid (ACTIVATE).

16 16
Invalid input: The specified MSGDLVRY option is invalid (ACTIVATE).

16 24
Invalid input: The specified authority level (OPERPARM area) is invalid.

16 32
Invalid input: The specified message level (OPERPARM area) is invalid.

16 44
Invalid input: The macro acronym or version indicator in the parameter list is invalid.

20 00
Service routine failure.

24 00
The caller is not in supervisor state or not in primary ASC mode or not in 31-bit addressing mode.

MCSOPMSG Macro

00 00
Successful completion. For REQUEST=GETMSG, reason code 00 also indicates that no more messages nor DOMs are currently queued for this console.

00 01
REQUEST=GETMSG completed successfully, and at least one more message is queued for this console.

00 02
REQUEST=GETMSG completed successfully, and at least one DOM is queued for this console.

00 03
REQUEST=GETMSG completed successfully, and at least one message and one DOM are queued for this console.

04 00
Console was not suspended (only applicable for REQUEST=RESUME).

08 00
No message available for the specified REQUEST=GETMSG search criteria (if any), and no more messages nor DOMs are currently queued for this console.

08 01
No message available for the specified REQUEST=GETMSG search criteria, but there are is at least one other message queued for this console.

08 02
No message available for the specified REQUEST=GETMSG search criteria, but there are is at least one DOM queued for this console.

08 03
No message available for the specified REQUEST=GETMSG search criteria, but there are is at least one message and one DOM queued for this console.

12 00
Console is suspended (applicable only for REQUEST=GETMSG). REQUEST=RESUME must be issued before messages can be retrieved again for this console.

16 00
Invalid input: The requested function is invalid (not GETMSG or RESUME).

16 01
Invalid console ID: The console is not active.

16 02
Invalid console ID: The console was not activated by this task.

20 00
The address of the parameter list or of an input parameter is invalid.

20 01
The parameter list contains an incorrect macro acronym or version indicator.

20 04
The console was activated with MSGDLVRY=NONE, or with MSGDLVRY=FIFO but CMDRESP=YES was specified.

20 05
The caller is not in supervisor state or not in primary ASC mode or not in 31-bit addressing mode.

24 00
Service routine failure.

MGCRE Macro

00 00
Processing completed successfully, input is accepted.

00 01
Input is accepted, but was recognized as sensitive, like a Job Control // ID statement possibly containing a password. The input text is logged with an overlay '(PARAMETERS SUPPRESSED)' and the modified text is returned in the CSA, allowing consoles to echo it instead of the original input text.

04 00
Console with specified name is already active.

08 01
Command not accepted because a previous command from the same console and for the same command processor is not yet completed.

08 02
Invalid reply ID. Either no message is pending for the specified reply ID or the console is not authorized to reply to the pending message.

08 03
The console is not authorized for the specified command.

08 04
The Attention command processor is not active.

08 05
The Redisplay command processor is not active.

08 06
Input from system console is inhibited due to REMOTE operating mode (there is a relation to the OPERATOR command).

08 07
Redisplay mode is already active for another user. This condition is only possible for consoles that operate on behalf of multiple users by means of the UTOKEN parameter.

08 08
The input was rejected by an exit routine.

08 09
REDISPLAY C or E is rejected because redisplay mode is not active.

08 10
REDISPLAY command rejected due to shortage of 24-bit system GETVIS storage.

08 11
A command was issued at a user console while this console was still in redisplay mode, explanation mode, or help mode.

08 16
Command not accepted because the specified console is suspended.

08 17
The specified command (e.g. REDISPLAY or EXPLAIN) is not supported for an inactive console (only possible when CONSNAME was specified).

08 18
No dummy console is available to process input for an inactive console (only possible when CONSNAME was specified).

12 00
The input text is all blanks.

12 01
The input length is 0 or larger than 126 (not EXPLAIN), or different from 0 and 12 for EXPLAIN requests.

12 02
The input starts with a numeric character, but there is no leading token of 1 to 4 numeric characters that can be interpreted as a reply ID.

16 01
Invalid console ID: The console is not active.

16 02
Invalid console ID: The console was not activated by this task.

16 08
Invalid console name: The name is shorter than 4 characters or contains invalid characters.

20 00
Service routine failure.

Command Processor Return and Reason Codes

The table below shows the return codes (RC) and reason codes (RS) of the three command processors

- Console Router
- Attention Routine
- Hardcopy File (HCF).

Table 3. Return and Reason Codes from Command Processors

RC	RS	Console Router	Attention Routine	HCF Processor
0	0	okay	okay	okay
0	1	not used	not used	okay, response is caused by RED E,xL

Table 3. Return and Reason Codes from Command Processors (continued)

RC	RS	Console Router	Attention Routine	HCF Processor
4	0	last message of command response	not used	last message of command response
4	1	not used	not used	last message of command response, caused by RED E,xL
4	2	not used	not used	last message of command response, top of HCF
4	3	not used	not used	last message of command response, bottom of HCF
8	0	not used	end of command response, command was processed by attention routine.	not used
8	1	not used	end of command response, command was passed to a subsystem.	not used
8	>8	MDB contains an information or error message (English version); could indicate end of command response.	not used	MDB contains an information or error message (English version); could indicate end of command response.

CORCMD Command for Problem Solving

An unsupported debug command is available which gives information about the internal status of the console router. In case of a problem, IBM service personnel might ask you to enter this command. In some situations you might even be able to analyze the command output yourself to solve a specific problem.

Just enter **CORCMD** plus one of the following keywords. For example

```
CORCMD STATUS=CONS
```

Please note that command parsing is not very fancy. Therefore, type a keyword exactly as shown. In particular, no blanks are allowed before or after the **=** sign.

STATUS=CONS

returns status information about the consoles known to the console router. *Number of consoles* shows the number consoles that are active, suspended, accept undeliverable messages, or can be alerted. Then a list of all consoles follows, showing the console name, the console ID, and the status of a console. A console ID with the high-order bit on, as shown with console name HAUS, indicates a user console. Otherwise it is a master console.

Note: This command can only display 10 lines of data. Therefore, only the first 30 consoles are displayed.

```
corcmd status=cons
Number of consoles:  Act=00000003 Sus=00000000 Ud=00000001 Al=00000001
SYS      00000002 A---     VMC      00000003 A---     IC 00000004 A--U
REXX     00000023 A---     HAUS     80000125 A---
End of STATUS=CONS
```

Explanation:

```
Numer of consoles:  ACTive, SUSpended, accepts UD msgs, accepts ALerts
Status:             Active, Suspended, Netview(automa.), Undel. msgs
```

CONS=console_name
returns status information about a specific console.

```
corcmd cons=sys
Name=SYS       ID=001BD3A4  Date=1995072  Time=09:14:24.63  Stat=Nrm
RtCd=FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  MLvl=FE00 Ud=N Al=Y Aut=N DOM=Def
EC=0000  RCS=08000000  CmdPnd=-----  MsgDlv=Srch  CSAFl=---
QFrst=00000000  QSrch=00000000  QDOM=00000000
MsgCt=00000000  DOMCt=00000000  SusCt=00000000
End of STATUS=CONS
```

Explanation:

```
Date & Time of last activate, suspend or resume
Stat     ACTive, SUSpended, RESumed, INItializing, NoRMal
RtCd     enabled routing codes
MLvl     enabled message level
Ud       undeliverable messages accepted: Y or N
Al       alert ECB specified: Y or N
Aut      automatable messages accepted: Y or N
DOM      DOM option: ALL, DEFault, NONe
EC       unique error code for specific return codes
RCS      last return and reason code passed via MCSOPMSG
CmdPnd   command pending:
         A: AR cmd, H: HCF cmd, C: Console router cmd
         M: HCF cancel/end msg, X: explanation request
MsgDlv   message delivery option: SeaRCH, FIFO, NONe
CSAFl    Console Status Area flags
         A: alerted, U: suspended, X: explain response
MsgCt    number of undelivered messages
DOMCt    number of undelivered DOms
SusCt    number of suspends since last activate
QFrst, QSrch, QDOM   debug information
```

FORCE
this command checks whether the oldest queue item of the console router main queue is blocked by a console and, because of this, the reusage of such a queue item is prevented. When this is the case, then such a console is suspended, no matter what console it is. The system responds with:

```
Number of consoles suspended due to CORCMD FORCE : 01
```

STATUS=QUEUE
returns status information about the console router queues and queue space management.

```
corcmd status=queue
GETVIS for  RI:  Lim=0010 Cur=0000  ML: Lim=0028 Cur=0000
Non-returnable:  RI=0000007F  ML=000001F3  QMGEmpty: TIK=0021 Code=0001
Returnable  RI:  Lim=0064  Hi=0000  Cur=0000
Returnable  ML:  Lim=0064  Hi=0000  Cur=0000
Alert       :  Pct=0032  RI-Base=00000071  RI-Pct=00000038
CRQ: Cur=001B Hi=001B    MRQ: Cur=0001 Hi=0001    DYQ: Cur=0000 Hi=0000
LRQ: Cur=0019 Hi=0019    DHQ: Cur=0000 Hi=0000
ARQ: Cur=0000 Hi=0001    HCQ: Cur=0000 Hi=0000
FRQ: Cur=0037 Hi=0044    YRQ: Cur=000A Hi=000A    XRQ: Cur=0022 Hi=0031
DOQ: Cur=0002 Hi=0002    XTQ: Cur=0000 Hi=0000    MOQ: Cur=000B Hi=0000
MLQ: Cur=000E Hi=0048    YMQ: Cur=000A Hi=000A    XMQ: Cur=0168 Hi=01A1
End of STATUS=QUEUE
```

Explanation:

```
RI       (routing) queue item
ML       message line
Lim      maximum number (e.g. GETVIS)
Cur      current number
Hi       highest number ever (can exceed LIM for returnable RI and ML)
QMGEmpty: TIK      task ID of last unsuccessful request to get an
                   empty queue item
          Code     Code of last unsuccessful request:
      00  no free RI or ML found
```

```
01  RI requ for APNormal: too many undeliver. msgs for task (RI
    NOTE: This is the typical case when a task wrote messages,
          but such messages were not retrieved from all consoles
02  QMEXMLSP: no GETVIS in IPL stage 2
03  QMEXMLSP: GETVIS failed
04  QMEXRISP: no GETVIS in IPL stage 2
05  QMEXRISP: max number of expansions reached
06  QMEXRISP: GETVIS failed
07  QMFREE1: CRQ is empty
08  QMFREE1: cannot be reused since not logged yet
09  QMFREE1: cannot be reused, blocked by a console
    NOTE: A console that cannot be suspended prevents reusage of
          a queue item. CORCMD FORCE drops such a console.
0A  QMFREE1: no more RIs in CRQ
0B  QMGETSRI: GETVIS for single RI failed
0C  QMGETSML: GETVIS for single ML failed
0D  PROCRIML: Did not get a ML
0E  PROCRIML: Did not get a ML
0F  PROCRIML: invalid condition
10  ML requ for APNormal: too many undeliver. msgs for task (ML
    NOTE: see note of code=01
11  QMFREE1: cannot be reused, blocked by console, RCSUSPND
    NOTE: see note of code=09
```

Note: The output of the *STATUS=QUEUE* command can only be interpreted with deep system knowledge and is therefore not very useful for the general user. To help you to determine whether a console router queue space problem exists, check the following:

GETVIS for RI (ML)
: During IPL, the console router allocates an initial amount of queue space. This space can be expanded when heavy message traffic occurs. Expansion is done via 31-bit system GETVIS requests. Today, the size of such a GETVIS request is about 4 KB. This space will never be returned. As long as *Cur* shows a value below *Lim*, the console router has no general space problem, except if 31-bit system GETVIS is generally exhausted. The number of queue items (RI) and message lines (ML) that were built from this permanent space is displayed under *Non-returnable RI (ML)*

Returnable RI (ML)
: displays the number of queue items and message lines that are returnable to system GETVIS space. Returnable queue space is requested in 31-bit system GETVIS space in emergency situations and if the permanent space is not sufficient. *Cur* shows the number of allocated queue items or message lines that will be returned as soon as possible. In very rare cases, *Cur* might even exceed *Lim* to prevent system hangups. Over the long run, *Cur* should return to zero. However, this can take some time.

Alert
: shows numbers needed for debug.

CRQ, etc.
: shows the number of queue items in the different queues managed by the console router. *CRQ* is the console router main queue. Problems might soon arise if the *Cur* number is very low, for example, below 5. These actions might give relief to the system:

  1. Reply as many outstanding replies as possible.
  2. Terminate as many jobs (tasks) as possible.
  3. Terminate as many consoles as possible.

  The numbers shown in all the other queues are for debug purpose.

QMGEmpty: TIK= Code=
: gives information about the last failing request of an application for an empty queue item to build a new queue item. This state is temporary and the information given in this field might no longer be true. However, if the information is true, the *STATUS* command shows *WAITING FOR ROUTER BUFFER SPACE*. *Code* gives a reason why the request for the task indicated by *TIK* failed.

TRACE
: returns the current trace setting (ON or OFF)

TRACE=ON
sets the console router trace to on. If no trace area exists, a trace area with the default size is allocated (8 KB).

Note: The **DEBUG** command also turns on and off the console router trace. Only if a console router trace during IPL is needed, the CORCMD TRACE=ON must be used.

TRACE=OFF
sets the console router trace to off.

TRACE=END
returns the storage of the trace area to the system.

TRACE=n
changes the size of the trace area. **n** is the value in KBytes.

Chapter 15. REXX Sockets Application Program Interface

The REXX Sockets application program interface (API) allows you to write socket applications in REXX for the TCP/IP environment. It follows the standard TCP/IP Socket API available on multiple platforms and therefore enables porting of socket programs from other platforms to REXX/VSE.

The REXX socket program external uses the LE/VSE Support to access the TCP/IP Socket interface. The program maps the socket calls from the C programming language to the REXX programming language. This allows you to use REXX to implement and test TCP/IP applications. Examples of the corresponding C socket call are included where they apply.

For general information about sockets see the TCP/IP Support.

Subtopics:

REXX Socket SSL Support

TCP/IP for VSE/ESA 1.4 provides SSL (Secure Sockets Layer) support. SSL is a security protocol and allows Internet servers and clients to authenticate each other and to encrypt the data flowing between them. Based on this SSL support, the REXX/VSE Socket function has been enhanced with VSE/ESA 2.7 to enable you to write SSL-enabled socket applications in REXX.

To use SSL functions in general, the VSE/ESA host must first be configured for SSL support. This is described in detail in the z/VSE e-business Connectors User's Guide.

The following Socket sub-functions have been enhanced for the SSL support:

- Initialize
- Accept
- Connect
- Takesocket

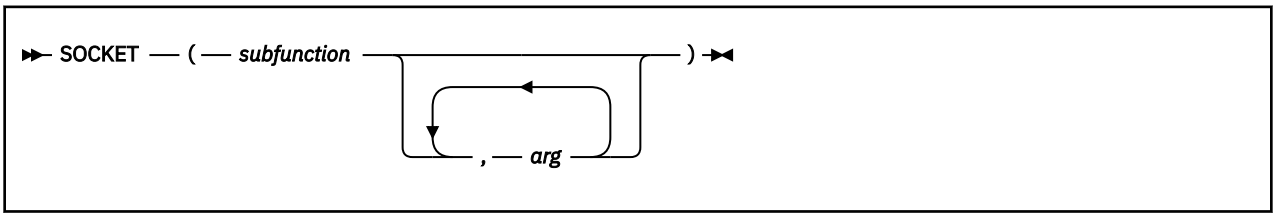
Programming Hints and Tips for Using REXX Sockets

This section contains some information that you might find useful if you plan to use REXX Sockets.

- To use the socket functions contained in this interface, a socket set must be active. The Initialize function creates a socket set and can be used to create as many socket sets as required. The *subtaskid* for a socket set identifies the socket set and should be set to a string value that resembles the application's purpose.
- The *socketname* parameter on a socket function contains values for *domain*, *portid*, and *ipaddress*. If *socketname* is specified as an input parameter on a socket call, you can specify *ipaddress* as a name to be resolved by a name server. For example, you can enter 'CUNYVM' or 'CUNYVM.CUNY.EDU'.

- A socket can be in blocking or nonblocking mode. In blocking mode, functions such as Send and Recv block the caller until the operation completes successfully or an error occurs. In nonblocking mode, the caller is not blocked, but the operation ends immediately with the return code 1102 (EWOULDBLOCK) or 1103 (EINPROGRESS. You can use the Fcntl or Ioctl function to switch between blocking and nonblocking mode.
- When a socket is in nonblocking mode, you can use the Select function to wait for socket events, such as data arriving at a socket for a Read or Recv function. If the socket is not ready to send data because buffer space for the transmitted message is not available at the receiving socket, your REXX program can wait until the socket is ready for sending data.

SOCKET External Function



The first parameter in the SOCKET function, *subfunction*, identifies a REXX socket function. The REXX socket function may have additional arguments. REXX socket functions are provided for:

- Processing socket sets
- Creating, connecting, changing, and closing sockets
- Exchanging data
- Resolving names and other identifiers for sockets
- Managing configurations, options, and modes for sockets

See "Tasks You Can Perform Using REXX Sockets" on page 276 and "REXX Socket Functions" on page 279.

SOCKET returns a character string that contains several values separated by blanks, so the string can be parsed using REXX. The first value in the string is the return code. If the return code is zero, the values following the return code are returned by the socket function (*subfunction*). If the return code is not zero, the second value is the name of an error, and the rest of the string is the corresponding error message.

For example:

Call
Return Values

```
Socket('GetHostId')
   '0 9.4.3.2'
Socket('Recv',socket)
   '1102 EWOULDBLOCK Operation would block'
```

For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

During initialization of the REXX Sockets module or when doing certain REXX socket functions, system messages may also be issued. See "REXX Sockets System Messages" on page 307.

The description of each REXX socket function in this topic provides at least one example of the call and return value string, and also includes an example of the corresponding C socket call, where applicable.

Tasks You Can Perform Using REXX Sockets

You can use REXX Sockets to perform the following tasks:

- **Processing socket sets**

A socket set is a number of preallocated sockets available to a single application. You can define multiple socket sets for one session, but only one socket set can be active.

The functions included in this task group are shown in Table 4 on page 277.

Table 4. REXX socket functions for processing socket sets

Function	Purpose
Initialize	Defines a socket set
Terminate	Closes all the sockets in a socket set and releases the socket set
SocketSet	Gets the name of the active socket set
SocketSetList	Gets the names of all the available socket sets
SocketSetStatus	Gets the status of a socket set

- **Creating, connecting, changing, and closing sockets**

  A socket is an endpoint for communication that can be named and addressed in a network. A socket is represented by a socket identifier (*socketid*). A socket ID used in a Socket call must be in the active socket set.

  The functions included in this task group are shown in Table 5 on page 277.

Table 5. REXX socket functions for creating, connecting, changing, and closing sockets

Function	Purpose
Socket	Creates a socket in the active socket set
Bind	Assigns a unique local name (network address) to a socket
Listen	Converts an active stream socket to a passive socket
Connect	Establishes a connection between two stream sockets
Accept	Accepts a connection from a client to a server
Shutdown	Shuts down a duplex connection
Close	Shuts down a socket
GiveSocket	Transfers a socket to another application
TakeSocket	Acquires a socket from another application

- **Exchanging data**

  You can send and receive data on connected stream sockets and on datagram sockets.

  The functions included in this task group are shown in Table 6 on page 277.

Table 6. REXX socket functions for exchanging data

Function	Purpose
Read	Reads data on a connected socket
Write	Writes data on a connected socket
Recv	Receives data on a connected socket
Send	Sends data on a connected socket
RecvFrom	Receives data on a socket and gets the sender's address

Table 6. REXX socket functions for exchanging data (continued)	
Function	**Purpose**
SendTo	Sends data on a socket, and optionally specifies a destination address

- **Resolving names and other identifiers**

  You can get information such as name, address, client identification, and host name. You can also resolve an Internet Protocol address (IP address) to a symbolic name or a symbolic name to an IP address.

  The functions included in this task group are shown in Table 7 on page 278.

Table 7. REXX socket functions for resolving names and other identifiers	
Function	**Purpose**
GetClientId	Gets the calling program's TCP/IP identifier
GetHostId	Gets the IP address for the host processor
GetHostName	Gets the name of the host processor
GetPeerName	Gets the name of the peer connected to a socket
GetSockName	Gets the local name to which a socket was bound
GetHostByAddr	Gets the host name for an IP address
GetHostByName	Gets the IP address for a host name
Resolve	Resolves the host name through a name server

- **Managing configurations, options, and modes**

  You can obtain the version number of the REXX Sockets function package, get socket options, set socket options, and set the mode of operation. You can also determine the network configuration.

  The functions included in this task group are shown in Table 8 on page 278.

Table 8. REXX socket functions for managing configurations, options, and modes	
Function	**Purpose**
Version	Gets the version and date of the REXX Sockets function package
Select	Monitors activity on selected sockets
GetSockOpt	Gets the status of options for a socket
SetSockOpt	Sets options for a socket
Fcntl	Sets or queries the mode of a socket
Ioctl	Controls the operating characteristics of a socket

- **Translating data and doing tracing**

  You can translate data from one type of notation to another. You can also enable or disable tracing facilities.

  The functions included in this task group are shown in Table 9 on page 279.

Table 9. REXX socket functions for translating data and doing tracing	
Function	**Purpose**
Translate	Translates data from one type of notation to another

REXX Socket Functions

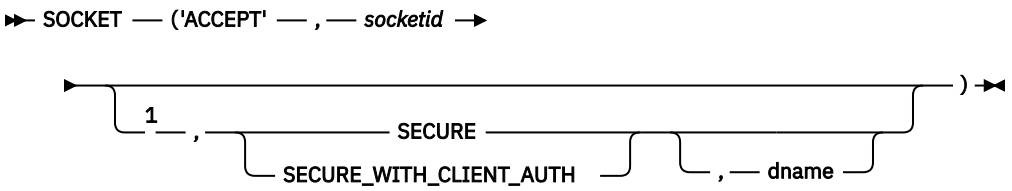
This section describes the REXX socket functions, which are listed alphabetically.

Subtopics:

- Accept
- Bind
- Close
- Connect
- Fcntl
- GetClientId
- GetHostByAddr
- GetHostByName
- GetHostId
- GetHostName
- GetPeerName
- GetSockName
- GetSockOpt
- GiveSocket
- Initialize
- Ioctl
- Listen
- Read
- Recv
- RecvFrom
- Resolve
- Select
- Send
- SendTo
- SetSockOpt
- ShutDown
- Socket
- SocketSet
- SocketSetList
- SocketSetStatus
- TakeSocket
- Terminate
- Translate
- Version
- Write

Accept

Format



Notes:

[1] The third and fourth operand are only allowed if the socketset has been initialized for SSL support.

Use the Accept function on a server to accept a connection request from a client. It is used only with stream sockets.

The Accept function accepts the first connection on the listening (passive) socket's queue of pending connections. Accept creates a new socket with the same properties as the listening socket and returns the new socket ID to the caller. If the queue has no pending connection requests, Accept blocks the caller unless the listening socket is in nonblocking mode. If no connection requests are queued and the listening socket is in nonblocking mode, Accept ends with return code 1102 (EWOULDBLOCK). The new socket is in active mode and cannot be used to accept new connections. The original socket remains in passive mode and is available to accept more connection requests.

Operands

socketid
is the identifier of the passive socket on which connections are to be accepted. This is a socket that was previously placed into passive mode (listening mode) by calling the Listen function.

SECURE | SECURE_WITH_CLIENT_AUTH
specifies to perform the SSL handshake. With SECURE the SSL server handshake is performed, with SECURE_WITH_CLIENT_AUTH the SSL handshake is performed as a server that requires client authentication.

dname
specifies a character string that is the member name of the desired entry (certificate) in the keyring library. If nothing is specified, the first keyring entry is used.

```
fc = SOCKET('ACCEPT',newsocketid,'SECURE_WITH_CLIENT_AUTH','SAMPLE')
```

If option SECURE_WITH_CLIENT_AUTH has been chosen for ACCEPT or TAKESOCKET, information from the client's certificate is stored into REXX variables. The following REXX variables are set:

Table 10. REXX Variables

Name	Description
GSK_CERT_BODY	Base64 certificate body
GSK_SESSIONID	Session ID for the connection
GSK_NEW_SESSION_ID	Flag to indicate if new session ID. If it is a new session ID, GSK_NEW_SESSION_ID is set to 1, otherwise it is set to 0.
GSK_SERIAL_NUM	Certificate Serial number
GSK_COMMON_NAME	Common name of client
GSK_LOCALITY	Locality

Table 10. REXX Variables (continued)	
Name	**Description**
GSK_STATE_OR_PROVINCE	State or Province
GSK_COUNTRY	Country
GSK_ORG	Organization
GSK_ORG_UNIT	Organizational unit
GSK_ISSUER_COMMON_NAME	Issuer's common name
GSK_ISSUER_LOCALITY	Issuer's locality
GSK_ISSUER_STATE_OR_PROVINCE	Issuer's state or province
GSK_ISSUER_COUNTRY	Issuer's country
GSK_ISSUER_ORG	Issuer's organization
GSK_ISSUER_ORG_UNIT	Issuer's organizational unit

Responses If successful, this function returns a string containing return code 0, the new socket ID, and the socket name. (The socket name is the socket's network address, which consists of the domain, port ID, and the IP address.) If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Call	**Return Values**
`Socket('Accept',5)`	`'0 6 AF_INET 5678 9.4.3.2'`
`Socket('Accept',5 ,'SECURE_WITH_CLIENT_AUTH','SAMPLE')`	`'0 6 AF_INET 5678 9.4.3.2'`

Examples

Call
Return Values

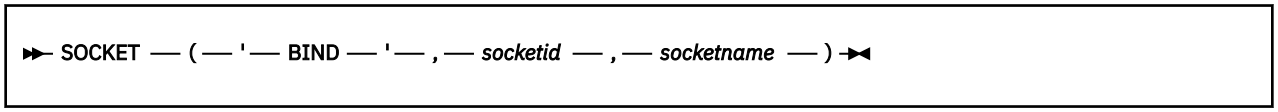
`Socket('Accept',5)`
`'0 6 AF_INET 5678 9.4.3.2'`

The C socket call is: `accept(s, name, namelen)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Bind

Format



Use the Bind function to assign a unique local name (network address) to a socket. When you create a socket with the Socket function, the socket does not have a name associated with it, but it does belong to an addressing family. The form of the name you assign to the socket with the Bind function depends on the addressing family. The Bind function also allows servers to specify the network interfaces from which they want to receive UDP packets and TCP connection requests.

Operands

socketid
is the identifier of the socket.

socketname
is the local name (network address) to be assigned to the socket. The name consists of three parts:

domain
The addressing family of the socket. This must be AF_INET (or the equivalent decimal value 2).

portid
The port number to which the socket must bind.

ipaddress
The IP address of the socket. This must be one of the following:

- Dotted decimal address of the local network interface
- INADDR_BROADCAST
- INADDR_ANY

Responses If successful, this function returns a string containing only return code 0. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

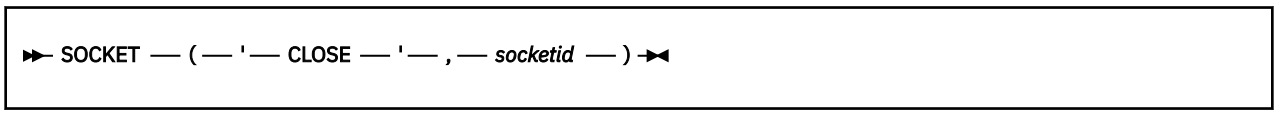
Call	Return Values
`Socket('Bind',5,'AF_INET 1234 128.228.1.2')`	`'0'`

The C socket call is: `bind(s, name, namelen)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Close

Format



Use the Close function to shut down a socket and free the resources allocated to it. If the socket ID refers to an open TCP connection, the connection is closed. If a stream socket is closed when there is input data queued, the TCP connection is reset rather than closed.

Operands

socketid
is the identifier of the socket to be closed.

Usage Notes The SO_LINGER socket option, which is set by the SetSockOpt function, can be used to control how unsent output data is handled when a stream socket is closed. See "SetSockOpt" on page 299.

Responses If successful, this function returns a string containing only return code 0. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

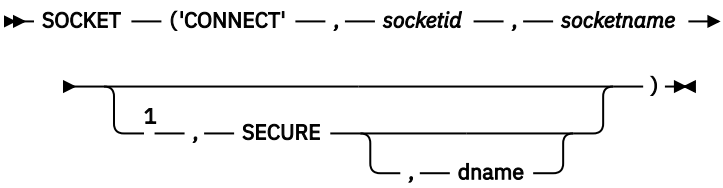
Call	Return Values
`Socket('Close',6)`	`'0'`

The C socket call is: `close(s)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Connect

Format



Notes:

[1] The fourth and fifth operand are only allowed if the socketset has been initialized for SSL support.

Use the Connect function to establish a connection between two stream sockets or to specify the default peer for a datagram socket.

When called for a stream socket, Connect performs two tasks:

1. If the Bind function has not been called for the socket used to originate the request, Connect completes the bind.
2. Connect then attempts to establish a connection to the other socket.

If the originating stream socket is in blocking mode, Connect blocks the caller until the connection is established or an error is received. If the originating socket is in nonblocking mode, Connect ends with return code 1102 (EINPROGRESS) or another return code indicating an error

Operands

socketid
is the identifier of the socket originating the connection request.

socketname
is the name (network address) of the socket to which a connection will be attempted. The name consists of three parts:

domain
The addressing family of the socket. This must be AF_INET (or the equivalent decimal value 2).

portid
The port number of the socket.

ipaddress
The IP address of the socket.

SECURE
specifies to perform the SSL handshake. It is performed as a client with or without client authentication.

dname
specifies a character string that is the member name of the desired entry (certificate) in the keyring library. If nothing is specified, the first key ring entry is used.

```
fc = SOCKET('CONNECT',newsocketid,'AF_INET 5678 9.164.111.11', ,
            'SECURE','SAMPLE')
```

Responses If successful, this function returns a string containing only return code 0. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

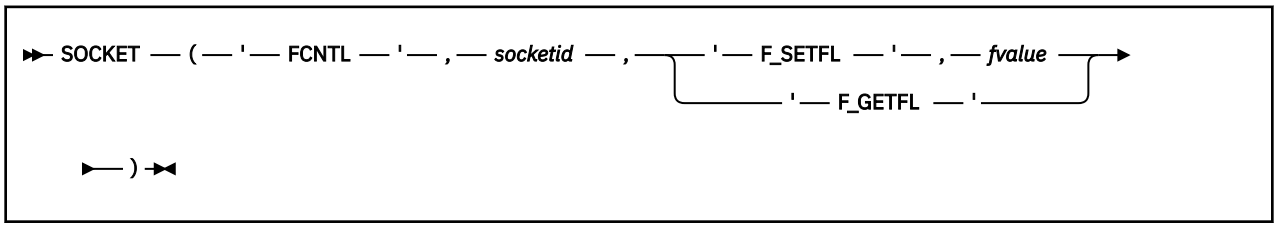
Call	Return Values
`Socket('Connect',5,'AF_INET 1234 128.228.1.2')`	`'0'`
`Socket('Connect',5,'AF_INET 1234 CUNYVM')`	`'0'`
`Socket('Connect',5,'AF_INET 1234 CUNYVM.CUNY.EDU')`	`'0'`
`Socket('Connect',5,'AF_INET 1234 128.228.1.2','SECURE','SAMPLE')`	`'0'`

The C socket call is: `connect(s, name, namelen)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Fcntl

Format



Use the Fcntl function to set blocking or nonblocking mode for a socket, or to get the setting for the socket.

Operands

socketid
is the identifier of the socket.

F_SETFL
sets the status flags for the socket. One flag, FNDELAY, can be set.

F_GETFL
gets the flag status for the socket. One flag, FNDELAY, can be retrieved.

fvalue
is the operating characteristic. The following values are valid:

NON-BLOCKING or FNDELAY
Turns the FNDELAY flag on, which marks the socket as being in nonblocking mode. If data is not present on calls that can block, such as Read and Recv, Fcntl returns error code 1102 (EWOULDBLOCK).

Responses If successful, this function returns a string containing return code 0. If F_GETFL is specified, the operating characteristic status is also returned. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

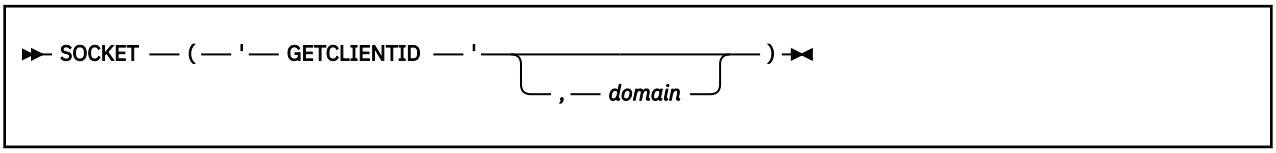
Call	Return Values
`Socket('Fcntl',5,'F_SETFL','NON-BLOCKING')`	`'0'`
`Socket('Fcntl',5,'F_GETFL')`	`'0 NON-BLOCKING'`

The C socket call is: `fcntl(s, cmd, data)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

GetClientId

Format



Use the GetClientId function to get the identifier by which the calling program is known to the TCP/IP virtual machine.

Operands

domain
is the addressing family. This must be one of the following:

- AF_INET (or the equivalent decimal value 2); AF_INET is the default.

Responses If successful, this function returns a string containing return code 0 and the TCP/IP identifier. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

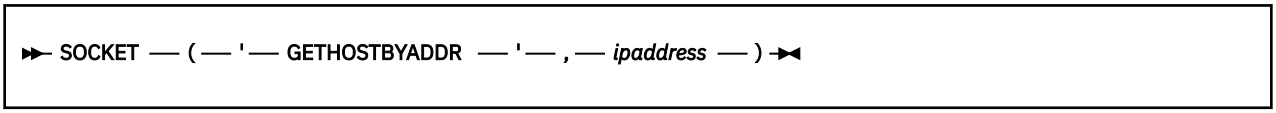
Call
Return Values

`Socket('GetClientId')`
`'0 AF_INET USERID1 myId'`

The C socket call is: `getclientid(domain, clientid)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

GetHostByAddr



Use the GetHostByAddr function to get the host name for a specified IP address. The name is resolved through a name server, if one is present.

Operands

ipaddress
is the IP address of the host, in dotted-decimal notation.

Responses If successful, this function returns a string containing return code 0 and the full host name. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

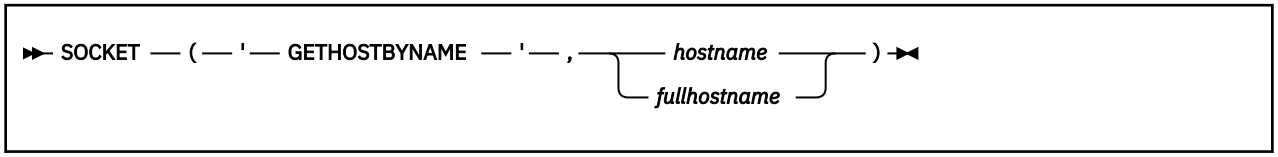
Examples

Call	**Return Values**
`Socket('GetHostByAddr','128.228.1.2')`	`'0 CUNYVM.CUNY.EDU'`

The C socket call is: `gethostbyaddr(addr, addrlen, domain)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

GetHostByName



Use the GetHostByName function to get the IP address for a specified host name. The name is resolved through a name server, if one is present. GetHostByName returns all the IP addresses for a multihome host.

Operands

hostname
is the host processor name as a character string.

fullhostname
is the fully qualified host name in the form *hostname.domainname*.

Responses If successful, this function returns a string containing return code 0 and an IP address list. The addresses in the list are separated by blanks. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

Call	Return Values
`Socket('GetHostByName','CUNYVM')`	`'0 128.228.1.2'`
`Socket('GetHostByName','CUNYVM.CUNY.EDU')`	`'0 128.228.1.2'`

The C socket call is: `gethostbyname(name)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

GetHostId

Use the GetHostId function to get the IP address for the current host. This address is the default home IP address.

Responses If successful, this function returns a string containing return code 0 and the IP address for the host. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

Call
Return Values

`Socket('GetHostId')`
`'0 9.4.3.2'`

The C socket call is: `gethostid()`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

GetHostName

Use the GetHostName function to get the name of the host processor on which the program is running.

Responses If successful, this function returns a string containing return code 0 and the name of the host processor. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

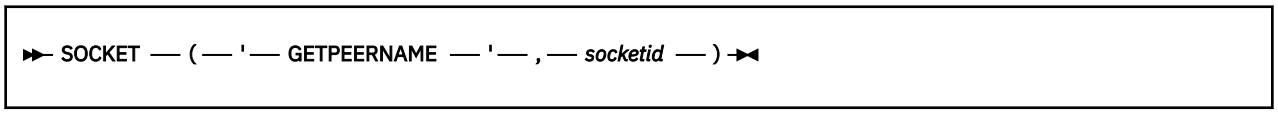
Call
Return Values

`Socket('GetHostName')`
`'0 ZURLVM1'`

The C socket call is: `gethostname(name, namelen)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

GetPeerName



Use the GetPeerName function to get the name of the peer connected to a socket.

Operands

socketid
is the identifier of the socket.

Responses If successful, this function returns a string containing return code 0 and the name of the peer. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

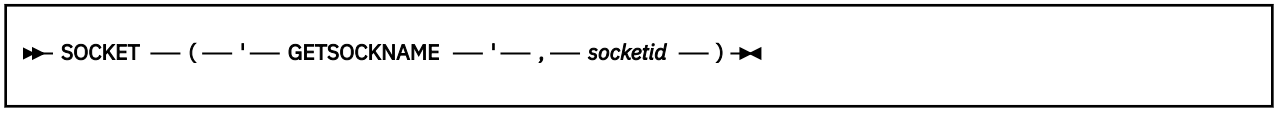
Call
Return Values

`Socket('GetPeerName',6)`
`'0 AF_INET 1234 128.228.1.2'`

The C socket call is: `getpeername(s, name, namelen)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

GetSockName



Use the GetSockName function to get the name to which a socket was bound. Stream sockets are not assigned a name until after a successful call to the Bind, Connect, or Accept function.

Operands

socketid
is the identifier of the socket.

Responses If successful, this function returns a string containing return code 0 and the socket name (network address, consisting of domain, port ID, and IP address). If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

Call
Return Values

```
Socket('GetSockName',7)
  '0 AF_INET 5678 9.4.3.2'
```

The C socket call is: `getsockname(s, name, namelen)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

GetSockOpt

Use the GetSockOpt function to get the status of options and other data associated with an AF_INET socket. Most socket options are set with the SetSockOpt function. Multiple options can be associated with each socket. You must specify each option or other item you want to query on a separate call.

Operands

socketid
is the identifier of the socket.

level
is the protocol level for which the socket option or other data is being queried. SOL_SOCKET is supported.

optname
is a value that indicates the type of information requested:

Value
Description

SO_LINGER
Gets the status of the SO_LINGER option, which controls whether the Close function will linger if data is present. The setting can be On or Off.

- If SO_LINGER is On and there is unsent data present when Close is called, the calling application is blocked until the data transmission is complete or the connection has timed out.
- If SO_LINGER is Off, a call to Close returns without blocking the caller. TCP/IP still tries to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because TCP/IP repeats the Send request for only a specified period of time.

In the return string, an On setting is followed by the number of seconds that TCP/IP continues trying to send the data after Close is called.

Responses If successful, this function returns a string containing return code 0 and the option status or other requested value. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

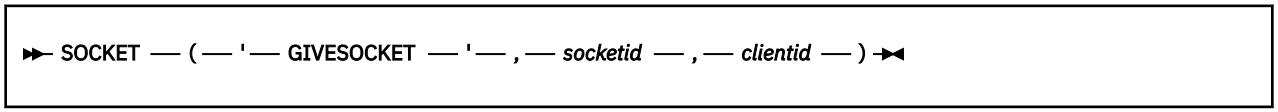
Examples

Call	Return Values
`Socket('GetSockOpt',5,'Sol_Socket','So_Linger')`	`'0 On 60'`

The C socket call is: `getsockopt(s, level, optname, optval, optlen)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

GiveSocket



Use the GiveSocket function to transfer a socket to another application. GiveSocket makes the socket available to a TakeSocket call issued by another application using the same TCP/IP server. Any connected stream socket can be given. GiveSocket is typically used by a concurrent server program that obtains sockets using the Accept function and then gives them to child server programs that handle one socket at a time.

Operands

socketid
is the identifier of the socket to be given.

clientid
is the identifier for the application that will be taking the socket. This consists of three parts:

domain
The addressing family. This must be AF_INET (or the equivalent decimal value 2).

userid
The VM/ESA user ID of the virtual machine in which the taking application is running.

subtaskid
The subtask ID used on the taking application. This is optional.

The method for obtaining the taking application's client ID is not defined by TCP/IP.

Responses If successful, this function returns a string containing only return code 0. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

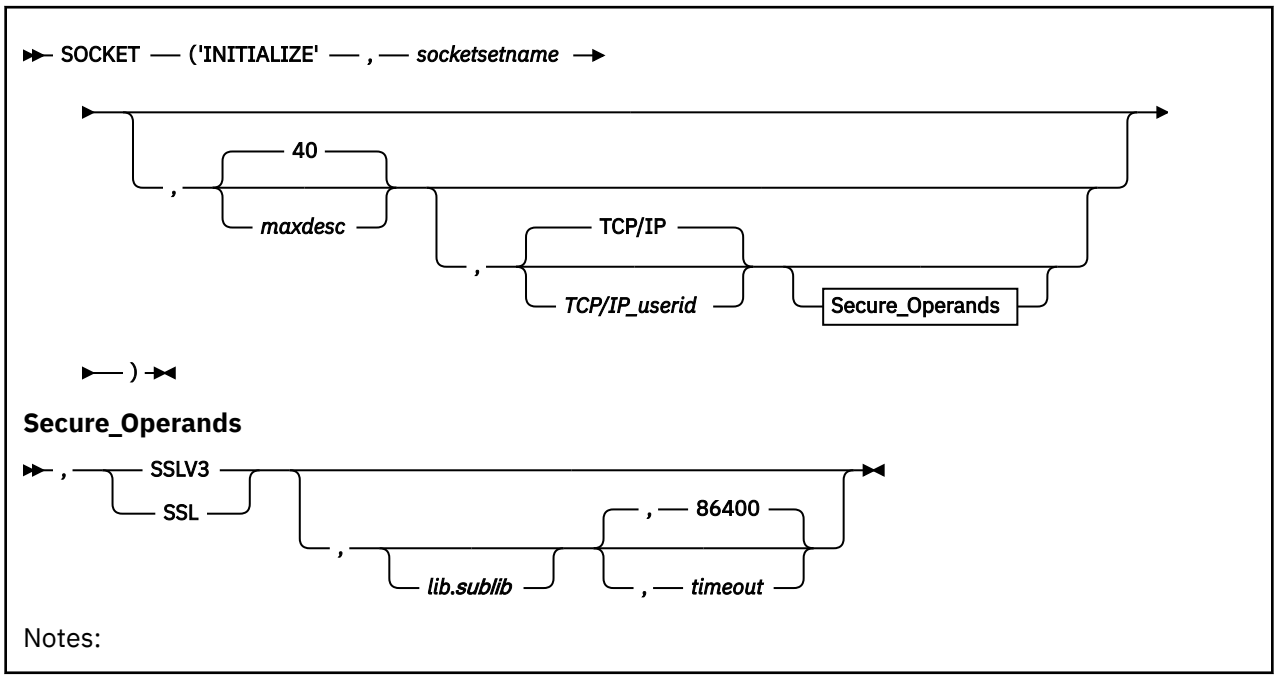
Examples

Call	Return Values
`Socket('GiveSocket',6,'AF_INET USERID2 hisId')`	`'0'`

The C socket call is: `givesocket(s, clientid)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Initialize



Notes:

Use the Initialize function to define a socket set. If the function is successful, this socket set becomes the active socket set.

Operands

socketsetname
is the name of the socket set. The name can be up to eight printable characters; it cannot contain blanks.

maxdesc
is the maximum number of preallocated sockets in the socket set. The number can be between 1 and the maximum number supported by TCP/IP for VM. The default is 40.

TCP/IP_userid
is the user ID of the TCP/IP server machine. If not specified, a value of 'TCPIP' is used.

SSLV3 | SSL
enables usage of the SSL support and identifies the security protocols that are to be used. A secure socket communication is only possible, if the VSE/ESA host has been configured for SSL support (see topic "*Configuring Your VSE/ESA Host for SSL*" in the z/VSE e-business Connectors User's Guide).

lib.sublib
identifies the sublibrary used for keys and certificates (the VSE Keyring Library, see skeleton SKSSLKEY in ICCF library 59). If nothing is specified, the private key and certificates are read from the default sequential disk files.

timeout
specifies the number of seconds for the SSLV3 session identifier to expire. The range is 0-86400 seconds (1 day). Default is 86400 seconds.

```
fc = SOCKET('INITIALIZE','SERVMIRR',,,'SSLV3','CRYPTO.KEYRING',86400)
```

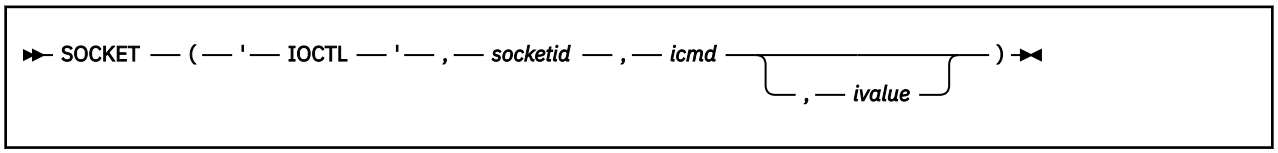
Responses If successful, this function returns a string containing return code 0, the name (subtask ID) of the initialized socket set, the maximum number of preallocated sockets in the socket set, and the user ID of the TCP/IP server machine. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

Call	Return Values
`Socket('Initialize','myId')`	`'0 myId 40 TCPIP'`
`Socket('Initialize','myId'),,,'SSLV3','CRYPTO,KEYRING',8640 0)`	`'0 myId 40 TCPIP'`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Ioctl



Use the Ioctl function to control the operating characteristics of a socket.

Operands

socketid
is the identifier of the socket.

icmd
is the operating characteristics command to be issued:

Command
Description

FIONBIO
Sets or clears nonblocking for socket I/O. You specify On or Off in *ivalue*.

ivalue
is the operating characteristics value. This value depends on the value specified for *icmd*. The *ivalue* parameter can be used as input or output or both on the same call.

Responses If successful, this function returns a string containing return code 0 and operating characteristics information. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

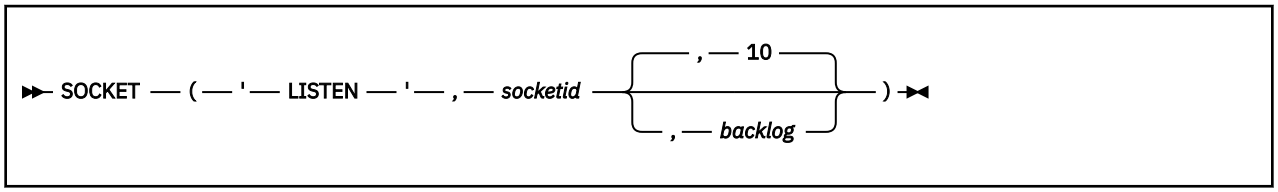
Examples

Call	Return Values
`Socket('Ioctl',5,'FionBio','On')`	`'0'`

The C socket call is: `ioctl(s, cmd, data)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Listen



Use the Listen function to transform an active stream socket into a passive socket. Listen performs two tasks:

1. If the Bind function has not been called for the socket, Listen completes the bind. (The domain, port ID, and IP address are set to AF_INET, INPORT_ANY, and INADDR_ANY.)
2. Listen creates a connection request queue for incoming connection requests. After the queue is full, additional connection requests are ignored.

Calling the Listen function indicates a readiness to accept client connection requests. After Listen is called, the socket can never be used as an active socket to initiate connection requests. Calling Listen is the third of four steps that a server performs to accept a connection. It is called after allocating a stream socket with the Socket function, and after binding a name to the socket with the Bind function, but before calling the Accept function.

Operands

socketid
is the identifier of the socket.

backlog
is the number of pending connection requests. This number is an integer between 0 and 10. The default is 10.

Responses If successful, this function returns a string containing only return code 0. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

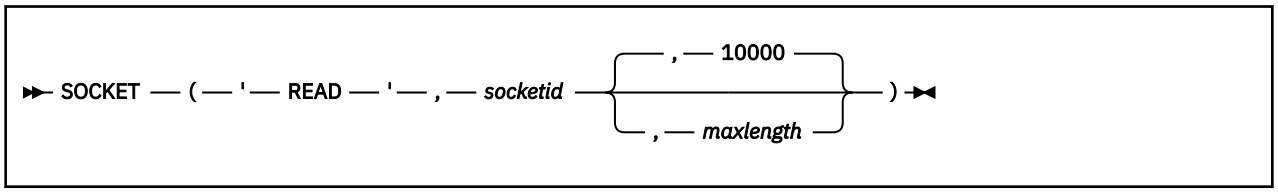
Call
Return Values

```
Socket('Listen',5,10)
  '0'
```

The C socket call is: `listen(s, backlog)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Read



Use the Read function to read data on a connected socket, up to a specified maximum number of bytes. This is the conventional TCP/IP read data operation. If less than the requested number of bytes is available, Read returns the number currently available. If data is not available at the socket, Read waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode.

For datagram sockets, Read returns the entire datagram that was sent, providing that the datagram fits into the specified buffer.

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if programs A and B are connected with a stream socket, and program A sends 1000 bytes, each call to this function can return any number of bytes, up to the entire 1000 bytes. The number of bytes returned is contained in the return values string. Therefore, programs using stream sockets should place this call in a loop that repeats until all the data has been received. If the length in the return values string is zero, the other side of the call has closed the stream socket.

Operands

socketid
is the identifier of the socket.

maxlength
is the maximum length of data to be read. This is a number of bytes between 1 and 100000. The default is 10000.

Responses If successful, this function returns a string containing return code 0, the length of the data read, and the data read. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

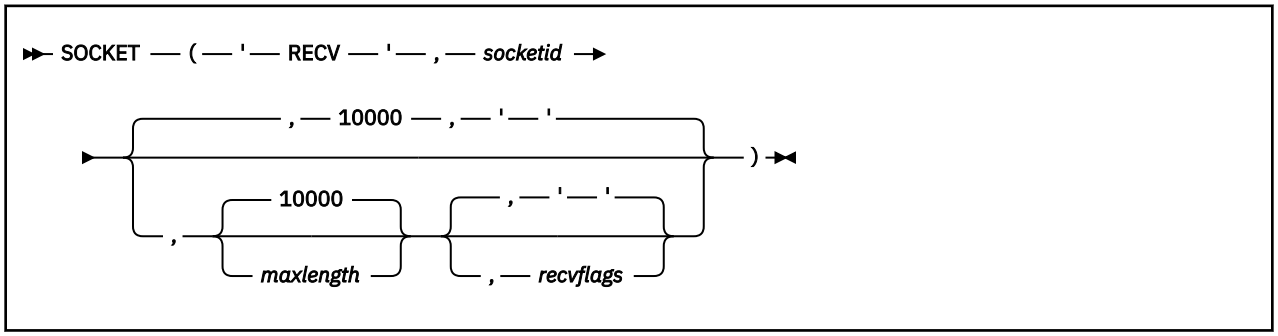
Call
Return Values

```
Socket('Read',6)
  '0 21 This is the data line'
```

The C socket call is: `read(s, buf, len)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Recv



Use the Recv function to receive data on a connected socket, up to a specified maximum number of bytes. By specifying option flags, you can also:

On a datagram socket, if more than the number of bytes requested is available, Recv discards the excess bytes. If less than the number of bytes requested is available, Recv returns the number of bytes currently available. If data is not available at the socket, Recv waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode.

On a stream socket, if the data length in the return string is zero, the other side has closed the socket.

Operands

socketid
is the identifier of the socket.

maxlength
is the maximum length of data to be received. This is a number of bytes between 1 and 100000. The default is 10000.

recvflags
are flags that control the Recv operation:

''
Receive the data. No flag is set. This is the default.

Responses If successful, this function returns a string containing return code 0, the length of the data received, and the data received. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

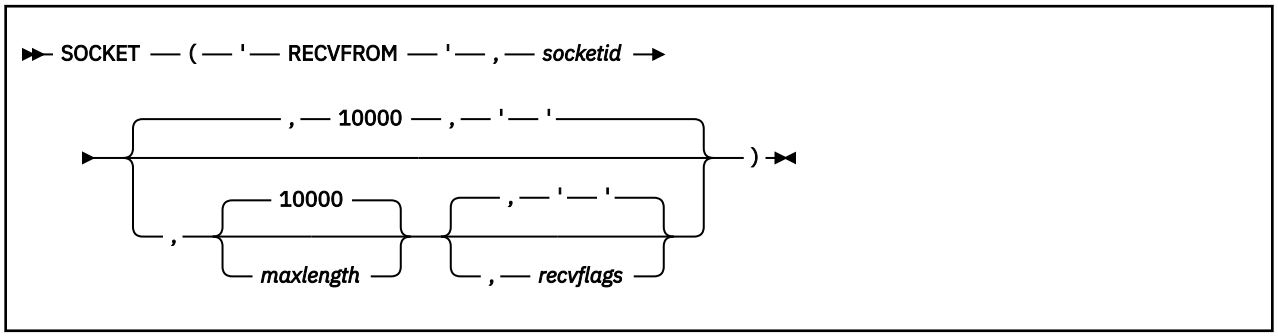
Examples

Call	Return Values
`Socket('Recv',6)`	`'0 21 This is the data line'`
`Socket('Recv',6,,'PEEK OOB')`	`'0 24 This is out-of-band data'`

The C socket call is: `recv(s, buf, len, flags)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

RecvFrom



Use the RecvFrom function to receive data on a socket, up to a specified maximum number of bytes, and get the sender's address.

On a datagram socket, if more than the number of bytes requested is available, RecvFrom discards the excess bytes. If less than the number of bytes requested is available, RecvFrom returns the number of bytes available.

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if programs A and B are connected with a stream socket, and program A sends 1000 bytes, each call to RecvFrom can return any number of bytes, up to the entire 1000 bytes. The number of bytes returned is specified in the return values string. Therefore, programs using stream sockets should place RecvFrom in a loop that repeats until all the data has been received. If a data length of zero is returned in the return values string, the socket has been closed by the other side.

If data is not available at the socket, RecvFrom waits for data to arrive and blocks the caller, unless the socket is in nonblocking mode.

Operands

socketid
is the identifier of the socket.

maxlength
is the maximum length of data to be received. This is a number of bytes between 1 and 100000. The default is 10000.

recvflags
are flags that control the RecvFrom operation:

''
Receive the data. No flag is set. This is the default.

Responses If successful, this function returns a string containing return code 0, the network address (domain, remote port, and remote IP address) of the sender, the length of the data received, and the data received. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

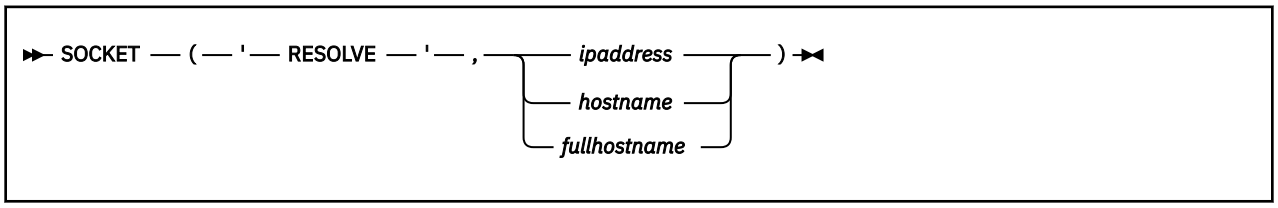
Call
Return Values

Socket('RecvFrom',6)
'0 AF_INET 5678 9.4.3.2 9 Data line'

The C socket call is: `recvfrom(s, buf, len, flags, name, namelen)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Resolve



Use the Resolve function to resolve the host name through a name server, if one is present.

Operands

ipaddress
is the IP address of the host, in dotted-decimal notation.

hostname
is the host processor name as a character string.

fullhostname
is the fully qualified host name in the form *hostname.domainname*.

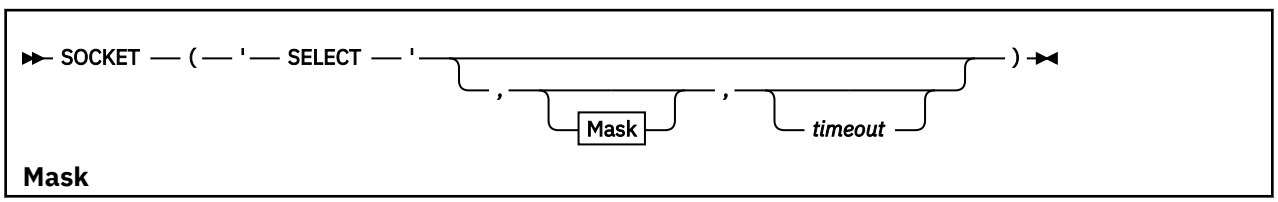
Responses If successful, this function returns a string containing return code 0, the IP address of the host, and the full host name. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

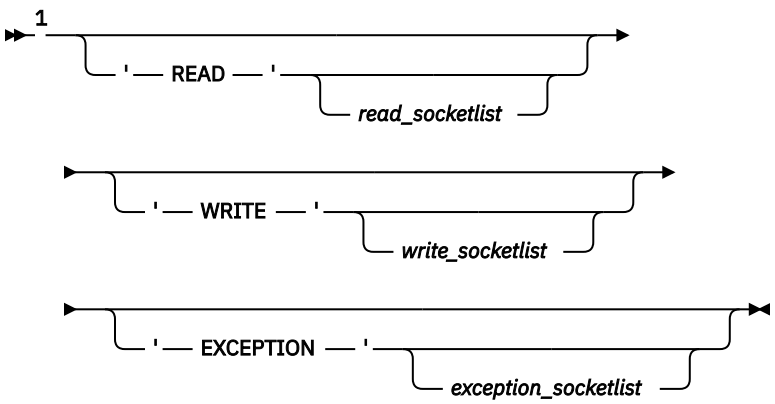
Call	Return Values
`Socket('Resolve','128.228.1.2')`	`'0 128.228.1.2 CUNYVM.CUNY.EDU'`
`Socket('Resolve','CUNYVM')`	`'0 128.228.1.2 CUNYVM.CUNY.EDU'`
`Socket('Resolve','CUNYVM.CUNY.EDU')`	`'0 128.228.1.2 CUNYVM.CUNY.EDU'`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Select



Mask



Notes:

[1] You can specify the lists of socket descriptors in any order.

Use the Select function to monitor activity on specified socket IDs to see if any of them are ready for reading or writing or have an exception condition pending. Select does not check for the order of event completion.

A Close on the other side of a socket connection is not reported as an exception, but as a Read event that returns zero bytes of data.

When Connect is called with a socket in nonblocking mode, the Connect call ends and returns code 1102 (EWOULDBLOCK). The completion of the connection setup is then reported as a Write event on the socket.

When Accept is called with a socket in nonblocking mode, the Accept call ends and returns code 1102 (EWOULDBLOCK). The availability of the connection request is reported as a Read event on the original socket, and Accept should be called only after the Read has been reported.

Operands

READ *read_socketidlist*
specifies a list of socket descriptors to be checked to see if they are ready for reading. A socket is ready for reading when incoming data is buffered for it or, for a listening socket, when a connection request is pending. Select returns the socket ID in the return value string if a call to read from that socket will not block. If you do not need to test any sockets for reading, you can pass a null for the list.

WRITE *write_socketidlist*
specifies a list of socket descriptors to be checked to see if they are ready for writing. A socket is ready for writing when there is buffer space for outgoing data. Select returns the socket ID in the return value string if a call to write to that socket will not block. If you do not need to test any sockets for writing, you can pass a null for the list.

EXCEPTION *exception_socketidlist*
specifies a list of socket descriptors to be checked to see if they have an exception condition pending. A socket has an exception condition pending if it has received out-of-band data or if another program has successfully taken the socket using the TakeSocket function. If you do not need to test any sockets for exceptions pending, you can pass a null for the list.

timeout
is a positive integer indicating the maximum wait time in seconds. The default is FOREVER.

Responses If successful, this function returns a string containing return code 0, the number of sockets that have completed events, the list of socket IDs that are ready for reading, the list of socket IDs that are ready for writing, and the list of socket IDs that have an exception pending. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

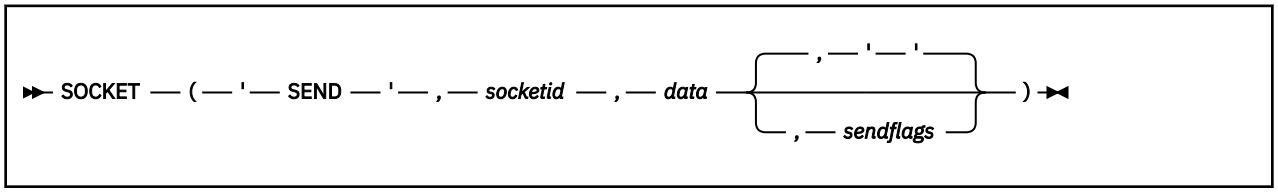
Examples

Call	Return Values
`Socket('Select','Read 5 Write Exception',10)`	`'0 1 READ 5 WRITE EXCEPTION'`

The C socket call is: `select(nfds, readfds, writefds, exceptfds, timeout)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Send



Use the Send function to send data on a connected socket.

If Send cannot send the number of bytes of data that is requested, it waits until sending is possible. This blocks the caller, unless the socket is in nonblocking mode. For datagram sockets, the socket should not be in blocking mode.

Operands

socketid
is the identifier of the socket.

data
is the message string to be sent.

sendflags
are flags that control the Send operation:

''
Send the data. No flag is set. This is the default.

Responses If successful, this function returns a string containing return code 0 and the length of the data sent. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

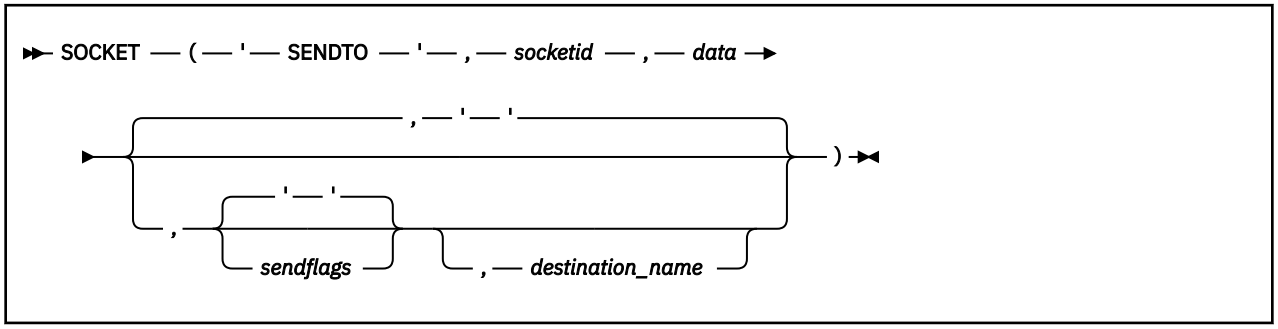
Examples

Call	Return Values
`Socket('Send',6,'Some text')`	`'0 9'`

The C socket call is: `send(s, buf, len, flags)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

SendTo



Use the SendTo function to send data on a socket. This function is similar to the Send function, except that you can specify a destination address to send datagrams on a UDP socket, whether the socket is connected or unconnected.

For datagram sockets, the socket should not be in blocking mode.

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if a program is required to send 1000 bytes, each call to the SendTo function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in the return values string. Therefore, programs using stream sockets should place SendTo in a loop that repeats the call until all the data has been sent.

Operands

socketid
is the identifier of the socket.

data
is the message string to be sent.

sendflags
are flags that control the SendTo operation:

''
Send the data. No flag is set. This is the default.

destination_name
is the destination network address, which consists of three parts:

domain
The addressing family. This must be AF_INET (or the equivalent decimal value 2).

portid
The port number.

ipaddress
The IP address.

Responses If successful, this function returns a string containing return code 0 and the length of the data sent. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

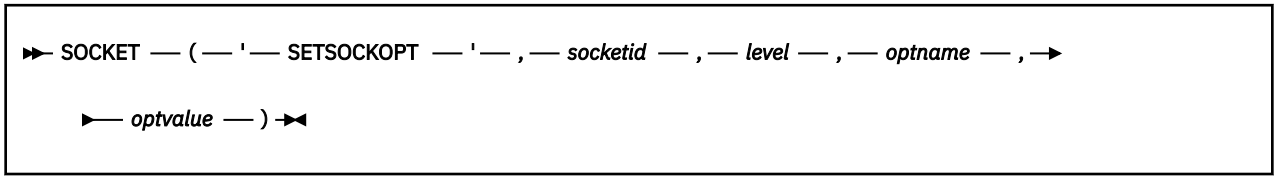
Examples

Call	Return Values
`Socket('SendTo',6,'some text',,'AF_INET 5678 9.4.3.2')`	`'0 9'`

The C socket call is: `sendto(s, buf, len, flags, name, namelen)`

Messages and Return Codes For a list of REXX Sockets system messages, see “REXX Sockets System Messages” on page 307. For a list of REXX Sockets return codes, see “REXX Sockets Return Codes” on page 307.

SetSockOpt



Use the SetSockOpt function to set the options associated with an AF_INET socket.

The *optvalue* parameter is used to pass data used by the particular set command. The *optvalue* parameter points to a buffer containing the data needed by the set command. The *optvalue* parameter is optional and can be set to 0, if data is not needed by the command.

Operands

socketid
is the identifier of the socket.

level
is the protocol level for which the socket option is being set. SOL_SOCKET and IPPROTO_TCP are supported. All *optname* values beginning with "SO_" are for protocol level SOL_SOCKET and are interpreted by the general socket code. All *optname* values beginning with "TCP_" are for protocol level IPPROTO_TCP and are interpreted by the TCP/IP internal code.

optname
is the socket option to be set:

Option
Description

SO_LINGER
Controls whether the Close function will linger if data is present:

- If SO_LINGER is On and there is unsent data present when Close is called, the calling application is blocked until the data transmission completes or the connection times out.
- If SO_LINGER is Off, a call to Close returns without blocking the caller. TCP/IP still tries to send the data. Although this transfer is usually successful, it cannot be guaranteed, because TCP/IP repeats the Send request for only a specified period of time.

optvalue
is the option setting.

For the SO_LINGER option, you can specify On *n*, *n*, or Off. If you specify only *n*, On is implied. The value *n* is the number of seconds that TCP/IP should continue trying to send the data after the Close function is called. If On is selected, the default number is 120.

Responses If successful, this function returns a string containing only return code 0. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

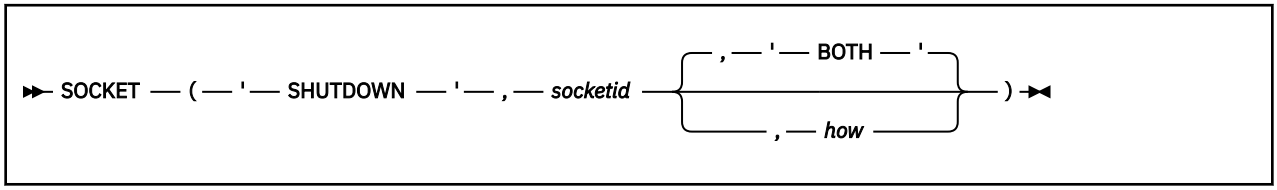
Examples

Call	Return Values
`Socket('SetSockOpt',5,'Sol_Socket','So_Linger',60)`	`'0'`

The C socket call is: `setsockopt(s, level, optname, optval, optlen)`

Messages and Return Codes For a list of REXX Sockets system messages, see “REXX Sockets System Messages” on page 307. For a list of REXX Sockets return codes, see “REXX Sockets Return Codes” on page 307.

ShutDown



Use the ShutDown function to shut down all or part of a duplex connection.

Operands

socketid
is the identifier of the socket.

how
sets the communication direction to be shut down:

BOTH
Disables further receive-type and send-type operations on the socket, ending communication from and to the socket. This is the default.

Responses If successful, this function returns a string containing only return code 0. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

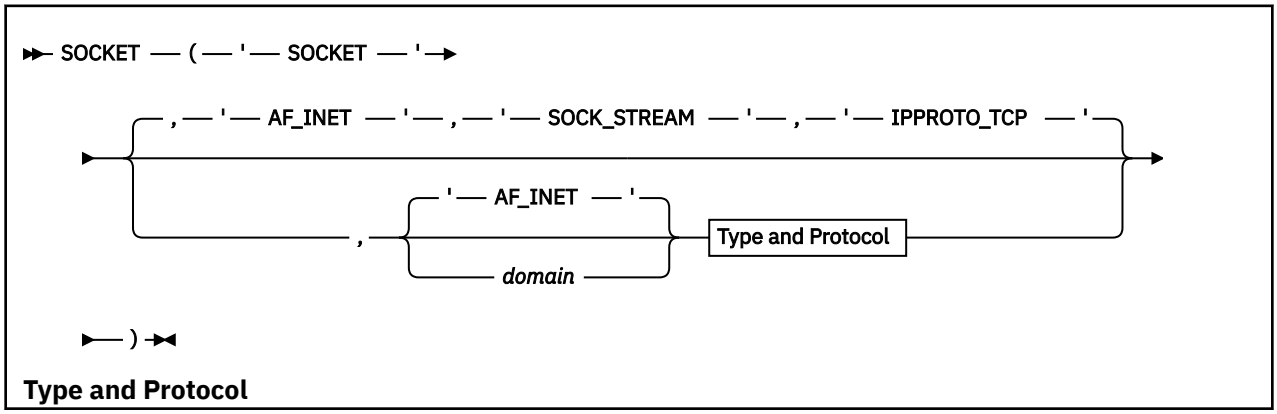
Call
Return Values

```
Socket('ShutDown',6,'BOTH')
  '0'
```

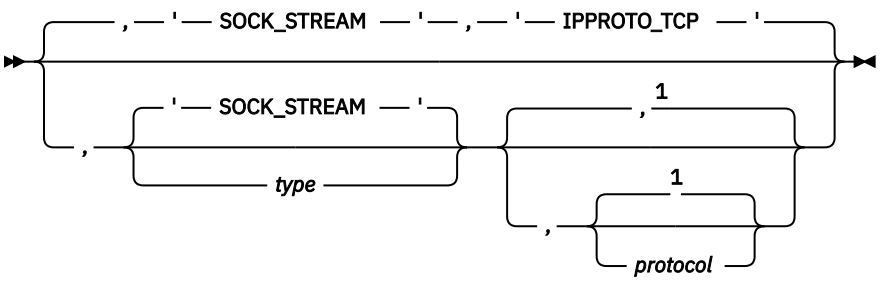
The C socket call is: `shutdown(s, how)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Socket



Type and Protocol



Notes:

[1] The default protocol depends on the domain and socket type.

Use the Socket function to create a socket in the active socket set. Different types of sockets provide different communication services.

Operands

domain
is the communications domain in which communication is to take place. This parameter specifies the addressing family (format of addresses within a domain) being used. This value must be AF_INET (or the equivalent integer value 2), which indicates the internet domain. This is also the default.

type
is type of socket to be created. The supported types are:

Type
Description

SOCK_STREAM
The abbreviated form STREAM is also permitted (or the equivalent integer value 1). This type of socket provides sequenced, two-way byte streams that are reliable and connection-oriented. Bytes are guaranteed to arrive, arrive only once, and arrive in the order sent. AF_INET stream sockets also support a mechanism for sending and receiving out-of-band data.

SOCK_DGRAM
The abbreviated form DATAGRAM is also permitted (or the equivalent integer value 2). This type of socket provides datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times.

The default type is SOCK_STREAM.

protocol
is the protocol to be used with the socket.

For stream and datagram sockets, you should set this field to 0 to allow TCP/IP to assign the default protocol for the domain and socket type selected. For the AF_INET domain, the default protocols are:

- IPPROTO_TCP for stream sockets
- IPPROTO_UDP for datagram sockets

Responses If successful, this function returns a string containing return code 0 and the identifier (socket ID) of the new socket. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

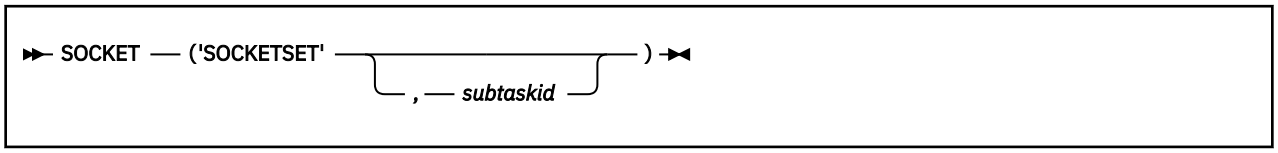
Call
Return Values

```
Socket('Socket')
  '0 5'
```

The C socket call is: `socket(domain, type, protocol)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

SocketSet



Use the SocketSet function to get the name (subtask ID) of the active socket set. If you specify a subtask ID on the call, that socket set becomes the active socket set.

Operands

subtaskid
is the name of a socket set. The name can be up to eight printable characters; it cannot contain blanks.

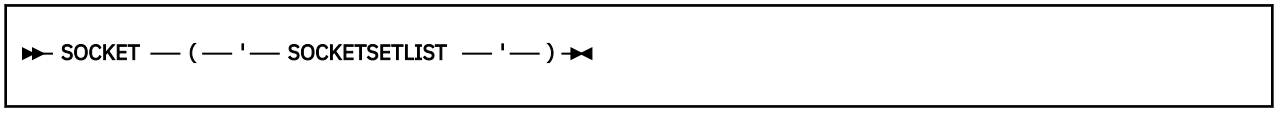
Responses If successful, this function returns a string containing return code 0 and the subtask ID of the active socket set. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

Call	Return Values
`Socket('SocketSet','firstId')`	`'0 firstId'`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

SocketSetList



Use the SocketSetList function to get a list of the names (subtask IDs) of all the available socket sets in the current order of the stack.

Responses If successful, this function returns a string containing return code 0, the subtask ID of the active socket set, and the subtask IDs of all the other available socket sets (if any) in the current order of the stack. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

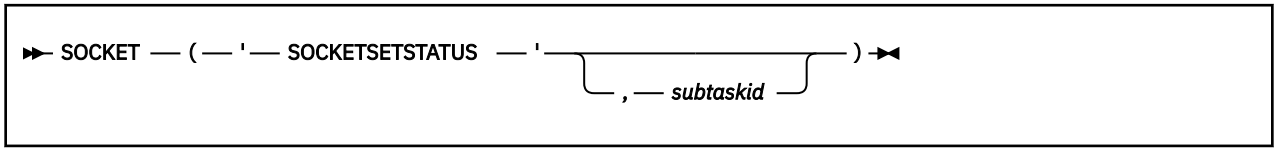
Examples

Call
Return Values

```
Socket('SocketSetList')
  '0 myId firstId'
```

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

SocketSetStatus



Use the SocketSetStatus function to get the status of a socket set. If you do not specify the name (subtask ID) of the socket set, the active socket set is used. If the socket set is connected, this function returns the number of free sockets and the number of allocated sockets in the socket set. If the socket set is severed, the reason for the TCP/IP sever is also returned. Initialized socket sets should be in connected status, and uninitialized socket sets should be in free status.

A socket set that is initialized but is not in connected status must be terminated before the subtask ID can be reused.

Operands

socketsetname
is the name of a socket set. The name can be up to eight printable characters; it cannot contain blanks.

Responses If successful, this function returns a string containing return code 0, the subtask ID of the socket set, and the status of the socket set. Connect and sever information may also be returned. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

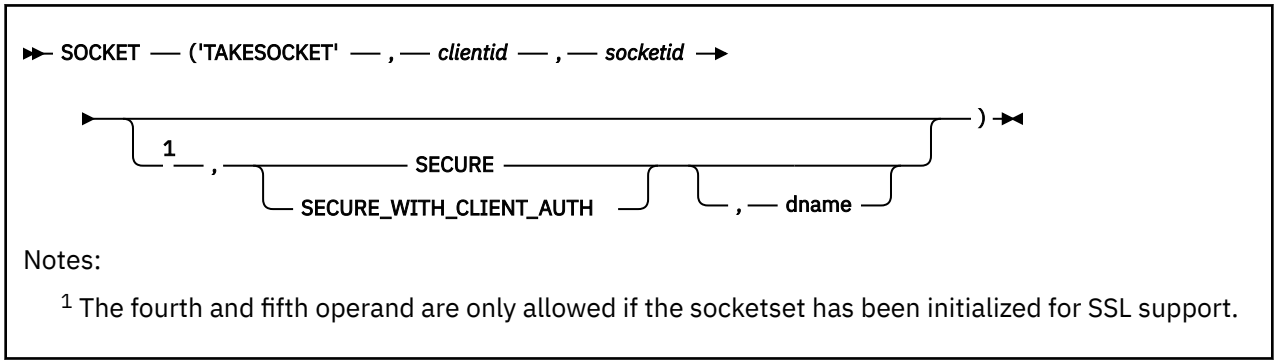
Examples

Call
Return Values

```
Socket('SocketSetStatus')
  '0 myId Connected Free 17 Used 23'
```

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

TakeSocket



Notes:

[1] The fourth and fifth operand are only allowed if the socketset has been initialized for SSL support.

Use the TakeSocket function to acquire a socket from another application. The giving application must have already issued a GiveSocket call. After Takesocket completes successfully, the giving application must close the socket.

Operands

clientid
is the identifier for the application that is giving the socket. This consists of three parts:

domain
The addressing family. This must be AF_INET (or the equivalent integer value 2).

userid
The VM/ESA user ID of the virtual machine in which the giving application is running.

subtaskid
The subtask ID used on the giving application.

The method for obtaining the giving application's client ID is not defined by TCP/IP.

socketid
is the identifier of the socket on the giving application. The method for obtaining this value is not defined by TCP/IP.

SECURE | SECURE_WITH_CLIENT_AUTH
specifies to perform the SSL handshake. With SECURE the SSL server handshake is performed, with SECURE_WITH_CLIENT_AUTH the SSL handshake is performed as a server that requires client authentication.

dname
specifies a character string that is the member name of the desired entry (certificate) in the keyring library. If nothing is specified, the first keyring entry is used.

```
fc = Socket('Takesocket',ClientId,SocketNr,'SECURE_WITH_CLIENT_AUTH','SAMPLE')
```

See Table 10 on page 280.

Responses If successful, this function returns a string containing return code 0 and a new socket ID (the identifier assigned to the socket on the taking application). If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

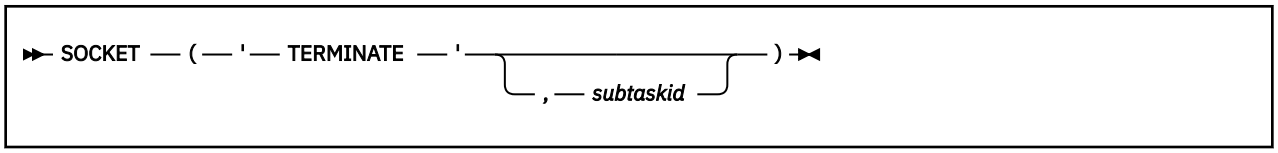
Examples

Call	Return Values
`Socket('TakeSocket','AF_INET USERID1 myId',6)`	`'0 7'`
`Socket('TakeSocket','AF_INET USERID1 myId',6, ,` `'SECURE_WITH_CLIENT_AUTH','SAMPLE')`	`'0 7'`

The C socket call is: `takesocket(clientid, hisdesc)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Terminate



Use the Terminate function to close all the sockets in a socket set and release the socket set. If you do not specify a socket set, the active socket set is terminated. If the active socket set is terminated, the next socket set in the stack (if available) becomes the active socket set.

Operands

subtaskid
is the name of the socket set. The name can be up to eight printable characters; it cannot contain blanks.

Responses If successful, this function returns a string containing return code 0 and the name (subtask ID) of the terminated socket set. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

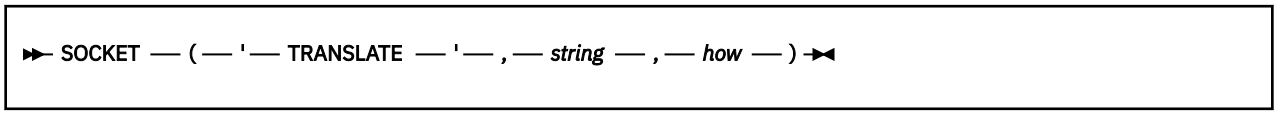
Call

Return Values

```
Socket('Terminate','myId')
  '0 myId'
```

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Translate



Use the Translate function to translate data from one type of notation to another.

Operands

string
is a character string that contains the data to be translated.

how
indicates the type of translation to be done. The supported types are (case is not significant in these values):

Type
Description

To_Ascii or Ascii
Translates the specified REXX character string to ASCII

To_Ebcdic or Ebcdic
Translates the specified REXX hexadecimal string to EBCDIC

To_IP_Address or To_IPaddress or IPaddress
Translates the specified dotted-decimal IP address into a 4-byte hexadecimal notation, or the specified 4-byte hexadecimal IP address into dotted-decimal notation

To_SockAddr_In or SockAddr_In
Translates the specified sockaddr_in structure from human-readable notation (a three-part character string containing AF_INET, the decimal port value, and either an IP address or a partially- or fully-qualified host name) into a 16-byte hexadecimal notation, or from 16-byte hexadecimal notation into a human-readable notation

Responses If successful, this function returns a string containing return code 0, the length of the translated string, and the translated string. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Usage Notes

1. In addition to the blanks between the three parts of the return string, the translated string may contain leading or trailing blanks. You must use caution when parsing the return string with the REXX Parse statement in order to preserve the possible leading or trailing blanks.
2. The length value returned should be used as an indication of the actual length of the translated string. The length value includes any leading or trailing blanks.
3. ASCII/EBCDIC translation tables INWPETOA and INWPATOE are used for ASCII/EBCDIC translation. See VSE/ESA Programming and Workstation Guide for modifying a table.

Examples

Call:Socket('Translate','Hello ','To_Ascii')
Return Values: '0 6 xxxxx' (xxxxx is X'48656C6C6F20')

Call: Socket('Translate','48656C6C6F20'X,'To_Ebcdic')
Return Values: '0 6 Hello ' (Note the trailing blank.)

Call: Socket('Translate','128.228.1.2','To_IP_Address')
Return Values: '0 4 xxxx' (xxxx is X'80E40102')

Call: Socket('Translate','80E40102'X,'To_IP_Address')
Return Values: '0 11 128.228.1.2'

Call: Socket('Translate','64.64.64.64','To_IP_Address')
Return Values: '0 4 xxxx' (xxxx is X'40404040', four EBCDIC blanks)

Call:Socket('Translate',' ','To_IP_Address')
Return Values: '0 11 64.64.64.64'

Call: Socket('Translate','AF_INET 123 CUNYVM.CUNY.EDU','To_SockAddr_In')
Return Values: '0 16 xxxxxxxxxxxxxxxx' (xxxxxxxxxxxxxxxx is X'0002 007B 80E40102 0000000000000000')

Call: Socket('Translate','0002007B80E401020000000000000000'X,'To_SockAddr_In')
Return Values: '0 23 AF_INET 123 128.228.1.2'

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Version

Use the Version function to get the version number and date for the REXX Sockets function package.

Responses If successful, this function returns a string containing return code 0 and the REXX Sockets version and date. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

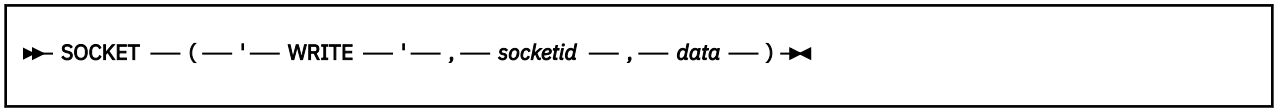
Examples

Call
Return Values

Socket('Version')
'0 REXX/SOCKETS *1.00 30 November 1999*'

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

Write



Use the Write function to write data on a connected socket. This function is similar to the Send function, except that it lacks the control flags available with Send. If it is not possible to write the data, Write waits until conditions are suitable for writing data. This blocks the caller, unless the socket is in nonblocking mode. For datagram sockets, the socket should not be in blocking mode.

Operands

socketid
is the identifier of the socket.

data
is the data to be written.

Responses If successful, this function returns a string containing return code 0 and the length of the data written. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

Call	Return Values
`Socket('Write',6,'some text')`	`'0 9'`

The C socket call is: `write(s, buf, len)`

Messages and Return Codes For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 307. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 307.

REXX Sockets System Messages

The following message indicates an error

```
ARX0960E ERROR Running Function SOCKET, RC=nnnnn
```

Code	Error Name
00008	CEEPIPI.PHASE not found (usually in PRD2.SCEEBASE, check your LIBDEF chain)
00009	CEEPIPI called from active LE-Environment(internal error, contact IBM)
00016	Storage problem (use a partition with more GETVIS space)
00020	Invocation of CEEPIPI-routine failed (internal error, contact IBM)
00024	Locking Problem (retry later on)
00028	REXX not initialized (invoke //EXEC ARXLINK)
00032	SECURE errorSecurity operands are specified with ACCEPT, CONNECT, or TAKESOCKET, but the preceding INITIALIZE has been invoked without security protocol.
00096	Internal Error (contact IBM)

REXX Sockets Return Codes

A REXX Sockets call returns a return code as the first token of the result string. If the return code is not zero, the second and third tokens in the result string are the error name and the corresponding error message. The following table lists the return code values defined for all REXX socket functions.

Code	Error Name	Error Message
2	ERANGE	Range error
111	EACCES	Permission denied
112	EAGAIN	Resource temporarily unavailable
113	EBADF	Bad file descriptor
114	EBUSY	Resource busy

Code	Error Name	Error Message
118	EFAULT	Bad address
119	EFBIG	File too large
120	EINTR	Interrupted function call
121	EINVAL	Invalid argument
122	EIO	Input/output error
124	EMFILE	Too many open files
127	ENFILE	Too many open files in system
129	ENOENT	No such file or directory
134	ENOSYS	Function not implemented
138	ENXIO	No such device or address
139	EPERM	Operation not permitted
140	EPIPE	Broken pipe
158	EMVSPARM	Bad parameters were passed to the service
1102	EWOULDBLOCK	Problem on non-blocking socket
1103	EINPROGRESS	Connection in progress
1104	EALREADY	Connection already in progress
1105	ENOTSTOCK	Descriptor does not refer to a socket
1106	EDESTADDRREQ	Destination address required
1107	EMSGSIZE	Message too long
1108	EPROTOTYPE	The socket type is not supported by the protocol
1109	ENOPROTOOPT	Protocol not available
1110	EPROTONOSUPPORT	Protocol not supported
1112	EOPNOTSUPPORT	Address family not supported
1115	EADDRINUSE	Address in use
1116	EADDRENOTAVAIL	Address not available
1118	ENETUNREACH	Network unreachable
1121	ECONNRESET	Connection reset
1122	ENOBUFS	No buffer space available
1123	EISCONN	Socket is already connected
1124	ENOTCONN	Socket not connected
35	ETIMEDOUT	Connection timed out
1128	ECONNREFUSED	Connection refused
2001	EINVALIDRXSOCKETCALL	Syntax error in RXSOCKET parameter list
2003	ESUBTASKINVALID	Subtask ID invalid
2004	ESUBTASKALREADYACTIVE	Subtsk already active

Code	Error Name	Error Message
2005	ESUBTASKNOTACTIVE	Subtask not active
2007	EMAXSOCKETSREACHED	Maximum number of sockets reached
2009	ESOCKETNOTDEFINED	Socket not defined
2016	EHOSTNOTFOUND	Host not found
2017	EIPADDRNOTFOUND	IP address not found
2018	ETRYAGAIN	Try again
2019	ENORECOVERY	No recovery
2020	ENODATA	No data
2021	ESOCKETNOTGIVEN	No Socket available to acquire

Sample Programs

This section describes two sample pairs of REXX socket programs:

1. Using a non-secured connection
   - REXX-EXEC RSCLIENT — a client sample program
   - REXX-EXEC RSSERVER — a server sample program
2. Using a secured connection via TCP/IP SSL support

Before you start the client program, you must start the server program in another address space. The two programs can run on different hosts, but the internet address of the host running the server program must be entered with the command starting the client program, and the hosts must be connected on the same network using TCP/IP.

REXX-EXEC RSCLIENT Sample Program

The client sample program (RSCLIENT EXEC) is a REXX socket program that shows you how to use the calls provided by REXX Sockets. The program connects to the server sample program and receives data, which is displayed on the screen. It uses sockets in blocking mode.

After parsing and testing the input parameters, RSCLIENT obtains a socket set using the Initialize function and a socket using the Socket function. The program then connects to the server and writes the user ID, the node ID, and the number of lines requested on the connection to the server. It reads data in a loop and displays it on the screen until the data length is zero, indicating that the server has closed the connection. If an error occurs, the client program displays the return code, determines the status of the socket set, and ends the socket set.

The server adds the EBCDIC new line character to the end of each record, and the client uses this character to determine the start of a new record. If the connection is abnormally closed, the client does not display partially received records.

```
trace o
signal on syntax

/* Set error code values                                          */
ecpref = 'RXS'
ecname = 'CLI'
initialized = 0

parse arg argstring
argstring = strip(argstring)
if substr(argstring,1,1) = '?' then do
  say 'RSSERVER and RSCLIENT  are a pair of programs which provide an'
  say 'example of how to use REXX/SOCKETS to implement a service. The'
  say 'server must be started before the clients get started.        '
```

```
  say '                                                              '
  say 'The RSSERVER program runs in its own dedicated partition      '
  say 'and returns a number of data lines as requested to the client.'
  say 'It is started with the JCL command:                           '
  say '    // EXEC REXX=RSSERVER                                     '
  say 'and terminated with the console command:                      '
  say '    MSG <pid>, DATA=H1                                        '
  say '                                                              '
  say 'The RSCLIENT program is used  to request a number of arbitrary'
  say 'data lines  from the  server  and can be run  concurrently any'
  say 'number  of times  by different  clients  until  the server  is'
  say 'terminated.  It is started with the command:                  '
  say '    // EXEC REXX=RSCLIENT,PARM="number <server>"              '
  say 'where "number" is the number of data lines to be requested and'
  say '"server" is the ipaddress of the service virtual machine. (The'
  say 'default ipaddress is the one of the host  on which RSCLIENT is'
  say 'running, assuming that RSSERVER runs on the same host.)       '
  exit 100
end

/* Split arguments into parameters and options                       */
parse upper var argstring parameters '(' options ')' rest

/* Parse the parameters                                              */
parse var parameters lines server rest
if ^datatype(lines,'W') then call error 'E', 24, 'Invalid number'
lines = lines + 0
if rest^='' then call error 'E', 24, 'Invalid parameters'

/* Parse the options                                                 */
do forever
  parse var options token options
  select
    when token='' then leave
    otherwise call error 'E', 20, 'Invalid option "'token'"'
  end
end

/* Initialize control information                                    */
port = '1952'               /* The port used by the server           */
userid = USERID()
call Sysvar('SYSPID')
locnode = SYSPID

/* Initialize                                                        */
call Socket 'Initialize', 'RSCLIENT'
if src=0 then initialized = 1
else call error 'E', 200, 'Unable to initialize RXSOCKET MODULE'
if server='' then do
  server = Socket('GetHostId')
  if src^=0 then call error 'E', 200, 'Cannot get the local ipaddress'
end
ipaddress = server

/* Initialize for receiving lines sent by the server                 */
s = Socket('Socket')
if src^=0 then call error 'E', 32, 'SOCKET(SOCKET) rc='src
call Socket 'Connect', s, 'AF_INET' port ipaddress
if src^=0 then call error 'E', 32, 'SOCKET(CONNECT) rc='src
call Socket 'Write', s, locnode userid lines
if src^=0 then call error 'E', 32, 'SOCKET(WRITE) rc='src

/* Wait for lines sent by the server                                 */
dataline = ''
num = 0
do forever

  /* Receive a line and display it                                   */
  parse value Socket('Read', s) with len newline
  if src^=0 | len<=0'' then leave
  dataline = dataline || newline
  do forever
    if pos('15'x,dataline)=0 then leave
    parse var dataline nextline '15'x dataline
    num = num + 1
    say right(num,5)':' nextline
  end
end

/* Terminate and exit                                                */
call Socket 'Terminate'
exit 0
```

```
/* Calling the real SOCKET function                                  */
socket: procedure expose initialized src
  a0 = arg(1)
  a1 = arg(2)
  a2 = arg(3)
  a3 = arg(4)
  a4 = arg(5)
  a5 = arg(6)
  a6 = arg(7)
  a7 = arg(8)
  a8 = arg(9)
  a9 = arg(10)
  parse value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with src res
return res

/* Syntax error routine                                              */
syntax:
  call error 'E', rc, '==> REXX Error No.' 20000+rc
return

/* Error message and exit routine                                    */
error: procedure expose ecpref ecname initialized
  type = arg(1)
  retc = arg(2)
  text = arg(3)
  ecretc = right(retc,3,'0')
  ectype = translate(type)
  ecfull = ecpref || ecname || ecretc || ectype
  say 'ecfull text'
  if type^='E' then return
  if initialized then do
    parse value Socket('SocketSetStatus') with . status severreason
    if status^='Connected' then do
      say 'The status of the socket set is' status severreason
    end
    call Socket 'Terminate'
  end
exit retc
```

REXX-EXEC RSSERVER Sample Program

The server sample program (RSSERVER EXEC) shows an example of how to use sockets in nonblocking mode. The program waits for connect requests from client programs, accepts the requests, and then sends data. The sample can handle multiple client requests in parallel processing.

The server program sets up a socket to accept connection requests from clients and waits in a loop for events reported by the select call. If a socket event occurs, it is processed. A read event can occur on the original socket for accepting connection requests and on sockets for accepted socket requests. A write event can occur only on sockets for accepted socket requests.

A read event on the original socket for connection requests means that a connection request from a client occurred. Read events on other sockets indicate either that there is data to receive or that the client has closed the socket. Write events indicate that the server can send more data. The server program sends only one line of data in response to a write event.

The server program keeps a list of sockets to which it wants to write. It keeps this list to avoid unwanted socket events. The TCP/IP protocol is not designed for one single-threaded program communicating on many different sockets, but for multithread applications where one thread processes only events from a single socket.

```
trace o
signal on syntax
signal on halt

/* Set error code values                                             */
initialized = 0

parse arg argstring
argstring = strip(argstring)
if substr(argstring,1,1) = '?' then do
  say 'RSSERVER and RSCLIENT  are a pair of programs which provide an'
  say 'example of how to use REXX/SOCKETS to implement a service. The'
  say 'server must be started before the clients get started.        '
```

```
  say '                                                              '
  say 'The RSSERVER program runs in its own partition.               '
  say 'It returns a number of data lines as requested to the client. '
  say 'It is started with the command:  // EXEC REXX=RSSERVER        '
  say 'and terminated by issuing "MSG <pid>,DATA=HI" at the console. '
  say '                                                              '
  say 'The RSCLIENT program is used  to request a number of arbitrary'
  say 'data lines  from the server.  One or more clients can access  '
  say 'the server until it is terminated.                            '
  say 'It is started with the command:                               '
  say '     // EXEC REXX=RSCLIENT,PARM="number <server>"             '
  say 'where "number" is the number of data lines to be requested and'
  say '"server" is the ipaddress of the service virtual machine. (The'
  say 'default ipaddress is the one of the host  on which RSCLIENT is'
  say 'running, assuming that RSSERVER runs on the same host.)       '
  say '                                                              '
  exit 100
end

/* Split arguments into parameters and options                       */
parse upper var argstring parameters '(' options ')' rest

/* Parse the parameters                                              */
parse var parameters rest
if rest^='' then call error 'E', 24, 'Invalid parameters specified'

/* Parse the options                                                 */
do forever
  parse var options token options
  select
    when token='' then leave
    otherwise call error 'E', 20, 'Invalid option "'token'"'
  end
end

/* Initialize control information                                    */
port = '1952'               /* The port used for the service         */

/* Initialize                                                        */
say 'RSSERVER: Initializing'
call Socket 'Initialize', 'RSSERVER'
if src=0 then initialized = 1
else call error 'E', 200, 'Unable to initialize SOCKET'
ipaddress = Socket('GetHostId')
if src^=0 then call error 'E', 200, 'Unable to get the local ipaddress'
say 'RSSERVER: Initialized: ipaddress='ipaddress 'port='port

/* Initialize for accepting connection requests                      */
s = Socket('Socket')
if src^=0 then call error 'E', 32, 'SOCKET(SOCKET) rc='src
call Socket 'Bind', s, 'AF_INET' port ipaddress
if src^=0 then call error 'E', 32, 'SOCKET(BIND) rc='src
call Socket 'Listen', s, 10
if src^=0 then call error 'E', 32, 'SOCKET(LISTEN) rc='src
call Socket 'Ioctl', s, 'FIONBIO', 'ON'
if src^=0 then call error 'E', 36, 'Cannot set mode of socket' s
/* Server can be stopped via "MSG <pid>,DATA=HI"                     */
/* call opermsg('ON')                                                */
/* Wait for new connections and send lines                           */
timeout    = 60
linecount. =  0
wlist = ''
do forever

  /* Wait for an event                                               */
  if wlist^='' then sockevtlist = 'Write'wlist 'Read * Exception'
  else sockevtlist = 'Write Read * Exception'
  sellist = Socket('Select',sockevtlist,timeout)
  if src^=0 then call error 'E', 36, 'SOCKET(SELECT) rc='src
  parse upper var sellist . 'READ' orlist 'WRITE' owlist 'EXCEPTION' .
  if orlist^='' | owlist^='' then do
     event = 'SOCKET'
     if orlist^='' then do
        parse var orlist orsocket .
        rest = 'READ' orsocket
     end
     else do
       parse var owlist owsocket .
       rest = 'WRITE' owsocket
     end
   end
   else event = 'TIME'
```

```
   select

    /* Accept connections from clients, receive and send messages     */
    when event='SOCKET' then do
      parse var rest keyword ts .

      /* Accept new connections from clients                          */
      if keyword='READ' . ts=s then do
        nsn = Socket('Accept',s)
        if src=0 then do
          parse var nsn ns . np nia .
          say 'RSSERVER: Connected by' nia 'on port' np 'and socket' ns
        end
      end

      /* Get nodeid, userid and number of lines to be sent            */
      if keyword='READ' . ts^=s then do
        parse value Socket('Recv',ts) with len nid uid count .
        if src=0 . len>0 . datatype(count,'W') then do
          if count<0 then count = 0
          if count>5000 then count = 5000
          ra = 'by' uid 'at' nid
          say 'RSSERVER: Request for' count 'lines on socket' ts ra
          linecount.ts = linecount.ts + count
          call addsock(ts)
        end
        else do
          call Socket 'Close',ts
          linecount.ts = 0
          call delsock(ts)
          say 'RSSERVER: Disconnected socket' ts
        end
      end

      /* Get nodeid, userid and number of lines to be sent            */
      if keyword='WRITE' then do
        if linecount.ts>0 then do
          num = random(1,sourceline())      /* Return random-selected */
          msg = sourceline(num) || '15'x    /*   line of this program */
          call Socket 'Send',ts,msg
          if src=0 then linecount.ts = linecount.ts - 1
          else linecount.ts = 0
        end
        else do
          call Socket 'Close',ts
          linecount.ts = 0
          call delsock(ts)
          say 'RSSERVER: Disconnected socket' ts
        end
      end

    end

    /* Unknown event (should not occur)                             */
    otherwise nop
  end
end

/* Terminate and exit                                               */
call Socket 'Terminate'
say 'RSSERVER: Terminated'
exit 0

/* Procedure to add a socket to the write socket list               */
addsock: procedure expose wlist
  s = arg(1)
  p = wordpos(s,wlist)
  if p=0 then wlist = wlist s
return

/* Procedure to del a socket from the write socket list             */
delsock: procedure expose wlist
  s = arg(1)
  p = wordpos(s,wlist)
  if p>0 then do
    templist = ''
    do i=1 to words(wlist)
      if i^=p then templist = templist word(wlist,i)
    end
    wlist = templist
  end
```

```
return

/* Calling the real SOCKET function                                  */
socket: procedure expose initialized src
  a0 = arg(1)
  a1 = arg(2)
  a2 = arg(3)
  a3 = arg(4)
  a4 = arg(5)
  a5 = arg(6)
  a6 = arg(7)
  a7 = arg(8)
  a8 = arg(9)
  a9 = arg(10)
  parse value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) with src res
return res


/* Syntax error routine                                              */
syntax:
  call error 'E', rc, '==> REXX Error No.' 20000+rc
return

/* Halt exit routine                                                 */
halt:
  call error 'E', 4, '==> REXX Interrupted'
return

/* Error message and exit routine                                    */
error:
  type = arg(1)
  retc = arg(2)
  text = arg(3)
  ecretc = right(retc,3,'0')
  ectype = translate(type)
  ecfull = 'RXSSRV' || ecretc || ectype
  say '===> Error:' ecfull text
  if type^='E' then return
  if initialized
     then do
       parse value Socket('SocketSetStatus') with . status severreason
       if status^='Connected'
          then say 'The status of the socket set is' status severreason
     end
  call Socket 'Terminate'
exit retc
```

Sample Programs Using the TCP/IP SSL Support with the REXX/VSE Socket Function

Server Program:

This procedure waits for a client to connect, receives a portion of data, reverses the data string and sends the reverse string back.

```
/* rexx procedure: socket server procedure */
rc = 0

/* initialize socketset */
fc = SOCKET('INITIALIZE','SERVMIRR',,,'SSLV3','CRYPTO.KEYRING',86400)
parse var fc socket_rc .
if socket_rc ¬= 0
then do
  say 'INITIALIZE failed with return info ' fc
  exit 99
end

/* create a TCP socket for client connection requests */
fc = SOCKET('SOCKET','AF_INET','SOCK_STREAM','IPPROTO_TCP')
parse var fc socket_rc newsocketid
if socket_rc ¬= 0
then do
  say 'SOCKET failed with return info ' fc
  fc = SOCKET('TERMINATE')
  exit 99
end
```

```
/* bind socket to well known port 5678 */
parse value Socket('GetHostId') with rc IpAddr
Host = "AF_INET 5678" IpAddr
fc = SOCKET('BIND',newsocketid,Host)
parse var fc bind_rc rest
if bind_rc ¬= 0
then do
  say 'BIND failed with return info ' fc
  fc = SOCKET('CLOSE',newsocketid)
  fc = SOCKET('TERMINATE')
  exit 99
end

/* create a connection queue for 1 client */
fc = SOCKET('LISTEN',newsocketid,'10')
parse var fc listen_rc rest
if listen_rc ¬= 0
then do
  say 'LISTEN failed with return info ' fc
  fc = SOCKET('CLOSE',newsocketid)
  fc = SOCKET('TERMINATE')
  exit 99
end

/* wait for a client to connect */
fc = SOCKET('ACCEPT',newsocketid,'SECURE','SAMPLE')
parse var fc accept_rc rest
if accept_rc ¬= 0
then do
  say 'ACCEPT failed with return info ' fc
  fc = SOCKET('CLOSE',newsocketid)
  fc = SOCKET('TERMINATE')
  exit 99
end
parse var rest accept_socket accept_socket_address
say "Client has established connection."

/* we don't want any more clients, close request socket */
fc = SOCKET('CLOSE',newsocketid)
parse var fc close_rc rest
if close_rc ¬= 0
then do
  say 'CLOSE failed with return info ' fc
  exit 99
end

/* read string from client, reverse it and send it back */
fc = SOCKET('READ',accept_socket,'10000')
parse var fc read_rc num_read_bytes read_string
if read_rc ¬= 0
then do
  say 'READ failed with return info ' fc
  rc = 99
  signal SHUTDOWN_LABEL
end
say "String read from client: '" read_string "'"
send_string = Reverse(read_string)
fc = SOCKET('SEND',accept_socket,send_string,'')
parse var fc send_rc num_sent_bytes
if send_rc ¬= 0
then do
  say 'SEND failed with return info ' fc
  rc = 99
  signal SHUTDOWN_LABEL
end
if num_read_bytes ¬= num_sent_bytes
then do
  say 'number of sent bytes does not match number of read bytes'
  rc = 99
  signal SHUTDOWN_LABEL
end

/* close client socket */
SHUTDOWN_LABEL:
fc = SOCKET('CLOSE',accept_socket)
parse var fc close_rc rest
if close_rc ¬= 0
then do
  say 'CLOSE failed with return info ' fc
  fc = SOCKET('TERMINATE')
  exit 99
```

```
end

fc = SOCKET('TERMINATE')
exit
rc
```

Client Program:

This procedure connects to a "mirror server", sends a string to this mirror server and receives the manipulated string again from the mirror server.

```
/* rexx procedure: socket client procedure */
rc = 0

/* ask user for string to send to the mirror server */
arg read_string

/* create a TCP socket */
fc = SOCKET('INITIALIZE','CLIEMIRR',,,'SSLV3','CRYPTO.KEYRING')
parse var fc socket_rc .
if socket_rc ¬= 0
then do
  say 'INITIALIZE failed with return info ' fc
  exit 99
end

/* create a TCP socket */
fc = SOCKET('SOCKET','AF_INET','STREAM','TCP')
parse var fc socket_rc newsocketid
if socket_rc ¬= 0
then do
  say 'SOCKET failed with return info ' fc
  fc = SOCKET('TERMINATE')
  exit 99
end

/* connect new socket to the specified server */
fc = SOCKET('CONNECT',newsocketid,'AF_INET 5678 9.164.155.71', ,
            'SECURE','SAMPLE')
parse var fc connect_rc rest
if connect_rc ¬= 0
then do
  say 'CONNECT failed with return info ' fc
  rc = 99
  signal SHUTDOWN_LABEL
end

/* send string to the mirror server */
fc = SOCKET('SEND',newsocketid,read_string,'')
parse var fc send_rc num_sent_bytes
if send_rc ¬= 0
then do
  say 'SEND failed with return info ' fc
  rc = 99
  signal SHUTDOWN_LABEL
end
if length(read_string) ¬= num_sent_bytes
then do
  say 'number of sent bytes does not match number of read bytes'
  rc = 99
  signal SHUTDOWN_LABEL
end

/* receive answer from mirror server */
fc = SOCKET('READ',newsocketid,'10000')
parse var fc read_rc num_read_bytes received_string
if read_rc ¬= 0
then do
  say 'READ failed with return info ' fc
  rc = 99
  signal SHUTDOWN_LABEL
end
say "String '" read_string "' was mirrored to: '" received_string "'"

SHUTDOWN_LABEL:
fc = SOCKET('CLOSE',newsocketid)
parse var fc close_rc rest
if close_rc ¬= 0
```

```
then do
  say 'CLOSE failed with return info ' fc
  fc = SOCKET('TERMINATE')
  exit 99
end

fc = SOCKET('TERMINATE')
exit
rc
```

Installation of REXX/VSE SOCKET Function

If you want to make use of the REXX/VSE Socket Function provided with REXX/VSE, you have to activate the REXX/VSE SOCKET Function Package ARXEFSO. This package is not contained in the default REXX environment initialization member ARXPARMS, but it can be easily made active using the customization job ARXPARMS.Z of library PRD1.BASE.

Here are the steps to be done:

1. Copy ARXPARMS.Z from PRD1.BASE into your primary ICCF library with

   ```
   LIBRP PRD1.BASE ARXPARMS.Z ARXPARMS [(REPLACE]
   ```

   You may also use DTRIINIT to move the member into the RDR queue and then move the RDR queue entry into your primary library. In this case the masked strings of POWER JECL and VSE JCL are automatically replaced and the next step can be omitted.
2. Edit member ARXPARMS in your primary library:

   Remove first line and the two lines at the end. Replace $$$$ by * $$,$$/* by /*, and $$/& by /&.
3. Choose the sublibrary for this new ARXPARMS.PHASE and insert it into the job. Default is PRD2.CONFIG, but you can use another sublibrary
4. Increase variable PACKTB_SYSTEM_USED by 1 as described in a comment of this member.
5. Run job ARXPARMS (option 7 within your primary library dialog).
6. To run a REXX SOCKET program make sure that the chosen sublibrary for ARXPARMS.PHASE precedes PRD1.BASE in the active PHASE chain. In this case system function package ARXEFSO, i.e. REXX/VSE function SOCKET, is available to your REXX programs.

For more information, see “Changing the Default Values for Initializing an Environment” on page 412.

Notes:

1. If more than one implementation of the REXX SOCKET function is installed on your system, the following search order applies:
   - search in function packages before search in sublibraries
   - search in user packages before search in local packages
   - search in local packages before search in system packages
   - If the same function is defined in more than one function package of the same level, the latest mentioned version in ARXPARMS is taken.

   Make sure that ARXPARMS and your LIBDEF phase chain are setup appropriately. Otherwise your desired SOCKET implementation is not executed.
2. To support coexistence and simultaneous usage of this SOCKET API with a different SOCKET function that might be established in your system, an extra function package *ARXEFSN* is provided. The only difference to ARXEFSO is the name of the function. *SOCKEN* is used instead of *SOCKET*. Thus, you can invoke within one REXX program the SOCKET functions described here via string *SOCKEN*, for example:

   ```
   ipaddress = Socken ('GETHOSTID')
   ```

   and another implementation of a SOCKET function available as an extra function package or an extra member SOCKET.PHASE via string *SOCKET*.

The TCP/IP PTF UQ38659 for instance provides an extra SOCKET.PHASE providing a proprietary Socket Applicaton Programming Interface for the VSE/ESA platform.

If you want to make use of name SOCKEN, change *ARXEFSO* into *ARXEFSN* within *ARXPARMS*.

Chapter 16. Debug Aids

In addition to the TRACE instruction, described on "TRACE" on page 53, there are the following debug aids:

- The interactive debug facility
- The immediate commands TS (Trace Start) and TE (Trace End).

  A REXX program must be running to use the immediate commands. HI halts interpretation of a REXX program. You specify it in a call to ARXIC from a non-REXX program. ARXIC is the trace and execution control routine.

  You can use TE and TS in a REXX program or call them through the ARXIC programming interface. (See "Immediate Commands" on page 143 for more information about the immediate commands.)
- The trace and execution control routine ARXIC. You can call ARXIC from a non-REXX program to use the following immediate commands:

  HI — Halt Interpretation
  TS — Trace Start
  TE — Trace End
  TQ — Trace Query
  HT — Halt Typing
  RT — Resume Typing.

  See "Trace and Execution Control Routine – ARXIC" on page 361 for more information.

Interactive Debugging of Programs

The debug facility permits interactively controlled execution of a REXX program. The operator's console can be used for input and output during interactive debug. Otherwise, the current input and output streams are used. ASSGN(STDIN) returns the name of the current input and ASSGN(STDOUT) returns the name of the current output. When interactive debug is first entered, a message indicating this is written. Changing the TRACE action to one with a prefix **?** (for example, TRACE ?A or the TRACE built-in function) turns on interactive debug and writes a message indicating it is on.

The language processor ignores further TRACE instructions in the program. If running from the operator's console, interactive debug pauses after nearly all instructions that are traced at the terminal—see the following for exceptions. If using the currently defined input and output streams, interactive debug reads the next line from the input stream at each pause point. Either way, the user can provide one of the following three inputs:

1. **A null line** (with no characters, including no blanks). This causes the language processor to continue execution until the next pause for debug input. Repeated input of a null line, therefore, steps from pause point to pause point. For TRACE ?A, for example, this is equivalent to single-stepping through the program.
2. **An equal sign (=)**, with no blanks. This causes the language processor to re-execute the clause last traced.

   Once 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 program). The same rules apply as in the INTERPRET instruction—for example, DO-END constructs must be complete. (The instruction you provide could be an assignment. For example, if an IF clause is about to take the wrong branch, you could change the value of the variable(s) on which it depends, and then re-execute it.) If an instruction has a syntax error in it, a standard message is produced. From the operator's console, you are prompted for input again; otherwise, the language processor reads the next line from the current

input. 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 sent to the output stream. Host Commands are always executed (that is, they are not affected by the prefix **!** on TRACE instructions), but the variable RC is not set.

Once the string has been processed, the language processor pauses again for further debug input, unless the last input was a TRACE instruction. 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 84). 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 includes a prefix, or if the input is TRACE 0 or TRACE with no options.

With the numeric form of the TRACE instruction (TRACE *n*) sections of the program run without pauses for debug input. If *n* is a positive number, interactive debug skips the next *n* pauses. If *n* is a negative number, this inhibits tracing for *n* clauses that would otherwise be traced.

The trace action specified on a TRACE instruction is saved and restored across subroutine calls. This means you can selectively trace the main routine or a subroutine. Suppose TRACE ?R (traces Results) is in effect and you enter a subroutine in which you have no interest. The input TRACE 0 would turn tracing off. No more instructions in the subroutine would be traced, but, on return to the main program, tracing would be restored.

If you are interested only in a subroutine, you can put TRACE ?R at its start. After tracing the subroutine, the language processor restores the original status of tracing. Therefore (if tracing was off on entry to the subroutine), tracing (and interactive debug) are off until the next entry to the subroutine.

You can switch tracing on or off asynchronously, (that is, while a program is running) using the TS and TE immediate commands. See "Interrupting Program Processing" on page 321 for the description of these facilities.

The ability to execute any instructions in interactive debug gives you considerable control over execution. Here are some examples of instructions you can enter in interactive debug.

```
Say expr     /* Produces the result of evaluating the         */
             /* expression.                                   */

name=expr    /* Changes 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 program.        */

exit         /* Ends execution of the program.                */

Do i=1 to 10; say stem.i; end  /* Produces 10 elements of     */
                               /* 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 tracing the target label)

- 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. (SIGNAL ON SYNTAX can trap these, but they cannot be re-executed.)

Interrupting Program Processing

HI (Halt Interpretation) interrupts the language processor during processing. To use HI, you include HI in a call to ARXIC from a non-REXX program. HI halts the interpretation of all REXX programs that are currently running as though a halt condition had been raised. This is especially useful when a program gets into a loop and you want to end processing.

When an HI interrupt halts the interpretation of a program, the data stack is cleared. You can trap an HI interrupt by enabling the halt condition using either the CALL ON or the SIGNAL ON instruction (see Chapter 7, “Conditions and Condition Traps,” on page 129).

The HI immediate command is processed as soon as control returns to the program, but before the next statement in the program is processed.

If the program is processing an external function or subroutine written in a programming language other than REXX or the program is processing a host command, when you halt program interpretation using HI, the halt is not processed until the function, subroutine, or command returns to the calling program. That is, the function, subroutine, or command completes processing before program processing is interrupted. HI cannot halt the program in all cases, such as the following:

- A program calls an external function or subroutine not written in REXX and the function or subroutine cannot return to the calling program (for example, it goes into a loop).
- Processing does not return to the program from a host command.

In these cases, HI cannot halt the program because it is not processed until the function, subroutine, or command returns to the program.

Starting and Stopping Tracing

The following describes how to start and stop tracing a program. You can start tracing REXX programs in several ways:

- You can use the TRACE instruction to start tracing. For more information, see “TRACE” on page 53.
- You can use the TS (Trace Start) immediate command in a REXX program to start tracing. TS puts the REXX program into interactive debug. You can then execute REXX instructions, for example, to SAY variables or to EXIT. Interactive debug is helpful if a program is looping. You can inspect the program and step through the execution before deciding whether or not to continue execution. The trace output is written to the current output stream. ASSGN(STDOUT) returns the name of the current output stream.

You can use TS in a REXX program or include it in a call to ARXIC from a non-REXX program.

You can end tracing in several ways:

- You can use the TRACE OFF instruction to end tracing. For more information, see “TRACE” on page 53.
- You can use TE to end tracing. Use TE in a REXX program or include it in a call to ARXIC from a non-REXX program.

See Chapter 10, “REXX/VSE Commands,” on page 143 for more information about the HI, TS, and TE immediate commands.

For more information about the trace and execution control routine ARXIC, see “Trace and Execution Control Routine – ARXIC” on page 361.

Chapter 17. Programming Services

Programming services for REXX processing let you interface with REXX and the language processor. Whenever you call a REXX/VSE routine, there are general conventions relating to registers that are passed on the call, parameter lists, and return codes the routines return. See "General Considerations for Calling REXX/VSE Routines" on page 324 for more information.

This topic summarizes the REXX programming services and then describes individual topics in detail.

Note: No applications or VSE services that call REXX should do so in an authorized state.

Calling REXX: You can call REXX directly through the JCL EXEC command. You specify REXX=*program_name* on the JCL EXEC statement. (See "Calling REXX Directly with the JCL EXEC Command" on page 329.)

ARXEXEC and ARXJCL are routines you can use to run a REXX program. They are programming interfaces to the language processor. (You can use ARXJCL to run a REXX program by specifying ARXJCL on the JCL EXEC statement; see "The ARXJCL Routine" on page 331 for details.) You can call ARXEXEC or ARXJCL to run a REXX program from a non-REXX program. (See "The ARXEXEC Routine" on page 334 or "The ARXJCL Routine" on page 331.)

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 programs. You can write an external function or subroutine in assembler or in REXX or another high-level programming language and store them in a sublibrary. You can also group frequently used external functions and subroutines into a *function package*. This provides quick access to the packaged functions and subroutines. When a REXX program calls an external function or subroutine, the function packages are searched before the active PROC or PHASE chain. (See "Search Order" on page 60 for a description of the search order.)

If you write external functions and subroutines, the language you use must support the system-dependent interfaces that the language processor uses to call the function or subroutine. To include an external function or subroutine in a function package, you must link-edit the function or subroutine into a phase. See "External Functions and Subroutines and Function Packages" on page 344 for a description of the system-dependent interfaces for writing external functions and subroutines and how to create function packages.

Variable Pool Access: The ARXEXCOM variable pool access interface lets commands and programs access and manipulate REXX variables. You can use ARXEXCOM to inspect, set, or drop variables. See "Variable Pool – ARXEXCOM" on page 352 for details about ARXEXCOM.

Maintain Host Command Environments: When a REXX program runs, there is at least one *host command environment* available for processing host commands. When a program begins running, there is an initial environment. You can change the host command environment with the ADDRESS instruction (see "ADDRESS" on page 27).

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 command processing. See "Commands to External Environments" on page 23 for information about host command environments.

A *host command environment table* defines:

- the valid host command environments
- the routines that are called to handle command processing within each environment
- the initial environment that is available to a REXX program when the program begins running.

You can customize REXX processing to define your own host command environment and provide a routine that handles command processing for that environment (see Chapter 18, "Customizing Services," on page 381).

The ARXSUBCM routine lets you access the entries in the host command environment table. You can use ARXSUBCM to add, change, or delete entries in the table and query the values for a particular host command environment entry. See "Maintain Entries in the Host Command Environment Table – ARXSUBCM" on page 357 for details about ARXSUBCM.

Trace and Execution Control: ARXIC is the trace and execution control routine. This lets you use the HI, HT, RT, TQ, TS, and TE commands to control the processing of REXX programs. For example, you can call ARXIC from a program written in assembler or a high-level language to control the tracing and execution of programs. See "Trace and Execution Control Routine – ARXIC" on page 361 for details about ARXIC.

Get Result Routine: ARXRLT is the *get result* routine. This lets you obtain the result from a REXX program that was called using ARXEXEC. You can also use ARXRLT if you write external functions and subroutines in a programming language other than REXX. ARXRLT lets your function or subroutine code get a large enough area of storage (EVALBLOK) to return the result to the calling program. ARXRLT also lets a compiler runtime processor obtain an evaluation block to handle the result from a compiled REXX program. See "Get Result Routine – ARXRLT" on page 363 for details about ARXRLT.

OUTTRAP Interface Routine: ARXOUT is the OUTTRAP interface routine. This lets programs write a character string to the REXX stem specified by the OUTTRAP external function. Only programs which have been invoked by the LINK or LINKPGM host command environment can use this interface. See "OUTTRAP Interface Routine – ARXOUT" on page 378 for details about ARXOUT.

SAY Instruction Routine: ARXSAY is the SAY instruction routine. ARXSAY lets you write a character string to the same output stream as the REXX SAY keyword instruction. See "SAY Instruction Routine – ARXSAY" on page 368 for details about ARXSAY.

Halt Condition Routine: ARXHLT is the halt condition routine. ARXHLT lets you query or reset the halt condition. See "Halt Condition Routine – ARXHLT" on page 370 for details about ARXHLT.

Text Retrieval Routine: ARXTXT is the text retrieval routine. ARXTXT lets you retrieve the same text that the language processor uses for the ERRORTEXT built-in function and for certain options of the DATE built-in function. For example, you can use ARXTXT in a program to retrieve the name of a month or the text of a syntax error message. See "Text Retrieval Routine – ARXTXT" on page 372 for details about ARXTXT.

LINESIZE Function Routine: ARXLIN is the LINESIZE function routine. ARXLIN lets you retrieve the same value that the LINESIZE built-in function returns. See "LINESIZE Function Routine – ARXLIN" on page 376 for details about ARXLIN.

General Considerations for Calling REXX/VSE Routines

Each description of a REXX/VSE routine explains how to use the routine, including entry and return specifications and parameter lists. The following topics provide general information about calling REXX/VSE routines.

All REXX/VSE routines, except for ARXINIT, the initialization routine, need a language processor environment. A language processor environment is the environment in which REXX operates, that is, in which the language processor processes a REXX program. REXX programs and routines run in a language processor environment.

REXX/VSE automatically initializes a language processor environment when one is needed. When you use the JCL EXEC command or call ARXEXEC or ARXJCL to run a program, REXX/VSE automatically initializes an environment if an environment does not already exist. The program then runs in that environment. The program can then call a REXX/VSE routine, such as ARXIC, and the routine runs in the same environment in which the program is running. See Chapter 19, "Language Processor Environments," on page 387 for details about environments, when they are initialized, and the different characteristics that make up an environment.

You can explicitly call the initialization routine, ARXINIT, to initialize language processor environments. Calling ARXINIT lets you *customize* the environment and how programs and services are processed and used. Using ARXINIT, you can create several different environments in a partition. See Chapter 18, "Customizing Services," on page 381 for details about customization.

If you explicitly call ARXINIT to initialize environments, whenever you call a REXX/VSE routine, you can specify the language processor environment in which you want the routine to run. During initialization, ARXINIT 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 ARXINIT and initialize several environments and then want to call a REXX/VSE 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 REXX/VSE 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 customizing services and the environment block, you can customize REXX processing and also control the environment in which you want REXX/VSE routines to run. For more information, see "Specifying the Address of the Environment Block" on page 326.

The following describes some general conventions for calling REXX/VSE routines:

- The REXX vector of external entry points is a control block that contains the addresses of the REXX/VSE routines and the REXX/VSE-supplied and user-supplied replaceable routines. The vector lets you easily access the address of a specific routine for calling the routine. See "Control Blocks Created for a Language Processor Environment" on page 414 for more information about the vector.
- **All calls must be in 31 bit addressing mode**.
- All data areas may be above 16 megabytes 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 REXX/VSE programming service called from the external function or subroutine. Passing the address of the environment block is particularly important if the environment is reentrant because programming services cannot automatically locate a reentrant environment. For more information on reentrant environments, see "Using the Environment Block for Reentrant Environments" on page 327.
- For most REXX/VSE 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 REXX/VSE Routines" on page 325.
- On calls to the REXX/VSE routines, you can pass the address of an environment block to specify the particular language processor environment in which you want the routine to run. For more information, see "Specifying the Address of the Environment Block" on page 326.
- Specific return codes are defined for each REXX/VSE routine. Some common return codes include 0, 20, 28, and 32. For more information, see "Return Codes for REXX/VSE Routines" on page 328.

Parameter Lists for REXX/VSE Routines

Most of the REXX/VSE routines have parameter lists. The parameters provide information to the routine about what type of processing you want to perform. They 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 18 on page 326 shows the format of the parameter lists for the REXX/VSE routines. A description of the parameter list follows the figure.

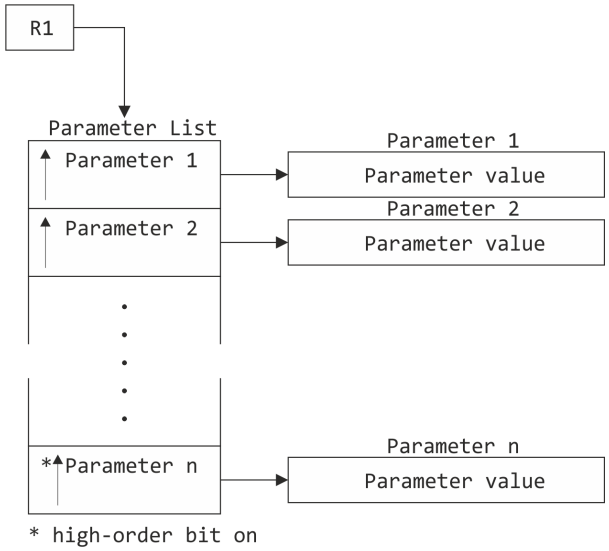


Figure 18. Overview of Parameter Lists for REXX/VSE Routines

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 18 on page 326. 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 18 on page 326. 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 may not be required. That is, some parameters may be optional. Because of this, the parameter lists are of *variable length*. Indicate the end of the parameter list by setting on the high-order bit 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 7 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, specify 0 or blanks, depending on whether the data is numeric or character data.

Suppose the routine has 7 parameters, parameters 6 and 7 are optional, and you do not want to use these optional parameters. You can end the parameter list at parameter 5 by setting on the high-order bit of the address for parameter 5.

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, ARXINIT, to initialize a language processor environment in a partition. If you explicitly call ARXINIT to initialize an environment, you can optionally specify this environment when you call any of the REXX/VSE 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.

For information about specifying the environment block address in the parameter list, see "Using the Environment Block Address Parameter" on page 327.

If you do not specify an address in the environment block address parameter, REXX/VSE 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 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 0, the routine searches for the last non-reentrant environment created, without checking whether register 0 contains a valid environment block address.

If you use ARXINIT to initialize reentrant environments, see "Using the Environment Block for Reentrant Environments" on page 327 for information about running in reentrant environments.

Using the Environment Block Address Parameter

The parameter lists of most of the REXX/VSE 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. Also, 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 incorrect 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 might specify the address of an existing environment that is not 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 a partition; environments 1, 2, 3, and 4. You want to call the trace and execution control routine, ARXIC, to halt the processing of programs in environment 2. However, when you call ARXIC, you specify the address of the environment block for environment 4, instead of environment 2. ARXIC completes successfully, but the processing of programs is halted in environment 4, rather than in environment 2. This is a subtle problem that may be difficult to identify. Therefore, if you use the environment block address parameter, 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. The parameter lists for the REXX/VSE 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 setting on the high-order bit in the last address in the parameter list. If you do not want to use the environment block address parameter or any parameters after it, you can end the parameter list at a preceding parameter. For more information about parameter lists, see "Parameter Lists for REXX/VSE Routines" on page 325.

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 specifying this parameter incorrectly.

Using the Environment Block for Reentrant Environments

If you want to use a reentrant environment, you must explicitly call the initialization routine, ARXINIT, to initialize the environment. REXX/VSE automatically initializes non-reentrant environments only. When you call ARXINIT to initialize a reentrant environment, you must set the RENTRANT flag on (see page "Flags and Corresponding Masks" on page 393).

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 that it does not want any other program to use. To ensure this, you would use ARXINIT 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.

REXX/VSE routines do not locate reentrant environments. Additionally, if you use ARXINIT to find an environment, ARXINIT 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 REXX/VSE programming routine. If you want to call a REXX/VSE 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 REXX/VSE Routines

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 obtain return code information more easily. 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 incorrect, the return code is returned in register 15 only.

Each REXX/VSE routine has specific return codes. The individual topics in this book describe the return codes for each routine. The common return codes that most of the REXX/VSE routines use are in Table 11 on page 328.

Table 11. Common Return Codes for REXX/VSE 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 current output that is defined for the environment. 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 “Flags and Corresponding Masks” on page 393. If multiple errors occurred and multiple error messages were issued, all error messages are written to the current output. 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.

Calling REXX

You can call REXX by using the JCL EXEC command or by calling ARXEXEC or ARXJCL. Calling REXX by using the JCL EXEC command lets you leave JCL statements on the stack. VSE/ESA can then process the JCL statements left on the stack. Thus, you can insert JCL statements or data into the current job stream.

JCL statements must be 80 characters. If a stack entry has fewer than 80 characters, it is padded with trailing blanks. If it has more than 80 characters, only the first 80 are used; the rest are ignored. After program processing is done, these 80-character entries are passed to VSE/ESA if the exit return code

is zero. VSE/ESA treats the statements that remain on the stack as a JCL procedure. See z/VSE System Control Statements for rules about the contents of a JCL procedure. See “Using the Data Stack” on page 422 for more information about the data stack.

ARXEXEC has more flexibility than ARXJCL. ARXEXEC permits you to pass more than one argument on the call and to preload a program in storage.

Note: To permit FORTRAN programs to call ARXEXEC, REXX/VSE provides an alternate entry point for the ARXEXEC routine. The alternate entry point name is ARXEX.

Calling REXX Directly with the JCL EXEC Command

You can use the JCL EXEC command to run a REXX program in batch. On the JCL EXEC statement, specify REXX=*program_name*; for example:

```
// EXEC REXX=MYPROG,SIZE=size
```

The *program_name* can be up to 8 characters. This is a member of a sublibrary in the active PROC chain. The SIZE parameter enables you to specify the size of the program area that may be used by REXX to load the user programs. As VSE JCL is already loaded at the beginning of the program area, 80 KB are added to the size specified in the SIZE parameter. See z/VSE System Control Statements for a full description of the SIZE parameter.

To include optional parameters, specify PARM=*parameters* in the format:

```
// EXEC REXX=program_name,PARM=parameters
```

You can specify a list of parameters in the PARM field of the EXEC statement.

Figure 19 on page 329 shows an example of JCL to run the program MYPROG.

```
*
// LIBDEF *,SEARCH=(PRD1.BASE,REXXLIB.SAMPLES)
// EXEC REXX=MYPROG,PARM='a b c d'
```

Figure 19. Example of Calling a REXX Program from a JCL EXEC Statement

If you omit the *program_name*, specify blanks, or specify a name of more than 8 characters, JCL reports an error and stops processing. REXX assumes the *program_name* is the name of a member of type PROC. REXX calls the Librarian services to search the active PROC chain for the PROC `myprog`. REXX accesses this program through the Librarian services. You can pass only one argument to the program being called, but the argument can consist of more than one token. In the example, the argument passed to the program is: a b c d.

The program being called needs to include a PARSE ARG keyword instruction such as PARSE ARG *exvars*. This instruction assigns a b c d (from the JCL EXEC statement) into the variable *exvars*.

The following example includes additional lines of SYSIPT data after the JCL EXEC statement.

```
// LIBDEF *,SEARCH=(PRD1.BASE,REXXLIB.SAMPLES)
// EXEC REXX=NEWPROG,PARM='a b c d'
input line 1
input line 2
/*
```

PULL (see “PULL” on page 48), PARSE EXTERNAL (see “PARSE” on page 44), or EXECIO (see “EXECIO” on page 145) can read the lines of input until encountering an end-of-file indicator, such as /*. When REXX/VSE does not read all input lines from SYSIPT, VSE JCL treats remaining SYSIPT data as JCL statements.

Note that reading inline SYSIPT data from nested JCL procedures is not possible.

If the REXX program runs successfully, the result that RETURN or EXIT returns is converted to binary and placed in the conditional JCL variable $RC. This permits conditional JCL to determine if the job should continue processing. If the result is greater than 4096, then the language processor applies modulo 4096 arithmetic to convert the result to a number in the range 0–4095.

If a REXX syntax error occurs, $RC contains 4095. In this case, the current output stream contains the REXX error code. See z/VSE Messages and Codes for details about REXX error messages.

Return Codes

REXX sets return codes when it detects an error in creating a language processor environment. When this occurs, JCL issues the message:

```
R002I REXX/VSE INITIALIZATION FAILED, RETURN CODE rr REASON CODE nn
```

The *rr* is the return code from the internal call to ARXINIT. If the return code is 20, the reason code *nn* is the ARXINIT reason code associated with the failure. If the return code is not 20, then the reason code is 0.

See Table 67 on page 435 for a complete list of reason codes and their meanings.

REXX sets return codes when it detects an error in the internal call to ARXEXEC. When this occurs, JCL issues the message:

```
R003I REXX/VSE EXEC PROCESSING FAILED, RETURN CODE rr
```

The *rr* is the return code from the internal call to ARXEXEC. If the return code is 40, an error occurred while processing the stack. An attempt to obtain SVA storage may have failed or the REXX/VSE stack service may have failed. See page Table 20 on page 343 for information about ARXEXEC return codes.

The ARXREXX Program

Job control calls the ARXREXX program when it detects the REXX= operand on the EXEC statement/command to invoke REXX/VSE. The list below describes the status of the registers for the ARXREXX program **on entry**.

Register
Contents

0
Address of an 8-byte field containing the name of the REXX program.

1
Address of the parameter field containing the arguments to be passed to the REXX program. The parameter field is a half-word field containing the length of the parameter data. The parameter data immediately follows the length. If there are no arguments to be passed to the program, register 1 is zero.

2-12
Reserved.

13
Address of an 18-word register save area.

14
Return address.

15
ARXREXX entry point address.

The list below describes the status of the registers at the time the ARXREXX program **returns control**.

Register
Contents

0
Reason code.

1
If the stack is empty, register 1 is zero.

Otherwise, register 1 contains an 8-byte field specifying the address and length of the stack storage (these are the same 8 bytes to which register 0 points on entry). The first word is the address of the stack storage. The second word specifies the length of storage. Each stack entry is 80 bytes long. A stack entry of less than 80 bytes is padded with trailing blanks. If a stack entry is longer than 80 bytes, only the first 80 bytes are used. The stack storage is located in the SVA.

Job control frees the stack storage after it completes processing of the stack entries.

2-14
Same as on entry.

15
Return code.

- User return codes are in the range of 0 to 4095. The use of the value of 4095 is not recommended since it is used by REXX (see below). If the user does not supply a value, zero is used. ARXREXX changes (modulo 4096) any user return code to bring it into the range of 0-4095.
- In case of a REXX syntax error, register 15 contains 4095.
- In case of an ARXINIT failure, register 15 contains 5000 plus the return code of ARXINIT. This indicates an initialization failure. Register 0 contains the reason code of the call to ARXINIT if the return code is 5020. Otherwise, it is zero.

  If the contents of register 15 is between 5000 and 5999, job control sets the return code of the last job step to 4095 and issues message R002 in addition.
- In case of an ARXEXEC failure, register 15 contains 6000 plus the return code of ARXEXEC. This indicates a failure when processing a REXX program. Register 0 has no meaning.

  If an error occurs during stack processing, register 15 contains 6040.

  If the contents of register 15 is between 6000 and 6999, job control sets the return code of the last job step to 4095 and issues message R003 in addition.

Note: Failures that occur during termination processing (ARXTERM) will not terminate the job stream and no error information is returned.

Calling REXX with ARXEXEC or ARXJCL

You can use ARXEXEC or ARXJCL to call REXX from a non-REXX program.

The ARXJCL Routine

ARXJCL is the simplest routine for calling REXX.

You can use ARXJCL to run a REXX program in two ways:

- Call ARXJCL from a non-REXX program
- Specify ARXJCL on the JCL EXEC statement.

To specify ARXJCL on the JCL EXEC statement, specify the name of the program and any arguments in the PARM field. For example, to run a REXX program named MYPROG and pass two arguments, you could use the following:

```
// LIBDEF *,SEARCH=(PRD1.BASE,REXXLIB.SAMPLES)
// EXEC ARXJCL,PARM='MYPROG arg1 arg2'
```

The remainder of this discussion about ARXJCL concerns calling ARXJCL from a non-REXX program.

On the call to ARXJCL, you pass the address of a parameter list in register 1.

Environment Customization Considerations

If you use the ARXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want ARXJCL to run. On the call to ARXJCL, 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 ARXJCL determines the address is not valid, ARXJCL locates the current environment and runs in that environment. "Chains of Environments and How Environments Are Located" on page 410 describes how environments are located. If a current environment does not exist, or the current environment was initialized on a different task, a new language processor environment is initialized. The program runs in the new environment. Before ARXJCL 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 326.

Entry Specifications: For the ARXJCL routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

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. To indicate the end of the parameter list, set the high-order bit of the last address in the parameter list to 1. Table 12 on page 332 describes the parameter for ARXJCL.

Table 12. Parameter for Calling the ARXJCL Routine

Parameter	Number of Bytes	Description
Parameter 1	variable	A buffer, which consists of a halfword length field followed by a data field. The length field contains the length of the data field that follows. (This length does not include the 2 bytes that specify the length itself.) The data field contains the name of the program, followed by one or more blanks, followed by the argument (if any) to be passed to the program. You can pass only one argument on the call, but the argument can consist of more than one token.

The following example shows an assembler program that calls ARXJCL to run a REXX program.

```
APISAMP  AMODE 31
APISAMP  RMODE ANY
APISAMP  CSECT
         STM   14,12,12(13)
         BALR  12,0
         USING *,12
```

```
         ST    13,SAVE+4
         LA    13,SAVE
         CDLOAD ARXJCL               Load ARXJCL into storage
         LR    15,1
         OI    PARM@,X'80'           Indicate the end of the Plist
         LA    1,PARM@               Load R1 with address of Plist
         BALR  14,15
         C     15,EXPECTED           Verify the program return code
         BNE   FAILURE               Handle failure
         WTO   'Exec called successfully',ROUTCDE=(2),DESC=(7)
         B     EXIT
FAILURE  WTO   'Exec return code incorrect',ROUTCDE=(2),DESC=(7)
*
EXIT     EQU   *
         L     13,SAVE+4
         LM    14,12,12(13)
         BR    14
*
EXPECTED DC    F'1'
SAVE     DS    18F
         DS    0F
PARM@    DC    A(*+4)
PARMLEN  DC    H'15'
PARMCMD  DC    CL15'APIEXEC 123 456'   Program called with 2 arguments
         LTORG
         END
```

Return Specifications: For the ARXJCL routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return Codes: If ARXJCL encounters an error, it returns a return code. If you call ARXJCL from a program, ARXJCL returns the return code in register 15. Table 13 on page 333 describes the return codes.

Table 13. Return Codes for ARXJCL Routine

Return Code	Description
0	Processing was successful. Program processing completed.
20	Processing was not successful. The program was not processed. One possible reason is a missing or failing REXX/VSE initialization. The execution of // EXEC ARXLINK has either been missing or failed.
20021	The JCL EXEC statement contained an incorrect parameter or the parameter list passed on the call to ARXJCL was incorrect. A parameter may have been blank or null or the name of the program may have been incorrect (longer than 8 characters).
Other	Any other return code is the return code from the REXX program on the RETURN or EXIT keyword instruction. **Note:** 1. No distinction is made between the REXX program returning a value of 0, 20, or 20021 on the RETURN or EXIT instruction and ARXJCL returning one of these return codes. 2. ARXJCL 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), VSE/ESA 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 program returns a return code of 8002, in decimal, on the RETURN or EXIT instruction. The value 8002 (decimal) is X'1F42' in hexadecimal. VSE/ESA 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 ARXEXEC Routine

You can use the ARXEXEC routine to call REXX from a non-REXX a program in any partition. ARXEXEC offers more flexibility:

- You can preload the REXX program in storage and pass the address of the preloaded program to ARXEXEC. This is useful if you want to run a program multiple times; the program is not loaded and freed each time you call it.
- With ARXEXEC, you can use your own load routine to load and free the program.
- The EXEC command and ARXJCL permit you to pass only one argument to the program (the argument can consist of several tokens). ARXEXEC lets you pass multiple arguments to the program, 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.)
- With ARXEXEC, one parameter on the call is the user field. You can use this field for your own processing.

Note: To permit FORTRAN programs to call ARXEXEC, REXX/VSE provides an alternate entry point for the ARXEXEC routine. The alternate entry point name is ARXEX.

If you use the EXEC command (page "EXEC" on page 145), you can pass only one argument to the program. The argument can consist of several tokens. Similarly, if you call ARXJCL, you can only pass one argument. Using ARXEXEC allows you to pass multiple arguments to the program, 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.

Environment Customization Considerations

If you use the ARXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want ARXEXEC to run. On the call to ARXEXEC, you can optionally specify the address of the environment block for the environment in either the parameter list or in register 0.

If you do not pass an environment block address or ARXEXEC determines the address is not valid, ARXEXEC locates the current environment and runs in that environment. "Chains of Environments and How Environments Are Located" on page 410 describes how environments are located. If a current environment does not exist, or the current environment was initialized on a different task, a new language processor environment is initialized. The program runs in the new environment. Before ARXEXEC 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 326.

Entry Specifications: For the ARXEXEC routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

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. To indicate the end of the parameter list, set

the high-order bit of the last address in the parameter list to 1. For general information about passing parameters, see “Parameter Lists for REXX/VSE Routines” on page 325.

Table 14 on page 335 describes the parameters for ARXEXEC.

Table 14. Parameters for ARXEXEC Routine

Parameter	Number of Bytes	Description
Parameter 1	4	Specifies the address of the exec block (EXECBLK). This is a control block that describes the program to load. It contains information needed to process the program, such as the member from which the program is to be loaded and the name of the initial host command environment when the program starts running. “The Exec Block (EXECBLK)” on page 337 describes the format of the exec block. If the program is preloaded and you pass the address of the preloaded program in parameter 4, specify an address of `0` for this parameter. If you specify both parameter 1 and parameter 4, ARXEXEC uses the value in parameter 4 and ignores parameter 1.
Parameter 2	4	Specifies the address of the first entry in a table that contains the arguments for the program. The arguments are arranged as a vector of address/length pairs followed by X'FFFFFFFFFFFFFFFF'. “Format of Argument List” on page 338 describes the format of the arguments.
Parameter 3	4	Flags describing the REXX program. ARXEXEC uses only bits 0, 1, 2, and 3. The remaining bits are reserved. Bits 0, 1, and 2 are mutually exclusive. PARSE SOURCE returns a token indicating how a program was called. The bit you set on in bit positions 0, 1, or 2 indicates the token (`COMMAND`, `FUNCTION`, or `SUBROUTINE`, respectively) that PARSE SOURCE uses. For example, if you set bit 2 on, PARSE SOURCE returns the token *SUBROUTINE*. The description of each bit follows: • Bit 0 - Set this bit on if the program is being called as a "command" (not from another program as an external function or subroutine). The program can optionally return a result. Do not set bit 0 on if you pass more than one argument to the program. • Bit 1 - Set this bit on if the program is being called as an external function (a function call). The program must return a result. • Bit 2 - Set this bit on if the program is being called as a subroutine, for example, using the CALL keyword instruction. The program can optionally return a result. • Bit 3 - Set this bit on if you want ARXEXEC to return *extended return codes* in the range 20001–20099. If a syntax error occurs, ARXEXEC returns a value in the range 20001–20099 in the evaluation block, regardless of the setting of bit 3. If a syntax error occurs and bit 3 is on, ARXEXEC returns with a return code in the range 20001–20099 that matches the value returned in the evaluation block. If a syntax error occurs and bit 3 is off, ARXEXEC returns with return code 0. For more information, see “How ARXEXEC Returns Information about Syntax Errors” on page 342.

Table 14. Parameters for ARXEXEC Routine (continued)

Parameter	Number of Bytes	Description
Parameter 4	4	Specifies the address of the *in-storage control block* (INSTBLK), which defines the structure of a preloaded program in storage. The INSTBLK contains pointers to each statement in the program and the length of each statement. "The In-Storage Control Block (INSTBLK)" on page 339 describes the control block. This parameter is required if the caller of ARXEXEC has preloaded the program. Otherwise, this parameter must be 0. If you specify this parameter, ARXEXEC ignores parameter 1 (address of the exec block).
Parameter 5	4	This parameter is reserved.
Parameter 6	4	Specifies the address of an evaluation block (EVALBLOCK). ARXEXEC uses the evaluation block to return the result from the program that was specified on either the RETURN or EXIT instruction. "The Evaluation Block (EVALBLOCK)" on page 341 describes the format of the evaluation block, how ARXEXEC uses the parameter, and whether or not 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 or if the evaluation block is too small, you can use the get result routine, ARXRLT, to obtain the result from the program.
Parameter 7	4	Specifies the address of an 8-byte field (the work-area header) that defines a work area for the ARXEXEC routine. In the 8-byte field, the: • The first 4 bytes contain the address of the work area • The last 4 bytes contain the length of the work area. The work area contains the storage for control blocks. The work area is passed to the language processor to use for processing the program. If the work area is too small, ARXEXEC returns with a return code of 20 and a message 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, ARXEXEC 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. ARXEXEC 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.

Table 14. Parameters for ARXEXEC Routine (continued)

Parameter	Number of Bytes	Description
Parameter 9	4	This parameter is optional. It is the address of the environment block to use when performing the requested service. If you specify a nonzero value for the environment block address parameter, ARXEXEC uses the value you specify and ignores register 0. However, ARXEXEC does not check whether the address is valid. Therefore, ensure that the address you specify is correct or unpredictable results can occur. For more information, see “Specifying the Address of the Environment Block” on page 326.
Parameter 10	4	This parameter is optional. ARXEXEC uses this field for the return code. If you use this parameter, ARXEXEC returns the return code in the parameter and also in register 15. Otherwise, ARXEXEC uses only register 15. If the parameter list is incorrect, the return code is returned in register 15 only. "Return Codes" describes the return codes.

The Exec Block (EXECBLK)

Parameter 1 specifies the address of the exec block (EXECBLK). This is a control block that describes the program to load. If the program is not preloaded, you must build the exec block and pass the address in parameter 1 on the call to ARXEXEC. You need not pass an exec block if the program is preloaded.

Note: If you want to preload the program, you can use the supplied exec load routine ARXLOAD or your own exec load replaceable routine (see “Exec Load Routine” on page 442).

A mapping macro for the exec block, ARXEXECB, is in PRD1.BASE. The following table shows the format of the exec block.

Note: In the following table, the field names ACRYN and LENGTH must include the prefix EXEC_BLK_. All other fields must include the prefix EXEC_.

Table 15. Format of the Exec Block (EXECBLK)

Offset (Decimal)	Number of Bytes	Field Name	Description
0	8	ACRYN	Identifies the exec block. This field must contain the character string ARXEXECB.
8	4	LENGTH	Specifies the length of the exec block in bytes.
12	4	—	Reserved.
16	8	MEMBER	Specifies the member name of the program if the program is in a sublibrary in the active PROC chain. A LIBDEF specifying the sublibrary must precede loading a member.
24	8	DDNAME	Reserved.

Table 15. Format of the Exec Block (EXECBLK) (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
32	8	SUBCOM	Specifies the name of the initial host command environment when the program starts running. If this field is blank, the environment specified in the INITIAL field of the host command environment table is used. The default is VSE. "Host Command Environment Table" on page 401 describes the table.
40	4	DSNPTR	Specifies the address of a sublibrary from which the member was loaded. The PARSE SOURCE instruction returns a string to which this address points. The name usually represents the name of the exec load member. The name can be up to 34 characters for the fully qualified sublibrary name (7 characters for the library name, 8 for the sublibrary name, 8 for the member name, and 8 for the type). If you do not want to specify a sublibrary name, specify an address of 0.
44	4	DSNLEN	Specifies the length of the sublibrary name to which the address at offset +40 points. The length can be 0–34. If no name is specified, the length is 0.

REXX programs are kept in PROC sublibraries. Programs must consist of fixed length, 80-byte records. (This is a Librarian restriction.) REXX programs are loaded from sublibraries. The interpreter uses Librarian services to locate, open, and read REXX programs.

A LIBDEF specifying the sublibrary must precede loading the member that the member at offset +16 specifies.

The fields at offset +40 and +44 in the exec block are only for input to the PARSE SOURCE instruction and are for informational purposes only.

If the program is preloaded, loading is not performed. Otherwise, the program is loaded using the member name in the active PROC chain.

Format of Argument List

Parameter 2 points to the arguments for the program. 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 16 on page 338 shows the format of the argument list. REXX/VSE provides a mapping macro ARXARGTB for the vector. The mapping macro is in PRD1.BASE.

Note: Each field name in the following table must include the prefix ARGTABLE_.

Table 16. 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

Table 16. Format of the Argument List (continued)

Offset (Dec)	Number of Bytes	Field Name	Description
12	4	ARGSTRING_LENGTH	Length of argument 2
16	4	ARGSTRING_PTR	Address of argument 3
20	4	ARGSTRING_LENGTH	Length of argument 3
		⋮	⋮
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 program in storage. The INSTBLK contains pointers to each record in the program and the length of each record.

If you preload the program 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 ARXEXEC returns. ARXEXEC simply reads information from the in-storage control block. It does not change any of the information.

To preload a program into storage, you can use the exec load replaceable routine ARXLOAD. Or you can provide your own routine to preload the program. “Exec Load Routine” on page 442 describes the replaceable routine.

If you are not preloading the program, 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 program, which are arranged as a vector of address/length pairs. Table 17 on page 339 shows the format of the in-storage control block header. Table 18 on page 340 shows the format of the vector of records. A mapping macro for the in-storage control block, ARXINSTB, is in PRD1.BASE.

Note: Each field name in the following table must include the prefix INSTBLK_.

Table 17. Format of the Header for the In-Storage Control Block

Offset (Decimal)	Number of Bytes	Field Name	Description
0	8	ACRONYM	Identifies the control block. This field must contain the characters `ARXINSTB`.
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 18 on page 340 for the format of the address/length pairs. If this field is `0`, the program contains no records.

Table 17. 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 program contains no records.
24	8	MEMBER	Specifies the name of the sublibrary from which the member was loaded. The PARSE SOURCE instruction returns the member name you specify. If this field is blank, PARSE SOURCE returns a question mark (?).
32	8	DDNAME	Reserved.
40	8	SUBCOM	Specifies the name of the initial host command environment when the program starts running.
48	4	---	Reserved.
52	4	DSNLEN	Specifies the length of the sublibrary name (from which the member was loaded) that is specified at offset +56. If a sublibrary name is not specified, this field must be 0.
56	72	DSNAME	This field contains the name of the sublibrary, if known, from which the program was loaded. The name can be up to 34 characters for the fully qualified sublibrary name (7 characters for the library name, 8 for the sublibrary name, 8 for the member name, and 8 for the type). The remaining bytes of the field (2 bytes plus 9 fullwords) are not used. They are reserved and contain binary zeros.

At offset +16 in the in-storage control block header, the field points to the vector of records that are in the program. The records are arranged as a vector of address/length pairs. Table 18 on page 340 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.

Note: Each field name in the following table must include the prefix INSTBLK_.

Table 18. 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
		⋮	
x	4	STMT@	Address of record *n*

Table 18. Vector of Records for the In-Storage Control Block (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
x+4	4	STMTLEN	Length of record *n*

The Evaluation Block (EVALBLOCK)

Parameter 6 specifies the address of an evaluation block (EVALBLOCK). This is a control block that ARXEXEC uses to return the result from the program. The program 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*. ARXEXEC returns the value of *var1* in the evaluation block.

If the program you are running will return a result, specify the address of an evaluation block when you call ARXEXEC (parameter 6). You must obtain the storage for the control block yourself.

If the program does not return a result or you want to ignore the result, you need not allocate an evaluation block. In this case, specify an address of 0 for the evaluation block.

If the result from the program fits into the evaluation block, the data is placed into the block (EVDATA field) and the length of the block is updated (EVLEN field). See "Using an Evaluation Block to Return a Result" for details about how to use ARXRLT to obtain the result if it does not fit in the area provided or if you did not allocate an evaluation block.

Note: The language processor environment is the environment in which the language processor processes the program. See Chapter 19, “Language Processor Environments,” on page 387 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 19 on page 341 shows the format of the evaluation block. Additional information about each field follows the table.

REXX/VSE provides a mapping macro ARXEVALB for the evaluation block. The mapping macro is in PRD1.BASE.

Note: Each field name in the following table must include the prefix EVALBLOCK_.

Table 19. 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.

Table 19. Format of the Evaluation Block (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
16	n	EVDATA	The field in which ARXEXEC returns the result from the program. 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 recommended that you use 250 bytes for the EVDATA field. For information about the values ARXEXEC returns if the language processor detects a syntax error in the program, see "How ARXEXEC Returns Information about Syntax Errors" on page 342.

If the result does not fit into the EVDATA field, ARXEXEC 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 ARXRLT routine to obtain the result. See "Get Result Routine – ARXRLT" on page 363 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 call the program as a "command" (bit 0 is set on in parameter 3), the result the program returns must be a numeric value. The result can be from -2,147,483,648 through +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 ARXEXEC Returns Information about Syntax Errors

If the language processor detects a syntax error in the program, ARXEXEC 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 between 1 and 99. Therefore, the range of values that ARXEXEC 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 ARX0026I. For error 26, ARXEXEC returns the value 20026 in the EVDATA field. The REXX error messages are described in *VSE/ESA Messages and Codes*.

The program may also return a value on the RETURN or EXIT instruction in the range 20001–20099. ARXEXEC returns the value from the program in the EVDATA field of the evaluation block. To determine whether the value in the EVDATA field is the value from the program 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 program processes successfully but the language processor detects a syntax error, the following occurs. ARXEXEC returns a return code of 0 in register 15. (ARXEXEC also places this return code in parameter 10 of the ARXEXEC routine.) ARXEXEC 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 program returned the 200*xx* value or the value represents a syntax error.

If you set bit 3 on and the program processes successfully but the language processor detects a syntax error, the following occurs. ARXEXEC 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. ARXEXEC also returns the 200*xx* value in the EVDATA field of the evaluation block. If you set bit 3 on and the program processes without a syntax error, ARXEXEC returns with a return code of 0 in register 15. If ARXEXEC returns a

value of 20001–20099 in the EVDATA field of the evaluation block, that value must be the value that the program 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 ARXEXEC to determine whether a syntax error occurred.

Return Specifications: For the ARXEXEC routine, the contents of the registers on return are:

Register 0
Address of the environment block

Registers 1-14
Same as on entry

Register 15
Return code

Return Codes: Table 20 on page 343 shows the return codes for the ARXEXEC routine. ARXEXEC returns the return code in register 15. If you specify the return code parameter (parameter 10), ARXEXEC also returns the return code in the parameter.

Table 20. ARXEXEC Return Codes

Return Code	Description
0	Processing was successful. The program has completed processing. If the program 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 ARXEXEC, you can set bit 3 in parameter 3 of the parameter list to indicate how ARXEXEC should handle information about syntax errors. If ARXEXEC returns with return code 0 and bit 3 is on, the language processor did not detect a syntax error. In this case, the value ARXEXEC returns in the EVDATA field of the evaluation block is the value the program returned. If ARXEXEC returns with return code 0 and bit 3 is off, the language processor may or may not have detected a syntax error. If ARXEXEC returns a value of 20001–20099 in the evaluation block, you cannot determine whether the value represents a syntax error or the value the program returned. For more information, see “How ARXEXEC Returns Information about Syntax Errors” on page 342.
20	Processing was not successful. An error occurred. The program has not been processed. REXX/VSE 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.
20001–20099	Processing was successful. The program completed processing, but the language processor detected a syntax error. The return code that ARXEXEC returns in register 15 is the value 20000 plus the REXX error number. See “How ARXEXEC Returns Information about Syntax Errors” on page 342.

Note: The language processor environment is the environment in which the program runs. If ARXEXEC cannot locate an environment in which to process the program, an environment is automatically initialized. If an environment was being initialized and an error occurred during the initialization process, ARXEXEC 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 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 programs.

You can write external functions or subroutines in REXX. You can store the program containing the function or subroutine in a sublibrary of member type PROC.

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 sublibrary of type PHASE. The language in which you write the program must support the system-dependent interfaces that the language processor uses to call 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 a *function package*. A function package is a group of external functions and subroutines that are *packaged* together. To include an external function or subroutine in a function package, you must link-edit the function or subroutine into a phase. You can link-edit only a *compiled* program into a phase. If you write a function or subroutine as a REXX program and the program is interpreted (that is, the interpreter executes the program), you cannot include the function or subroutine in a function package. However, if you write the function or subroutine in REXX and the REXX program is compiled, you can include the program in a function package because the compiled program can be link-edited into a phase. See “Compiler Publications” on page 500 for a list of books for the IBM Compiler and Library for REXX/370.

Interface for Writing External Function and Subroutine Code

You can use the same interface to call a subroutine or function. 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 REXX/VSE programming services called from the external function or subroutine. This is particularly important if the environment is reentrant because programming services cannot automatically locate a reentrant environment. For more information about reentrant environments, see “Using the Environment Block for Reentrant Environments” on page 327.

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 program that called 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 21 on page 345 describes the parameter list. A mapping macro for the external function parameter list, ARXEFPL, is in PRD1.BASE.

Table 21. External Function Parameter List

Offset (Decimal)	Number of Bytes	Field Name	Description
0	4	EFPLCOM	Reserved.
4	4	EFPLBARG	Reserved.
8	4	EFPLEARG	Reserved.
12	4	EFPLFB	Reserved.
16	4	EFPLARG	An address that points to the parsed argument list. Each argument is represented by an address/length pair. X'FFFFFFFFFFFFFFFF' ends the argument list. (See Table 16 on page 338 for the format of the argument list.) If the function or subroutine call includes no arguments, the address points to X'FFFFFFFFFFFFFFFF'.
20	4	EFPLEVAL	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 22 on page 346 describes the evaluation block.

Argument List: See Table 16 on page 338 for the format of the argument list the function or subroutine code receives at offset +16 (decimal) in the external function parameter list. A mapping macro for the argument list, ARXARGTB, is in PRD1.BASE.

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 22 on page 346 shows the format of the evaluation block.

A mapping macro for the evaluation block, ARXEVALB, is in PRD1.BASE.

Note:

1. The ARXEXEC routine also uses an evaluation block to return the result from a program that is specified on either the RETURN or EXIT instruction. The format of the evaluation block that ARXEXEC uses is identical to the format of the evaluation block passed to your function or subroutine code. “The Evaluation Block (EVALBLOCK)” on page 341 describes the control block for ARXEXEC.
2. Each field name in the following table must include the prefix EVALBLOCK_.

Table 22. 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.
8	4	EVLEN	On entry, this field is set to X'80000000'. This 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.
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). The EVDATA field of the evaluation block that REXX/VSE passes to your code is 250 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 ARXRLT (get result) routine to get a larger evaluation block. Call ARXRLT using the GETBLOCK function. ARXRLT 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 ARXRLT, see “Get Result Routine – ARXRLT” on page 363.

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 23 on page 347 shows the return codes.

Table 23. 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, syntax error 44 occurs. See z/VSE Messages and Codes for information about error numbers and their corresponding messages. 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.
Nonzero	Function or subroutine code processing was not successful. The language processor stops processing the REXX program that called your function or subroutine with an error code of 40, unless you trap the error with a SYNTAX trap. See z/VSE Messages and Codes for information about error numbers and their corresponding messages.

Function Packages

Function packages are groups of external functions and subroutines that are packaged together. When the language processor encounters a function call or call to a subroutine, the language processor searches the function packages before searching sublibraries. Grouping frequently used external functions and subroutines in a function package permits faster access to the function or subroutine. "Search Order" on page 60 describes the complete search order. There are three types of function packages:

- User packages are function packages that an individual user can write to replace or supplement certain supplied functions.
- Local packages are function packages that a system support or application group can write. Local packages may contain functions and subroutines that are available to a specific group of users or to the entire installation.
- System packages are function packages that an installation can write for system-wide use or for use in a particular language processor environment.

It is important to consider the search order when assigning a function to a particular type of package. The search order for the types of function packages is:

1. User packages
2. Local packages
3. System packages.

To provide function packages, there are several steps:

1. First write the individual external functions and subroutines you want to include in a function package. You can write an external function or subroutine in REXX or in any language that supports the interfaces the language processor uses to call the function or subroutine. "Interface for Writing External Function and Subroutine Code" on page 344 describes the interfaces. To add an external function or subroutine to a function package, you must link-edit the function or subroutine into a phase. You can link-edit only a compiled program into a phase. For information about compiled REXX programs, see "Compiler Publications" on page 500 for a list of books for the IBM Compiler and Library for REXX/370.
2. Write the directory for the function package. Each function package must contain a directory. The function package directory is contained in a phase. 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 348 describes the directory for function packages.

3. Specify the function package name (the name of the entry point at the beginning of the directory) in the function package table for a language processor environment. “Function Package Table” on page 403 describes the format of this table. There are several ways to do this, depending on the type of function package (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 ARXFLOC (local package) or ARXFUSER (user package). These two "dummy" directory names are in the default parameters module ARXPARMS. By naming your local function package directory ARXFLOC and your user function package directory ARXFUSER, the external functions and subroutines in the packages are automatically available to REXX programs.

   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 REXX/VSE uses to initialize a default language processor environment. “Specifying Directory Names in the Function Package Table” on page 352 describes how to define directory names in the function package table.

Note: If you explicitly call the ARXINIT routine, you can pass the address of a function package table containing your directory names on the call.

REXX/VSE provides the ARXEFVSE system function package. The function package provides the external functions ASSGN, REXXIPT, REXXMSG. SETLANG, SLEEP, STORAGE, SYSVAR, and OUTTRAP. (“External Functions” on page 92 describes these.) The default parameters module defines the ARXEFVSE function package. (See “Values in the ARXPARMS Default Parameters Module” on page 406.)

Other IBM products may also provide system function packages that you can use for REXX processing. If you install a product that provides a system function package for REXX/VSE, you must change the function package table and provide your own parameters module. 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 57 on page 404 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.
2. Provide your own ARXPARMS parameters module. The function package table is part of the parameters module that REXX/VSE uses to initialize language processor environments.

   Chapter 19, “Language Processor Environments,” on page 387 describes environments, their characteristics, and the format of the parameters module. In the same chapter, “Changing the Default Values for Initializing an Environment” on page 412 describes how to provide your own parameters module.

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.

The function package directory is contained in a phase. The function package directory name is the name of the entry point at the beginning of the directory. The name of the directory is specified only on the CSECT. The function package directory also 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 24 on page 349 shows the format of the directory header. Table 25 on page 349 illustrates the rows of entries in the function package directory. A mapping macro for the function package directory header and entries, ARXFPDIR, is in PRD1.BASE.

Note: Each field name in the following table must include the prefix FPCKDIR_.

Table 24. Format of the Function Package Directory Header

Offset (Decimal)	Number of Bytes	Description
0	8	A character field that must contain the character string ARXFPACK.
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	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.

As stated earlier, the function package table for the default parameters module ARXPARMS contains two "dummy" function package directory names: ARXFLOC for a local function package and ARXFUSER for a user function package.

If you create a local or user function package, you can name the directory ARXFLOC and ARXFUSER, respectively. By using ARXFLOC and ARXFUSER, 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 352 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 phase. You can link-edit the code and directory into separate phases or into the same phase. Place the sublibrary with the phases in the search sequence for a CDLOAD. The sublibrary must be in the active PHASE chain.

Note: For best performance, link-edit the code for individual functions or subroutines in the same phase 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 phase separate from the function package directory, that phase is loaded prior to each call of the function or subroutine and then deleted after that function or subroutine has completed.

Format of Entries in the Directory

Table 25 on page 349 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 25. Format of Entries in Function Package Directory

Offset (Decimal)	Number of Bytes	Field Name	Description
0	8	FUNCNAME	The name of the first function or subroutine (entry) in the directory.
8	4	FUNCADDR	The address of the entry point of the function or subroutine code (for the first entry).
12	4	---	Reserved.

Table 25. Format of Entries in Function Package Directory (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
16	8	SYSNAME	The name of the entry point in a phase that corresponds to the function or subroutine code (for the first entry).
24	8	SYSDD	Reserved.
32	8	FUNCNAME	The name of the second function or subroutine (entry) in the directory.
40	4	FUNCADDR	The address of the entry point of the function or subroutine code (for the second entry).
44	4	---	Reserved.
48	8	SYSNAME	The name of the entry point in a phase that corresponds to the function or subroutine code (for the second entry).
56	8	SYSDD	Reserved.

The following describes each entry (row) in the directory.

FUNCNAME
The 8-character name of the external function or subroutine. This is the name that is used in the REXX program. The name must be in uppercase, left justified, and padded to the right with blanks.

If this field is blank, the entry is ignored.

FUNCADDR
A 4-byte field containing 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`, REXX/VSE uses the SYSNAME. REXX/VSE issues a CDLOAD for the entry point SYSNAME specifies. If the address is present, REXX/VSE ignores the SYSNAME field.

SYSNAME
An 8-byte character name of the entry point in a phase that corresponds to the function or subroutine code to be called for the FUNCNAME. The name must be in uppercase, left justified, and padded to the right with blanks.

If the address is present, this field can be blank. If the address is `0` and this field is blank, REXX/VSE ignores the entry.

Example of a Function Package Directory

Figure 20 on page 351 shows an example of a function package directory. The example is explained after the figure.

```
ARXFUSER CSECT
         DC    CL8'ARXFPACK'       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 program
         DC    FL4'0'              Address of preloaded code
         DC    FL4'0'              Reserved field
         DC    CL8'ABCFUN1 '       Name of entry point
         DC    CL8'        '       Reserved
*                                  Start of definition of second entry
         DC    CL8'MYF2    '       Name used in program
         DC    FL4'0'              Address of preloaded code
         DC    FL4'0'              Reserved field
         DC    CL8'ABCFUN2 '       Name of entry point
         DC    CL8'        '       Reserved
*                                  Start of definition of third entry
         DC    CL8'MYS3    '       Name used in program
         DC    AL4(ABCSUB3)        Address of preloaded code
         DC    FL4'0'              Reserved field
         DC    CL8'ABCFUN3 '       Name of entry point
         DC    CL8'        '       Reserved
*                                  Start of definition of fourth entry
         DC    CL8'MYF4    '       Name used in program
         DC    VL4(ABCFUNC4)       Address of preloaded code
         DC    FL4'0'              Reserved field
         DC    CL8'ABCFUN4 '       Name of entry point
         DC    CL8'        '       Reserved
         SPACE 2
ABCSUB3  EQU  *
*  Subroutine code for subroutine MYS3
*
*  End of subroutine code
         END  ARXFUSER



            - - - - - New Object Module - - - - -


ABCFUNC4 CSECT
*  Function code for function MYF4
*
*  End of function code
         END  ABCFUNC4
```

Figure 20. Example of a Function Package Directory

In Figure 20 on page 351, the name of the function package directory is ARXFUSER, which is one of the "dummy" function package directory names in the default parameters module. This function package defines four entries:

- 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 a program calls the external function MYF1, REXX/VSE loads the phase with entry point ABCFUN1. If a program calls MYF2, REXX/VSE loads the phase with entry point ABCFUN2 from the active phase chain or the SVA because the ADDRESS is 0.

The phases 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 phase as the directory. The assembler, linkage editor, and loader have resolved the addresses.

If the name of the directory is not ARXFLOC or ARXFUSER, you must specify the directory name in the function package table for an environment. "Specifying Directory Names in the Function Package Table" on page 352 describes how you can do this.

When a language processor environment is initialized, either by default or when ARXINIT is explicitly called, the phases containing the function package directories for the environment are automatically loaded. External functions or subroutines that are link-edited as separate, stand-alone phases and are not defined in any function package are loaded prior to each invocation and then deleted after completion.

For best performance, link-edit the code for individual functions or subroutines in the same phase 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 programs running in a specific language processor environment. Each environment that is initialized has its own function package table. "Function Package Table" on page 403 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).

Variable Pool – ARXEXCOM

The language processor provides an interface that commands and programs can use to 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. The interface code checks names for validity and optionally does substitution into compound symbols according to REXX rules. The interface also makes available certain other information about the program that is running.

You can use the variable pool access interface ARXEXCOM to access and manipulate REXX program variables. ARXEXCOM can be used only if a REXX program has been *enabled for variable pool access* in the language processor environment. That is, a program must have been called, but is not currently being processed. For example, you can call a REXX program that calls a routine and the routine can then call ARXEXCOM. When the routine calls ARXEXCOM, the REXX program is *enabled for variable pool access*, but it is not being processed. If a routine calls ARXEXCOM and a program has not been enabled, ARXEXCOM returns with an error.

Note: To permit FORTRAN programs to call ARXEXCOM, there is an alternate entry point for the ARXEXCOM routine. The alternate entry point name is ARXEXC.

You can obtain the address of the ARXEXCOM routine from the REXX vector of external entry points. "Format of the REXX Vector of External Entry Points" on page 418 describes the vector. A program can also access ARXEXCOM by using a VSE CDLOAD macro to obtain the entry point address.

If a program uses ARXEXCOM, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the ARXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want ARXEXCOM to run. On the call to ARXEXCOM, 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 326.

Entry Specifications: For the ARXEXCOM routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

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. To indicate the end of the parameter list, set the high-order bit of the last address in the parameter list to 1. For more information about passing parameters, see “Parameter Lists for REXX/VSE Routines” on page 325.

Table 26 on page 353 describes the parameters for ARXEXCOM.

Table 26. Parameters for ARXEXCOM

Parameter	Number of Bytes	Description
Parameter 1	8	This field that must contain the character string 'ARXEXCOM'.
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 register 1 points to 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	Same as Parameter 2.
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 "SHVBLOCK".
Parameter 5	4	The address of the environment block of the environment in which you want ARXEXCOM to run. This parameter is optional. If you specify a nonzero value for the environment block address parameter, ARXEXCOM uses the value you specify and ignores register 0. However, ARXEXCOM 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 326.
Parameter 6	4	A field that ARXEXCOM uses to return the return code. This is optional. If you use this parameter, ARXEXCOM returns the return code in the parameter and also in register 15. Otherwise, ARXEXCOM uses register 15 only. If the parameter list is incorrect, the return code is returned in register 15 only. "Return Codes" describes the return codes.

The Shared Variable (Request) Block - SHVBLOCK: Parameter 4 is the address of 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 21 on page 354.

```
*******************************************************
* SHVBLOCK:  Layout of shared-variable PLIST element
*******************************************************
SHVBLOCK DSECT           SHARED VARIABLE REQUEST BLOCK
SHVNEXT  DS    A         Chain pointer to next SHVBLOCK
*                          (0 if last block)
SHVUSER  DS    F         Available for private use, except during
*                          "Fetch Next" when it identifies the
*                          length of the buffer SHVNAMA points to.
SHVCODES DS    0F
SHVCODE  DS    CL1       Individual function code indicating
*                          the type of variable pool access request
*                          (S,F,D,s,f,d,N, or P)
SHVRET   DS    XL1       Individual return code flags
         DS    H'0'      Reserved, should be 0
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 buffer
*                          (Set on fetch)
SHVBLEN  EQU *-SHVBLOCK  Length of SHVBLOCK
*                          (length of this block = 32)
         SPACE 1
*
*     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
SHVSYFET EQU   C'f'       Symbolic name Fetch variable
SHVSYSET EQU   C's'       Symbolic name Set variable
SHVSYDRO EQU   C'd'       Symbolic name Drop variable
SHVNEXTV EQU   C'N'       Fetch "next" variable
SHVPRIV  EQU   C'P'       Fetch private information
         SPACE 1
*
*     Return Code Flags (Stored in SHVRET):
*
SHVCLEAN EQU   X'00' Execution was successful
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' Variable name not valid
SHVBADV  EQU   X'10' Value not valid, may be too long
SHVBADF  EQU   X'80' Function code (SHVCODE) not valid
         SPACE 1
*
*     R15 return codes
*
         SPACE 1
SHVRCOK  EQU   0      Entire Plist chain processed
SHVRCINV EQU  -1      Entry conditions not valid
SHVRCIST EQU  -2      Insufficient storage available
         SPACE
         MEND
```

Figure 21. Request Block (SHVBLOCK)

Table 27 on page 355 describes the SHVBLOCK. A mapping macro for the SHVBLOCK, ARXSHVB, is in PRD1.BASE. The services you can perform using ARXEXCOM are specified in the SHVCODE field of each SHVBLOCK. "SHVCODE" describes the values you can use.

"Return Codes" describes the return codes from the ARXEXCOM routine.

Table 27. 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 pool access request. "SHVCODE" describes the valid codes.
9	1	SHVRET	Specifies the return code flag, whose values are shown in Figure 21 on page 354.
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 buffer. 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) can be in either lowercase or uppercase.

Lowercase
(The **Symbolic** interface). The names must be valid REXX symbols (in mixed case if desired). REXX substitution occurs 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.

Note: If you want generality, use the **Direct** interface rather than the **Symbolic** interface.

The other function codes, N and P, must always be 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 set. SHVVALA/SHVVALL describes the value to be assigned to it. The name is validated to ensure it contains only characters that can appear in names. The variable is then set. If the name is a stem, all variables with that stem are set, just as for 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 fetch. SHVVALA specifies the address of a buffer into which the data is copied. SHVBUFL contains the length of the buffer. The name is validated to ensure that it contains only characters that can appear in names. The variable is then located and copied to the buffer. The total length of the variable is put

into SHVVALL; 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 20.)

D and d
Drop variable. The SHVNAMA/SHVNAML address/length pair describes the name of the variable to drop. SHVVALA/SHVVALL are not used. The name is validated to ensure that it contains only characters that can appear in names. 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 searches through all the variables known to the language processor (that is, all those of the current generation except those "hidden" with PROCEDURE instructions). The order in which the variables are revealed is not specified.

The language processor maintains a pointer to its list of variables. This 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 ARXEXCOM interface.

Whenever an N (Next) function is processed, the name and value of the next variable available are copied to two buffers the caller supplies.

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; 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 program can 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. Only the first letter of each name is checked (though callers should supply the whole name). The following names are recognized:

ARG
Fetch primary argument string. Copies to the user's buffer area the first argument string that ARG would parse.

SOURCE
Fetch source string. Copies to the user's buffer the source string, as described for PARSE SOURCE on page “PARSE SOURCE ” on page 45.

VERSION
Fetch version string. Copies to the user's buffer the version string, as described for PARSE VERSION on page “PARSE VERSION ” on page 46.

Return Specifications: For the ARXEXCOM routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

The output from ARXEXCOM is stored in each SHVBLOCK.

Return Codes: Table 28 on page 357 shows the return codes for the ARXEXCOM routine. ARXEXCOM returns the return code in register 15. If you specify the return code parameter (parameter 6), ARXEXCOM also returns the return code in the parameter.

Figure 21 on page 354 shows the return code flags that are stored in the SHVRET field in the SHVBLOCK.

Table 28. Return Codes from ARXEXCOM (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; their SHVRET bytes are unchanged.
-1	Processing was not successful. The parameter list was incorrect or the environment was not valid. 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 program was not currently running • Another task is accessing the variable pool • A REXX program is currently running but is not enabled for variable pool 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 incorrect. 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 is a composite formed by the logical OR of SHVRETs, excluding SHVNEWV and SHVLVAR.

Maintain Entries in the Host Command Environment Table – ARXSUBCM

Use the ARXSUBCM routine to maintain entries in the host command environment table. The table contains the names of the valid host command environments that REXX programs can use to process host commands. In a program, 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 called to handle the processing of commands for each specific environment. "Host Command Environment Table" on page 401 describes the table in more detail.

Note: To permit FORTRAN programs to call ARXSUBCM, there is an alternate entry point for the ARXSUBCM routine. The alternate entry point name is ARXSUB.

Using ARXSUBCM, you can add, delete, update, or query entries in the table. (You can also use ARXSUBCM to dynamically update the host command environment table while a REXX program is running.)

You can obtain the address of the ARXSUBCM routine from the REXX vector of external entry points. "Format of the REXX Vector of External Entry Points" on page 418 describes the vector. A program can also access ARXSUBCM by using a VSE CDLOAD macro to obtain the entry point address.

If a program uses ARXSUBCM, it must create a parameter list and pass the address of the parameter list in register 1.

ARXSUBCM 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 REXX/VSE Routines" on page 324 for information). ARXSUBCM 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 ARXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want ARXSUBCM to run. On the call to ARXSUBCM, 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 326.

If the environment in which ARXSUBCM runs is part of a chain of environments and you use ARXSUBCM 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 ARXSUBCM updates the table.

Entry Specifications: For the ARXSUBCM routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

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. To indicate the end of the parameter list, set the high-order bit of the last address in the parameter list to 1. For more information about passing parameters, see "Parameter Lists for REXX/VSE Routines" on page 325.

Table 29 on page 359 describes the parameters for ARXSUBCM.

Table 29. Parameters for ARXSUBCM

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. If the function is ADD, UPDATE, or QUERY, then parameter 3, the string length, must be the length of a SUBCOMTB entry. If the function is DELETE, parameter 2, the string address, and parameter 3, the string length, must be 0. See "Functions" for descriptions of each function.
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" describes the entry in more detail.
Parameter 3	4	The length of the string (entry) to which parameter 2 points.
Parameter 4	8	The name of the subcommand. 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 for the environment in which you want ARXSUBCM to run. This parameter is optional. If you specify a nonzero value for the environment block address parameter, ARXSUBCM uses the value you specify and ignores register 0. However, ARXSUBCM 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 326.
Parameter 6	4	A 4-byte field that ARXSUBCM uses to return the return code. The return code parameter is optional. If you use this parameter, ARXSUBCM returns the return code in the parameter and also in register 15. Otherwise, ARXSUBCM uses register 15 only. If the parameter list is incorrect, the return code is returned in register 15 only. "Return Codes" describes the return codes.

Functions: Parameter 1 contains the name of the function ARXSUBCM is to perform. The functions are:

ADD
Adds an entry to the table using the values that the call specifies. ARXSUBCM does not check for duplicate entries. If you add a duplicate entry and then call ARXSUBCM to delete the entry, ARXSUBCM 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 the call specifies. ARXSUBCM does not change the entry name (the name of the host command environment).

QUERY
Returns the values associated with the last occurrence of the entry the call specifies.

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 30 on page 360 shows the format of an entry. A mapping macro for the table entries, ARXSUBCT, is in PRD1.BASE. "Host Command Environment Table" on page 401 describes the table in more detail.

Note: Each field name in the following table must include the prefix SUBCOMTB_.

Table 30. 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.
8	8	ROUTINE	The name of the *host command environment routine* that is called 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 456 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 called.

Return Specifications: The contents of the registers on return from ARXSUBCT are:

Registers 0-14
Same as on entry

Register 15
Return code

Return Codes: Table 31 on page 360 shows the return codes, which ARXSUBCM puts in register 15. If you specify parameter 6, the return code parameter, ARXSUBCM also puts the return code in this parameter.

Table 31. Return Codes for ARXSUBCM

Return Code	Description
0	Processing was successful.
4	Entry was not found (for FIND and DELETE functions only).
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.

Table 31. Return Codes for ARXSUBCM (continued)

Return Code	Description
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list was incorrect. The parameter list contains too few or too many parameters, or the high-order bit of the last address is not set to 1 to indicate the end of the parameter list.

Note: ARXSUBCM does not support the use of DBCS characters in host command environment names.

Trace and Execution Control Routine – ARXIC

Use the ARXIC routine to control the tracing and execution of REXX programs. A program can call ARXIC to use the following REXX immediate commands:

- HI (Halt Interpretation) — to halt the interpretation of REXX programs
- HT (Halt Typing) — to suppress output that REXX programs generate
- RT (Resume Typing) — to restore output you previously suppressed
- TQ (Trace Query) — to test if tracing of REXX programs is set on or off by TS or TE.
- TS (Trace Start) — to start tracing of REXX programs
- TE (Trace End) — to end tracing of REXX programs.

The immediate commands are described in Chapter 10, “REXX/VSE Commands,” on page 143.

You can obtain the address of the ARXIC routine from the REXX vector of external entry points. “Format of the REXX Vector of External Entry Points” on page 418 describes the vector. A program can also access ARXIC by using a VSE CDLOAD macro to obtain the entry point address.

If a program uses ARXIC, 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 ARXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want ARXIC to run. On the call to ARXIC, 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 326.

ARXIC affects only the language processor environment in which it runs.

Entry Specifications: For the ARXIC routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

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. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. For more information about passing parameters, see “Parameter Lists for REXX/VSE Routines” on page 325. Table 32 on page 362 describes the parameters for ARXIC.

Table 32. Parameters for ARXIC

Parameter	Number of Bytes	Description
Parameter 1	4	The address of the name of the command you want ARXIC to process. The valid command names are HI, HT, RT, TQ, TS, and TE. Descriptions of the command names follow the table.
Parameter 2	4	The length of the command name to which parameter 1 points.
Parameter 3	4	This parameter is optional. It is the address of the environment block to use when performing the requested service. If you specify a nonzero value for the environment block address parameter, ARXIC uses the value you specify and ignores register 0. However, ARXIC 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 326.
Parameter 4	4	This parameter is optional. It is a field that ARXIC uses to return the return code. If you use this parameter, ARXIC returns the return code in the parameter and also in register 15. Otherwise, ARXIC uses register 15 only. If the parameter list is incorrect, the return code is returned in register 15 only. "Return Codes" describes return codes.

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 processing of REXX programs. If HI has been issued, the language processor stops processing REXX programs. HI is reset if a halt condition is enabled or when no programs are running in the environment.

Note: The RXHLT exit can also raise the halt condition. See “REXX Exit Data Areas and Parameters” on page 473.

HT (Halt Typing)
This suppresses output from REXX programs (for example, the SAY instruction does not produce its output). HT does not affect output from any other part of REXX/VSE and does not affect error messages. HT is reset when the last program running in the environment ends.

RT (Resume Typing)
Resets the halt typing condition, restoring the production of output from REXX programs.

TQ (Trace Query)
Checks if tracing of REXX programs is set on or off.

TS (Trace Start)
Starts tracing of REXX programs.

TE (Trace End)
Ends tracing of REXX programs.

Return Specifications: For the ARXIC routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code.

Return Codes: Table 33 on page 363 shows the return codes for the ARXIC routine. ARXIC returns the return code in register 15. If you specify the return code parameter (parameter 4), ARXIC also returns the return code in the parameter.

Table 33. Return Codes for ARXIC

Return Code	Description
0	Processing was successful. For TQ, it indicates REXX trace was set OFF.
4	Processing was successful. REXX trace is set ON. This return code only applies for the TQ (Trace Query) command.
20	Processing was not successful. An error occurred. REXX/VSE issues a message that explains the error.
28	Processing was not successful. A language processor environment could not be found.
32	Processing was not successful. The parameter list is incorrect. The parameter list contains either too few or too many parameters, or the high-order bit of the last address is not set to 1 to indicate the end of the parameter list.

Get Result Routine – ARXRLT

Use the ARXRLT (get result) routine to obtain:

- The result from a program that was processed by calling the ARXEXEC routine.
- A larger evaluation block to return the result from an external function or subroutine that you have written in a programming language that supports the parameter interface.
- An evaluation block that a compiler runtime processor can use to handle the result from a compiled REXX program.

See page "Using an Evaluation Block to Return a Result" for details about obtaining the result or a larger evaluation block.

You can access ARXRLT through the REXX vector of external entry points. “Format of the REXX Vector of External Entry Points” on page 418 describes this vector. A program can also access ARXRLT by using a VSE CDLOAD macro to obtain the entry point address.

A compiler runtime processor can also use ARXRLT to obtain an evaluation block to handle the result from a compiled REXX program that is currently running. The evaluation block that ARXRLT returns has the same format as the evaluation block for ARXEXEC or for external functions or subroutines. For information about when a compiler runtime processor might require an evaluation block, see Chapter 24, “Support for the Library for REXX/370 in REXX/VSE,” on page 499. For information about the format of the evaluation block, see “The ARXEXEC Routine” on page 334 and "Evaluation Block".

Environment Customization Considerations

If you use the ARXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want ARXRLT to run. On the call to ARXRLT, 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 326.

Entry Specifications: For the ARXRLT routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return point 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. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. For more information about passing parameters, see “Parameter Lists for REXX/VSE Routines” on page 325.

Table 34 on page 364 describes the parameters for ARXRLT.

Table 34. Parameters for ARXRLT

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. "Functions" describes the valid functions.
Parameter 2	4	The address of the evaluation block. On input, this parameter is used only for the GETRLT and GETRLTE functions but not 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 program. On output, this parameter is used only for the GETBLOCK and GETEVAL 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 program 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 program that is currently running. This parameter is not used for the GETRLT and GETRLTE functions.

Table 34. Parameters for ARXRLT (continued)

Parameter	Number of Bytes	Description
Parameter 4	4	This parameter is optional. It is the address of the environment block to use when performing the requested service. If you specify a nonzero value for the environment block address parameter, ARXRLT uses the value you specify and ignores register 0. However, ARXRLT does not check whether the address is valid. Therefore, 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 326.
Parameter 5	4	This code parameter is optional. It is a field that ARXRLT uses to return the return code. If you use this parameter, ARXRLT returns the return code in the parameter and also in register 15. Otherwise, ARXRLT uses register 15 only. If the parameter list is incorrect, the return code is returned only in register 15. "Return Codes" describes the return codes.

Using an Evaluation Block to Return a Result: The REXX instructions RETURN and EXIT can return a result from a REXX program. When the program you are running does not return a result or you want to ignore the result, you need not allocate an evaluation block. If you want to return a result, when you call ARXEXEC you can allocate an evaluation block and pass its address to ARXEXEC for use in returning the result in the evaluation block. If the result from the program fits into the evaluation block, the data is placed in the EVDATA field of the block (padded with blanks on the right), and the length of the block (EVLEN field) is updated. Even if you did not specify an evaluation block, or if the evaluation block is too small to contain the result, the result is not lost. REXX/VSE stores the result in its own evaluation block. ARXEXEC also places as much of the result as fits into the EVDATA field of the evaluation block and returns (in EVLEN) the negative of the length needed. You can get the result as follows:

1. Obtain an evaluation block that is large enough to contain the result. Call ARXRLT (GETBLOCK function), specifying the length of the data area you require in parameter 3. ARXRLT returns the address of the new evaluation block in parameter 2. (It also frees the original evaluation block, which was too small to contain the result.)
2. Call ARXRLT (GETRLT or GETRLTE function) to return the result. (You pass the address of the new evaluation block in parameter 2.)

ARXRLT copies the result from the program that was stored in the REXX's evaluation block into your evaluation block and returns. (If the evaluation block you specify on a call to ARXRLT is too small, you can use the same technique to get the result.)

The result is available until one of the following occurs:

- You call ARXRLT to obtain the result and this is successful
- Another REXX program runs in the same language processor environment, or
- The language processor environment is terminated.

Note: The language processor environment is the environment in which REXX programs and routines run. See “General Considerations for Calling REXX/VSE Routines” on page 324 for information. Chapter 19, “Language Processor Environments,” on page 387 provides more details about environments and customization services. See "Evaluation Block" for more information about the format of the evaluation block.

Functions: Parameter 1 contains the name of the function ARXRLT is to perform:

GETBLOCK
Use the GETBLOCK function to obtain an evaluation block if you did not do so before or if a larger one is needed for the external function or subroutine that is running. The GETBLOCK function is valid only when a program is currently running.

You can write external functions and subroutines in REXX or in any programming language that VSE/ESA supports and that can follow the REXX conventions for passing parameters. If your external function or subroutine is not in 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.

GETRLT and GETRLTE
These functions obtain the result from the last REXX program that was processed in the language processor environment. If you use the ARXEXEC routine to run a program and then need to call ARXRLT to obtain the result from the program, call ARXRLT with the GETRLT or GETRLTE function. You can use GETRLT only if a program is not currently running in the language processor environment. You can use GETRLTE regardless of whether or not a program is currently running in the environment; this provides support for nested REXX programs.

For example, suppose you use the ARXEXEC routine to run a program and the result from the program does not fit into the evaluation block. After ARXEXEC returns control, you can call the ARXRLT routine with the GETRLT function to get the result from the program. At this point, the REXX program is no longer running in the environment.

For example, suppose you have a program that calls an external function that is written in assembler. The external function (assembler program) uses the ARXEXEC routine to call a REXX program. However, the result from the called program 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 ARXRLT with the GETRLTE function to obtain the result from the program. At this point, the original program that called the external function is still running in the language processor environment. GETRLTE obtains the result from the last program that completed in the environment, which, in this case, is the program the external function called.

For more information about running a program using the ARXEXEC routine and the evaluation block, see “The ARXEXEC Routine” on page 334.

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 program that is currently running. The GETEVAL function is supported only when a compiled program 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 ARXRLT get result routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code.

Return Codes: ARXRLT returns a return code in register 15. If you specify the return code parameter (parameter 5), ARXRLT also returns the return code in the parameter.

Table 35 on page 366 shows the return codes if you call ARXRLT with the GETBLOCK function or GETEVAL function.

Table 35. ARXRLT Return Codes for GETBLOCK or GETEVAL

Return Code	Description
0	Processing was successful. ARXRLT allocated a new evaluation block and returned the address of the evaluation block.

Table 35. ARXRLT Return Codes for GETBLOCK or GETEVAL (continued)

Return Code	Description
20	Processing was not successful. A new evaluation block was not allocated. This could be because: • The length you specified (in parameter 3) was incorrect. The length may have been negative or exceeded the maximum value of 16 megabytes minus the length of the evaluation block header. • REXX/VSE could not obtain the storage. • The function was requested at an incorrect time. (Perhaps a program was not running in the language processor environment.) • You specified the function name incorrectly in parameter 1. The evaluation block is set to all blanks and the length field is set to 0.
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 36 on page 367 shows the return codes if you call ARXRLT with the GETRLT or GETRLTE function.

Table 36. ARXRLT Return Codes for the GETRLT and GETRLTE Functions

Return Code	Description
0	Processing was successful. A return code of 0 indicates that ARXRLT completed successfully. You receive this return code when: • ARXRLT copied the entire result and set the length field to the length of the data. • The complete result was not returned. In this case, the evaluation block was too small. ARXRLT sets the length field to the negative of the length needed. • No result was available (ARXRLT could not find an evaluation block).
20	Processing was not successful. This could be because: • You specified parameter 2, the pointer to the evaluation block, as 0 • The evaluation block was incorrect (for example, the value in the EVLEN field was less than 0). • ARXRLT could not locate a valid REXX environment under the current task. (You may have called GETRLT or GETEVAL when a program was not running in the language processor environment. Or you may have called GETEVAL when a *compiled* program was not running in the language processor environment) • You specified the function name incorrectly in parameter 1.
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.

SAY Instruction Routine – ARXSAY

The SAY instruction routine, ARXSAY, lets you write a character string to the same output stream as the REXX keyword instruction SAY. For example, you can write a string to the default output stream SYSLST. “SAY” on page 50 describes the SAY keyword instruction.

You can obtain the address of the ARXSAY routine from the REXX vector of external entry points. “Format of the REXX Vector of External Entry Points” on page 418 describes the vector. A program can also access ARXSAY by using a VSE CDLOAD macro to obtain the entry point address.

If a program uses ARXSAY, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the ARXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want ARXSAY to run. On the call to ARXSAY, 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 326.

Entry Specifications: For the ARXSAY routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

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 can the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. For more information about passing parameters, see “Parameter Lists for REXX/VSE Routines” on page 325.

Table 37 on page 368 describes the parameters for ARXSAY.

Table 37. Parameters for ARXSAY

Parameter	Number of Bytes	Description
Parameter 1	8	The function to be performed. This must be in uppercase, left justified, and padded to the right with blanks. The valid functions are: • WRITE • WRITEERR. "Functions" describes the functions in more detail.

Table 37. Parameters for ARXSAY (continued)

Parameter	Number of Bytes	Description
Parameter 2	4	The address of a fullword in storage that points to an input buffer containing a string. The caller supplies the string, which is a string of bytes that you want ARXSAY to write to the output stream. There are no restrictions on the contents of the string. However, the target device for producing the data may limit the characters you can specify.
Parameter 3	4	The length, in bytes, of the string to which parameter 2 points.
Parameter 4	4	This parameter is optional. It is address of the environment block that represents the environment to use when performing the requested service. If you specify a nonzero value for the environment block address parameter, ARXSAY uses the value you specify and ignores register 0. However, ARXSAY does not check whether the address is valid. Therefore, 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 326.
Parameter 5	4	This parameter is optional. It is a field that ARXSAY uses to return the return code. If you use this parameter, ARXSAY returns the return code in the parameter and also in register 15. Otherwise, ARXSAY uses register 15 only. If the parameter list is incorrect, the return code is returned in register 15 only. "Return Codes" describes the return codes.

Functions: Parameter 1 contains the name of the function ARXSAY is to perform:

WRITE
Specifies that you want ARXSAY to write the input string you provide to the output stream. Output is directed to the current output stream. ASSGN(STDOUT) returns the name of the current output stream.

WRITEERR
Specifies that you want ARXSAY 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. “Flags and Corresponding Masks” on page 393 describes the flags.

Return Specifications: For the ARXSAY routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code.

Return Codes: Table 38 on page 370 shows the return codes for the ARXSAY routine. ARXSAY returns the return code in register 15. If you specify the return code parameter (parameter 5), ARXSAY also returns the return code in the parameter.

Table 38. Return Codes for ARXSAY

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. One of the parameters is incorrect. An error occurred and the requested function is not performed. REXX/VSE 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 format of the parameter list is incorrect. The parameter list has too few or too many parameters, or the high-order bit of the last address is not set to 1 to indicate the end of the parameter list.

Halt Condition Routine – ARXHLT

The halt condition routine, ARXHLT, lets you query or reset the halt condition. Using ARXHLT, you can determine whether a halt condition has been set, for example, with the HI immediate command. You can also reset the halt condition.

You can obtain the address of the ARXHLT routine from the REXX vector of external entry points. "Format of the REXX Vector of External Entry Points" on page 418 describes the vector. A program can also access ARXHLT by using a VSE CDLOAD macro to obtain the entry point address.

If a program uses ARXHLT, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the ARXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want ARXHLT to run. On the call to ARXHLT, 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 326.

Entry Specifications: For the ARXHLT routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

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 can pass the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for REXX/VSE Routines" on page 325.

Table 39 on page 371 describes the parameters for ARXHLT.

Table 39. Parameters for ARXHLT

Parameter	Number of Bytes	Description
Parameter 1	8	The function to be performed. This must be in uppercase, left justified, and padded to the right with blanks. Valid functions are: • TESTHLT • CLEARHLT. "Functions" describes the functions.
Parameter 2	4	This parameter is optional. It is the address of the environment block that represents the environment in which you want ARXHLT to run. If you specify an environment block address, ARXHLT uses the value you specify and ignores register 0. However, ARXHLT does not check whether the address is valid. Therefore, 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, ARXHLT checks whether the address is valid. For more information, see “Specifying the Address of the Environment Block” on page 326.
Parameter 3	4	This parameter is optional. It is a field that ARXHLT uses to return the return code. If you use this parameter, ARXHLT returns the return code in the parameter and also in register 15. Otherwise, ARXHLT uses only register 15. "Return Codes" describes the return codes.

Functions: Parameter 1 contains the name of the function ARXHLT is to perform:

TESTHLT
Determines whether the halt condition has been set. For example, the HI immediate command or ARXIC (the trace and execution control routine) can set the halt condition.

Return codes 0 and 4 from ARXHLT indicate whether or not the halt condition has been set. See "Return Codes" for more information.

CLEARHLT
Resets the halt condition.

Return Specifications: For the ARXHLT routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code.

Return Codes: Table 40 on page 372 shows the return codes for the ARXHLT routine. ARXHLT returns the return code in register 15. If you specify the return code parameter (parameter 3), ARXHLT also returns the return code in the parameter.

Table 40. Return Codes for ARXHLT

Return Code	Description
0	Processing was successful. For TESTHLT, this indicates the halt condition was tested and is not set. This means that REXX program processing continues. For CLEARHLT, it indicates successfully resetting the halt condition.
4	Processing was successful. A return code of 4 is used only for the TESTHLT function. It indicates the halt condition was tested and is set. This means that REXX processing will be halted, for example, just as if HI were processed.
20	Processing was not successful. One of the parameters was incorrect. An error occurred, and the requested function is not performed. ARXHLT returns a return code of 20 if the function name you specify in parameter 1 is incorrect.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The format of the parameter list is incorrect. It contains too few or too many parameters, or the high-order bit of the last address is not set to 1 to indicate the end of the parameter list. **Note:** The ARXHLT routine also calls the RXHLT exit, if one exists. See "REXX Exit Data Areas and Parameters" on page 473 for more information.

Text Retrieval Routine – ARXTXT

The text retrieval routine, ARXTXT, lets you retrieve data from the message repository. Besides error messages (ERRORTEXT built-in function output), this data includes information that the DATE built-in function could return. Using ARXTXT, 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 ARX0026I), the message text is:

```
Invalid whole number
```

You can obtain the address of the ARXTXT routine from the REXX vector of external entry points. "Format of the REXX Vector of External Entry Points" on page 418 describes the vector. A program can also access ARXTXT by using a VSE CDLOAD macro to obtain the entry point address.

If a program uses ARXTXT, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the ARXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want ARXTXT to run. On the call to ARXTXT, 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 326.

Entry Specifications: For the ARXTXT routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

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. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. For more information about passing parameters, see “Parameter Lists for REXX/VSE Routines” on page 325.

Table 41 on page 373 describes the parameters for ARXTXT.

Table 41. Parameters for ARXTXT

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. Valid functions are: • DAY • MTHLONG • MTHSHORT • SYNTXMSG. "Functions and Text Units" describes the functions.
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 ARXTXT to return. "Functions and Text Units" describes the text units in more detail.
Parameter 3	4	The address of an area in storage to hold the text that ARXTXT retrieves.
Parameter 4	4	The length of the area in storage to which parameter 3 points. You are recommended to provide a large buffer area to hold the result, for example, 250 bytes. If the buffer is too small to hold the returned text, ARXTXT returns with return code 20. On output, ARXTXT updates parameter 4 to contain the length of the actual text it returns.

Table 41. Parameters for ARXTXT (continued)

Parameter	Number of Bytes	Description
Parameter 5	4	This parameter is optional. It is the address of the environment block that represents the environment in which you want ARXTXT to run. If you specify a nonzero value for the environment block address parameter, ARXTXT uses the value you specify and ignores register 0. However, ARXTXT does not check whether the address is valid. Therefore, 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 326.
Parameter 6	4	This parameter is optional. It is a field that ARXTXT uses to return the return code. If you use this parameter, ARXTXT returns the return code in the parameter and also in register 15. Otherwise, ARXTXT uses only register 15. If the parameter list is incorrect, the return code is returned only in register 15. "Return Codes" describes the return codes.

Functions and Text Units: Parameter 1 contains the name of the function ARXTXT is to perform. Parameter 2 specifies the text unit you want ARXTXT to retrieve for the particular function. The functions and their corresponding text units you can request are:

DAY
returns the English name of a day of the week in mixed case. The names that ARXTXT retrieves are the same values the language processor uses for the DATE(Weekday) function.

The name of the day that ARXTXT retrieves depends on the text unit you specify in parameter 2. Table 42 on page 374 shows the text units for parameter 2 and the corresponding day ARXTXT retrieves for each text unit. For example, if you want ARXTXT to return the value `Saturday`, you specify text unit 3.

Table 42. 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
returns the English name of a month, in mixed case. The names that ARXTXT retrieves are the same values the language processor uses for the DATE(Month) function.

The name of the month that ARXTXT retrieves depends on the text unit you specify in parameter 2. Table 43 on page 375 shows the text units for parameter 2 and the corresponding name of the month ARXTXT retrieves for each text unit. For example, if you want ARXTXT to return the value `April`, you specify text unit 4.

Table 43. 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
11	November
12	December

MTHSHORT
returns the first three characters of the English name of a month in mixed case. ARXTXT retrieves the same values that the language processor uses for the month in the DATE(Normal) function.

The abbreviated name of the month that ARXTXT retrieves depends on the text unit you specify in parameter 2. Table 44 on page 375 shows the text units for parameter 2 and the corresponding abbreviated names of the month that ARXTXT retrieves for each text unit. For example, if you want ARXTXT to return the value Sep, you specify text unit 9.

Table 44. 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. ARXTXT retrieves the same text that the ERRORTEXT function returns.

The message text that ARXTXT 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 ARX0026I. The message text for ARX0026I is:

```
Invalid whole number
```

The SYNTXMSG function returns this value if you specify text unit 26.

REXX reserves the values 1–99 for error numbers. However, REXX does not use all these values for syntax error messages. See z/VSE Messages and Codes for REXX error numbers and messages. If you specify a text unit in the range 1-99 and the value is not supported, ARXTXT returns a string of length 0.

Return Specifications: For the ARXTXT 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 376 shows the return codes for the ARXTXT routine. ARXTXT returns the return code in register 15. If you specify the return code parameter (parameter 6), ARXTXT also returns the return code in the parameter.

Table 45. Return Codes for ARXTXT

Return Code	Description
0	Processing was successful. ARXTXT 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. ARXTXT 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 name you specified for parameter 1 is incorrect • Text unit you specified for parameter 2 is incorrect 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 incorrect. It contains too few or too many parameters, or the high-order bit of the last address is not 1 to indicate the end of the parameter list.

LINESIZE Function Routine – ARXLIN

The LINESIZE function routine, ARXLIN, lets you obtain the same value that the LINESIZE built-in function returns. "LINESIZE" on page 92 describes the built-in function.

You can obtain the address of the ARXLIN routine from the REXX vector of external entry points. "Format of the REXX Vector of External Entry Points" on page 418 describes the vector. A program can also access ARXLIN by using a VSE CDLOAD macro to obtain the entry point address.

If a program uses ARXLIN, it must create a parameter list and pass the address of the parameter list in register 1.

Environment Customization Considerations

If you use the ARXINIT initialization routine to initialize language processor environments, you can specify the environment in which you want ARXLIN to run. On the call to ARXLIN, 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 326.

Entry Specifications: For the ARXLIN routine, the contents of the registers on entry are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

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. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for REXX/VSE Routines" on page 325.

Table 46 on page 377 describes the parameters for ARXLIN.

Table 46. Parameters for ARXLIN

Parameter	Number of Bytes	Description
Parameter 1	8	The name of the function to be performed. This must be in uppercase, left justified, and padded to the right with blanks. The only valid function name is LINESIZE. ARXLIN returns the same value that the LINESIZE built-in function returns.
Parameter 2	4	ARXLIN returns the LINESIZE value in this parameter. ARXLIN returns the same value that the LINESIZE built-in function returns. "LINESIZE" on page 92 describes the built-in function. The value ARXLIN returns in this parameter is valid only if the return code is 0.
Parameter 3	4	This parameter is optional. It is the address of the environment block that represents the environment in which you want ARXLIN to run. If you specify an environment block address, ARXLIN uses the value you specify and ignores register 0. However, ARXLIN does not check whether the address is valid. Therefore, 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, ARXLIN checks whether the address is valid. For more information, see "Specifying the Address of the Environment Block" on page 326.

Table 46. Parameters for ARXLIN (continued)

Parameter	Number of Bytes	Description
Parameter 4	4	This parameter is optional. It is a field that ARXLIN uses to return the return code. If you use this parameter, ARXLIN returns the return code in the parameter and also in register 15. Otherwise, ARXLIN uses only register 15. "Return Codes" describes the return codes.

Return Specifications: For the ARXLIN routine, the contents of the registers on return are:

Registers 0-14
Same as on entry

Register 15
Return code

Return Codes: Table 47 on page 378 shows the return codes for the ARXLIN routine. ARXLIN returns the return code in register 15. If you specify the return code parameter (parameter 4), ARXLIN also returns the return code in the parameter.

Table 47. Return Codes for ARXLIN

Return Code	Description
0	Processing was successful. ARXLIN returned the LINESIZE value in parameter 2.
20	Processing was not successful. You may have specified an incorrect function name in 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 incorrect. It contains too few or too many parameters, or the high-order bit of the last address is not 1 to indicate the end of the parameter list.

OUTTRAP Interface Routine – ARXOUT

Use the OUTTRAP interface routine, ARXOUT, to allow programs to write a character string to the REXX stem specified by the OUTTRAP external function. Only programs can use this interface which have been invoked by the LINK or LINKPGM host command environment. ARXOUT writes into the OUTTRAP stem specified by the REXX program which calls one of these two ADRESS LINK environments.

Environment Customization Considerations

On the call to the OUTTRAP interface routine you pass the address of the parameter list in register 1. On the call to ARXOUT you can optionally specify the address of the environment block in either the parameter list or in register 0. If you specify a nonzero value as environment block in the parameter list, ARXOUT uses the value and ignores register 0. However, ARXOUT does not check whether the address is valid. If you do not specify an environment block or the specified value is not valid ARXOUT locates the current environment and runs in that environment. If a current environment does not exist, or the current environment was initialized on a different task, ARXOUT returns with return code 28.

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 326.

Entry Specifications:

ARXOUT has RMODE 24 and AMODE ANY.

ARXOUT returns in the mode of the calling program which can have any AMODE.

For the ARXOUT routine, the contents of the registers on entry are:

Register 0
unpredictable

Register 1
Address of the parameter list the caller passes

Registers 2-12
Unpredictable

Register 13
Address of the save area

Register 14
Return address

Register 15
Entry point address.

Parameters: In register 1, you can 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 two parameters are mandatory. Parameters 3 and 4 are optional. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for REXX/VSE Routines" on page 325.

Table 48 on page 379 describes the parameters for ARXOUT.

Table 48. Parameters for ARXOUT

Parameter	Number of Bytes	Description
Parameter 1		Specifies the address of a fullword in storage that points to an input buffer containing the character string. The caller supplies the string you want ARXOUT to write into OUTTRAP.
Parameter 2		Specifies the length in bytes of the string parameter 1 points to.
Parameter 3		This parameter is optional. It is the address of the environment ARXOUT uses. If you specify a zero value for parameter 3, ARXOUT uses the environment block address specified in register 0.
Parameter 4		This parameter is optional. It is the field ARXOUT uses to return the return code. If you use this parameter, ARXOUT returns the return code in this parameter and register 15, else only in register 15.

If the caller is in AMODE 24, all specified addresses are interpreted as 24 bit addresses.

If the caller is in AMODE 31, all specified addresses are interpreted as 31 bit addresses.

Return Codes: Table 49 on page 379 shows the return codes for the ARXOUT routine. ARXOUT returns the return code in register 15. If you specify the return code parameter (parameter 4), ARXOUT also returns the return code in the parameter.

Table 49. Return Codes for ARXOUT

Return Code	Description
0	Processing was successful. The input string was accepted and written to OUTTRAP if the maximum number of output records are not exceeded. Input records are ignored with a return code of 0 if • OUTTRAP is full (maximum number of output records are exceeded). • OUTTRAP is OFF.

Table 49. Return Codes for ARXOUT (continued)

Return Code	Description
8	Processing was not successful. GETVIS cannot be obtained.
20	Processing was not successful. An error occurred, and the requested function is not performed.
24	Processing was not successful. The caller was not authorized to use ARXOUT, it was either not called by ADDRESS LINK or LINKPGM or the REXX tables were not initialized.
28	Processing was not successful. The current environment does not exist, or the current environment was initalized with a different task.
32	Processing was not successful. Invalid parameter list.

Chapter 18. Customizing Services

REXX/VSE provides customizing services for REXX processing to let you change how REXX programs are processed and how the language processor accesses and uses system services.

Customizing services include the following:

Environment Characteristics
These routines and services let you customize the environment in which the language processor runs a REXX program. This environment is known as the *language processor environment*. It defines various characteristics relating to how programs are processed and how the language processor accesses and uses system services. REXX/VSE 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 program runs, the language processor uses various system services, such as services for loading and freeing a program, performing I/O, obtaining and freeing storage, and handling data stack requests. Routines that handle these types of system services are known as *replaceable routines* because you can provide your own routine to replace the one REXX/VSE provides.

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 19, “Language Processor Environments,” on page 387 describes how you can customize the environment in which the language processor executes a REXX program and accesses and uses system services.
- Chapter 20, “Initialization and Termination Routines,” on page 427 describes the ARXINIT and ARXTERM routines for initializing and terminating language processor environments.
- Chapter 21, “Replaceable Routines and Exits,” on page 439 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 Program Processing

Figure 22 on page 382 shows the processing of a REXX program.

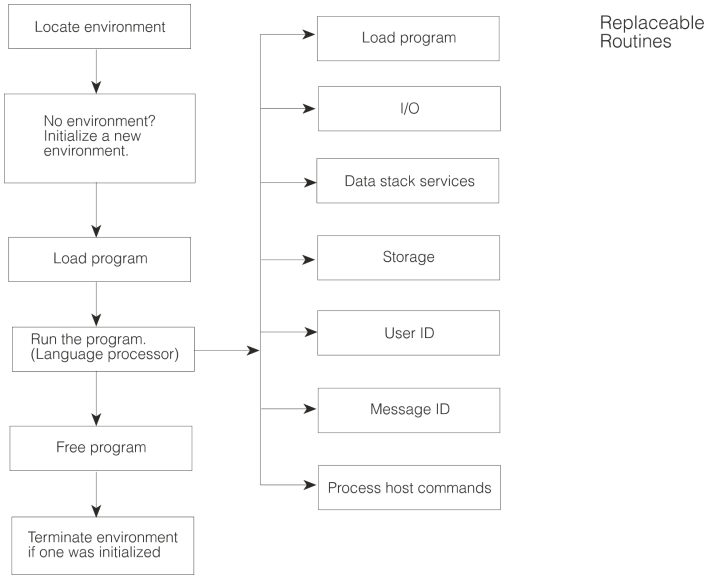


Figure 22. Overview of REXX Program Processing

As the figure shows, before the language processor runs a REXX program, a language processor environment must exist. After an environment is located or initialized, the program is loaded into storage and is then run. While a program is running, the language processor may need to access different system services, for example, to handle data stack requests or for I/O processing. Routines known as replaceable routines handle these services. The following topics describe the initialization and termination of language processor environments, the loading and freeing of a program, and the replaceable routines. There are also several exits you can provide to customize REXX processing. See "REXX Exit Routines" on page 468 for a summary of these exits.

Language Processor Environment Initialization and Termination

Before the language processor can process a REXX program, a *language processor environment* must exist. A language processor environment is the environment in which the language processor processes the program. This environment defines characteristics concerning how the program is processed and how the language processor accesses system services.

A language processor environment defines various characteristics, such as:

- The search order for locating commands and external functions and subroutines
- The member names for reading and writing data and from which REXX programs are loaded
- The host command environments you can use in a program to process host commands (that is, the environments you can specify using the ADDRESS instruction)
- The function packages (user, local, and system) that are available to programs that run in the environment and the entries in each package
- Whether programs that run 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 a program, obtaining and freeing storage, and data stack requests. These routines are known as replaceable routines.

Note: 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 program runs. This includes how a program 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 program

are one characteristic of a language processor environment. For more information about executing host commands from a REXX program, see “Commands to External Environments” on page 23.

When you use the JCL EXEC command to call a batch job or call ARXEXEX or ARXJCL to run a program, REXX/VSE automatically initializes an environment (if one does not already exist) by calling the initialization routine ARXINIT. (REXX/VSE terminates a language processor environment by calling the termination routine ARXTERM.)

As previously described, many language processor environments can exist in one partition. A language processor environment is associated with a task and environments can be chained together. Chapter 19, “Language Processor Environments,” on page 387 discusses this in more detail.

Whenever a REXX program is called, REXX/VSE first determines whether or not a language processor environment exists. If an environment does exist, the REXX program runs in that environment. If an environment does not exist, REXX/VSE automatically initializes one by calling the ARXINIT routine. When the program completes, REXX/VSE terminates the environment. “Chains of Environments and How Environments Are Located” on page 410 describes how REXX/VSE locates a previous environment.

The *parameters module* ARXPARMS contains the default values that define a language processor environment. To change the default values for initializing a language processor environment, you can provide your own parameters module. Your phase is then used instead of the default module. Chapter 19, “Language Processor Environments,” on page 387 describes the parameters module in detail.

You can also explicitly call ARXINIT to initialize a language processor environment and define the environment characteristics on the call. When you call ARXINIT, you specify any or all of the characteristics you want defined for the language processor environment. Using ARXINIT gives you the flexibility to define your own environment. This lets you customize how REXX programs run within the environment and the handling of system services. If you explicitly call ARXINIT, you must use the ARXTERM routine to terminate that environment. REXX/VSE does not automatically terminate an environment that you initialized by explicitly calling ARXINIT. See Chapter 20, “Initialization and Termination Routines,” on page 427 for descriptions of ARXINIT and ARXTERM.

Loading and Freeing a REXX Program

After a language processor environment has been located or initialized, the program must be loaded into storage for the language processor to process it. After the program runs, storage must be freed. The exec load routine loads and frees REXX programs. The default exec load routine is ARXLOAD.

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 default or that performs pre-processing and then calls the default routine ARXLOAD. The name of the load routine is defined for each language processor environment.

Note: If you use ARXEXEC to run a REXX program, you can preload the program in storage and pass the address of the preloaded program on the call to ARXEXEC. In this case, the exec load routine is not called to load the program. “Calling REXX” on page 328 describes the ARXEXEC routine and how you can preload a program.

Processing of the REXX Program

After the REXX program is loaded into storage, the language processor is called to process the program. During processing, the program 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 program 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 program is running.

Overview of Replaceable Routines

When a REXX program runs, specific routines are called to perform requested services (for example, obtaining and freeing storage, I/O, data stack requests, and so on). These routines are called *replaceable routines* because you can provide your own routines to replace the ones REXX/VSE provides.

Your routine can check the request for a system service, change the request if needed, and then call the 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 REXX/VSE routine.

Replaceable routines are defined on a language processor environment basis and are specified in the parameters module for an environment (see "Characteristics of a Language Processor Environment" on page 390).

Table 50 on page 384 provides a brief description of the functions your replaceable routine must perform. Chapter 21, "Replaceable Routines and Exits," on page 439 describes each replaceable routine in detail, its input and output parameters, and return codes.

Table 50. Overview of Replaceable Routines

Replaceable Routine	Description
Exec load	The exec load routine loads a REXX program into storage and frees the program when it is no longer needed.
Read input and write output (I/O)	The I/O routine reads a record from or writes a record to a file. (The file can be a member of a sublibrary, a SAM file, SYSIPT, or SYSLST.) 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 file.
Data stack	This routine handles 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 obtains and frees storage.
User ID	This routine obtains the user ID. You can use the USERID built-in function to obtain this result.
Message identifier	This routine determines if the message identifier (message ID) accompanies 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 21, "Replaceable Routines and Exits," on page 439 describes each routine in detail.
- Define the routine name to a language processor environment.

  If you use ARXINIT to initialize a new environment, you can pass the names of your routines on the call.

Chapter 19, "Language Processor Environments," on page 387 describes the concepts of replaceable routines and their relationship to language processor environments in more detail.

The replaceable routines are external interfaces that you can call from a program. For example, a program can call the 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 your replaceable routine or a supplied replaceable routine only if a language processor environment exists in which the routine can run.

Exit Routines

There are several exit routines you can use to customize REXX processing. Several exits have fixed names. Other exits do not. You supply the name of these exits on the call to ARXINIT or by changing the ARXPARMS default parameters module. Chapter 21, “Replaceable Routines and Exits,” on page 439 describes the exits in more detail. A summary of each exit follows.

- ARXINITX—Pre-environment initialization exit routine. The exit receives control whenever ARXINIT is called to initialize a new language processor environment. It gets control before ARXINIT evaluates any parameters.
- ARXITMV—Post-environment initialization exit routine. This exit receives control whenever ARXINIT is called to initialize a new language processor environment. It receives control after ARXINIT initializes a new environment but before ARXINIT completes.
- ARXTERMX—Environment termination exit routine. The exit receives control whenever ARXTERM is called to terminate a language processor environment. It gets control before ARXTERM starts termination processing.
- Exec Initialization — The exit receives control after the variable pool for a REXX program has been initialized but before the language processor processes the first clause in the program.
- Exec Termination — The exit receives control after a REXX program has completed processing but before the variable pool has been terminated.
- Exec processing exit (exit for the ARXEXEC routine)—This exit receives control whenever the ARXEXEC routine is called to run a REXX program. REXX/VSE or a user can explicitly call ARXEXEC to run a program. REXX/VSE always calls ARXEXEC to handle program execution. For example, if you use the JCL EXEC command or ARXJCL to call a program, REXX/VSE calls ARXEXEC to run the program. If you provide an exit for ARXEXEC, the exit is called.
- RXHLT—This is the halt processing exit.

Chapter 19. Language Processor Environments

As described in Chapter 18, “Customizing Services,” on page 381, a language processor environment is the environment in which the language processor processes a REXX program. Such an environment must exist before a program can run.

The topics in this chapter explain language processor environments and the default parameters module in more detail. They explain the various tasks you can perform to customize the environment in which REXX programs run. This chapter describes:

- Different aspects of a language processor environment and the characteristics that make up such an environment. The topic explains when REXX/VSE calls the initialization routine, ARXINIT, to initialize an environment and the values ARXINIT uses to define the environment. The topic describes the values in the default parameters module and how to change the values. It also describes what values you can and cannot specify.
- 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 to chain language processor environments together.
- How to use the data stack in different language processor environments.

Note: The control blocks 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 the processing of programs and the access and use of system services. Some of the environment characteristics include the following:

- The language in which REXX/VSE produces REXX messages
- The names of several *replaceable routines* that you can provide for system services, such as I/O processing, loading REXX programs, and processing data stack requests
- The names of exit routines that REXX/VSE calls at different points in REXX processing, such as when the ARXEXEC routine is called
- 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 programs that run in the environment
- The name of the partition (the default is VSE)
- Bit settings (flags) that define many characteristics, such as:
  - The search order for commands, functions, and subroutines
  - Whether REXX/VSE produces primary and alternate messages.

“Characteristics of a Language Processor Environment” on page 390 describes the environment characteristics.

When a language processor environment is initialized, you can define different routines that REXX/VSE calls for system services, such as obtaining and freeing storage and handling I/O requests. The language processor environment provides for consistency by ensuring that REXX programs run independently of the way in which REXX/VSE accesses system services. At the same time, the language processor environment provides flexibility to handle the differences between the partitions and lets you customize how REXX programs are processed and how REXX/VSE accesses and uses system services.

Initialization of an Environment: The initialization routine, ARXINIT, initializes language processor environments. When you use the JCL EXEC command or call ARXEXEC or ARXJCL to run a REXX program, REXX/VSE calls ARXINIT to automatically initialize an environment. Because REXX/VSE automatically initializes language processor environments, you 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" on page 390 describes when REXX/VSE initializes environments.

When REXX/VSE calls ARXINIT to automatically initialize an environment, REXX/VSE uses default values. The default parameters module (phase) ARXPARMS contains the parameter values ARXINIT uses to initialize the language processor environment.

"Characteristics of a Language Processor Environment" on page 390 describes the parameters module that contains all of the characteristics for defining a language processor environment. "Values in the ARXPARMS Default Parameters Module" on page 406 describes the defaults in ARXPARMS. You can change the default parameters by providing your own phase. "Changing the Default Values for Initializing an Environment" on page 412 describes how to change the parameters.

You can also explicitly call ARXINIT and pass the parameter values for ARXINIT to use in initializing the environment. Using ARXINIT lets you customize the environment in which REXX programs run and how REXX/VSE accesses and uses system services.

Chains of Environments: Many language processor environments can exist in a particular partition. Each language processor environment is associated with a single task. More than one environment can be associated with a particular task. Language processor environments are chained together in a hierarchical structure to form a *chain of environments* to supply a default environment if one is not specified. Each environment on a chain is related to the other environments on that chain. Environments on a particular chain may share various resources, such as files and the data stack. ("Chains of Environments and How Environments Are Located" on page 410 provides more information about the relationship between language processor environments and tasks and how environments are chained together.) A single partition can contain multiple chains of language processor environments

Maximum Number of Environments: Although many language processor environments can be initialized in a single partition, there is a maximum. ARXANCHR is a non-reentrant phase that anchors the chains of language processor environments. It contains an environment table that defines the maximum number of environments for one partition. The maximum is not a set number of environments. It 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 partition. However, if ARXINIT is initializing a new environment and this exceeds the maximum, ARXINIT 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 by providing a new ARXANCHR phase. "Changing the Maximum Number of Environments in a Partition" on page 421 describes the ARXANCHR phase and how to provide a new one.

Control Blocks: When ARXINIT initializes a new language processor environment, ARXINIT creates a number of control blocks that contain information about the environment. The main control block that ARXINIT creates is called the *environment block* (ENVBLOCK). There is an environment block for each language processor environment. 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 program currently running in the environment. "Control Blocks Created for a Language Processor Environment" on page 414 describes all of the control blocks that ARXINIT creates. ARXINIT creates an environment block for each language processor environment that it creates. Except for ARXINIT, no REXX program or service can operate without an environment being available.

Note 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 ARXINIT 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. Except for the ARXINIT initialization routine, no REXX routine can run unless an environment block exists, that is, a language processor environment must exist. When ARXINIT initializes a new language processor environment, ARXINIT always returns the address of the environment block in register 0. (If you explicitly call the ARXINIT routine, ARXINIT also returns the address of the environment block in the parameter list.) You can also use ARXINIT to obtain the address of the environment block for the current non-reentrant environment (see "Initialization Routine – ARXINIT" on page 427). ARXINIT returns the address in register 0 and also in Parameter 6 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 ARXINIT created for the environment. If you call any of the REXX/VSE routines (for example, ARXEXEC to process a program or the variable pool access interface ARXEXCOM), 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 program or the service to run. This is particularly useful if you use the ARXINIT routine to initialize several environments on a chain and then want to process a REXX/VSE routine in a specific environment. When you call 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. All calls to any programming service should pass this environment block address. Passing the environment block address is particularly important when the environment is a reentrant environment because programming services cannot automatically locate a reentrant environment. For more information about reentrant environments, see "Using the Environment Block for Reentrant Environments" on page 327.

If you call a REXX/VSE 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.

If you call ARXEXEC or ARXJCL and a language processor environment does not exist, REXX/VSE calls ARXINIT to initialize an environment in which the program runs. When the program completes processing, REXX/VSE 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 ARXINIT used in defining the environment and the addresses of REXX/VSE routines, such as ARXINIT, ARXEXEC, and ARXTERM, 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 414.

Note 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.

The following topics in this chapter describe the characteristics of a language processor environment, the different types of environments, and the default parameters module. Chapter 20, "Initialization and Termination Routines," on page 427 describes the initialization and termination routines ARXINIT and ARXTERM.

When Environments Are Automatically Initialized

If a language processor environment does not already exist on the current task, REXX/VSE automatically initializes one whenever:

- You use the JCL EXEC command to call a batch job
- A program calls ARXEXEC or ARXJCL to call a REXX program.

"Calling REXX with ARXEXEC or ARXJCL" on page 331 describes these routines in detail. When ARXEXEC or ARXJCL is called, it determines whether a language processor environment already exists. (As discussed previously, more than one environment can be initialized in a single partition. The environments are chained together in a hierarchical structure). ARXEXEC and ARXJCL do not invoke ARXINIT to initialize an environment if one already exists. The routines use the current environment to run the program. "Chains of Environments and How Environments Are Located" on page 410 describes how language processor environments are chained together and how environments are located.

If ARXEXEC or ARXJCL invokes the ARXINIT routine to initialize an environment, after the REXX program completes processing, REXX/VSE calls the ARXTERM routine to terminate the environment that ARXINIT initialized.

Note: If several language processor environments already exist, you can pass the address of an environment block in register 0 on the call to ARXEXEC or ARXJCL. This indicates the environment in which the program should run. See "Using the Environment Block" on page 389 for more information. Chapter 21, "Replaceable Routines and Exits," on page 439 describes the replaceable routines and exits in more detail.

"Specifying Values for Different Environments" on page 413 describes the environment characteristics you can specify for language processor environments.

Characteristics of a Language Processor Environment

When ARXINIT initializes a language processor environment, ARXINIT 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 ARXINIT used to define the environment. The parameter 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.

REXX/VSE provides a default *parameters module* ARXPARMS. This is a phase that contains the values for initializing language processor environments. "Values in the ARXPARMS Default Parameters Module" on page 406 shows the default values for this module. A parameters module consists of the parameter block (PARMBLOCK), the module name table, the host command environment table, and the function package table. Figure 23 on page 391 shows the format of the parameters module.

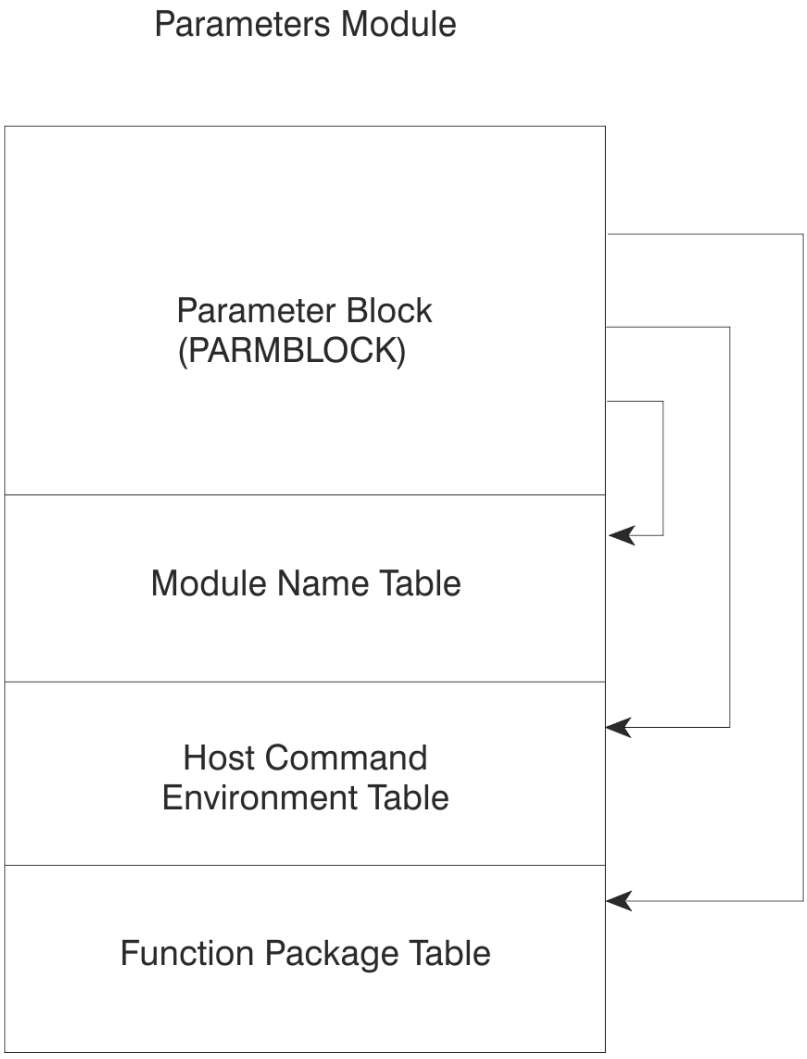


Figure 23. Overview of Parameters Module

Table 51 on page 391 shows the format of PARMBLOCK. Each field is described in more detail after the table. Indicate the end of the PARMBLOCK with X'FFFFFFFFFFFFFFFF'. Subsequent topics describe the format of the module name table, host command environment table, and function package table. A mapping macro for the parameter block, ARXPARMB, is in PRD1.BASE.

Note: Each field name in the following table must include the prefix PARMBLOCK_.

Table 51. 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.

Table 51. Format of the Parameter Block (PARMBLOCK) (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
24	4	PACKTB	Address of function package table.
28	8	PARSETOK	Token for PARSE SOURCE instruction.
36	4	FLAGS	A fullword of bits that ARXINIT uses as flags to define characteristics for the environment.
40	4	MASKS	A fullword of bits that ARXINIT uses as a mask for the setting of the flag bits.
44	4	SUBPOOL	This field is reserved.
48	8	ADDRSPN	Name of the partition.
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 the default parameters module or use ARXINIT to initialize a language processor environment, read "Changing the Default Values for Initializing an Environment" on page 412 for information about changing the values that define an environment.

ID
This field identifies the parameter block that ARXINIT creates. The value for ID is ARXPARMS. Do not change this value.

VERSION
This field identifies the version of the parameter block for a particular release and level of REXX/VSE.

The value for VERSION is 0001. Do not change this value.

LANGUAGE
This field contains a language code that identifies the language for producing REXX messages. The only valid values are:

- ENU—the language code for US English in mixed case (upper and lowercase)
- ENP (US English in upper case).

Reserved
This field is reserved.

MODNAMET
This field contains the address of the module name table. The table contains the file names for reading and writing data and the names of several replaceable routines and exit routines. "Module Name Table" on page 398 describes the table in detail.

SUBCOMTB
This field contains the address of the host command environment table.

This table contains the names of the host command environments for processing host commands. These are the environments that REXX programs can specify using the ADDRESS instruction. "Commands to External Environments" on page 23 describes how to issue host commands from a REXX program and the different environments for command processing.

The table also contains the names of the routines that are called to handle the processing of commands that are issued in each host command environment. "Host Command Environment Table" on page 401 describes the table in detail.

PACKTB
This field contains the address of the function package table for function packages. "Function Package Table" on page 403 describes the table in detail.

PARSETOK
This field is a character string containing the value of a token that the PARSE SOURCE instruction uses. This token is the last token of the string that PARSE SOURCE returns.

FLAGS
The FLAGS field is a fullword of bits that ARXINIT uses as flags. The flags define certain characteristics for the new language processor environment and how the environment and programs running in the environment operate.

See "Flags and Corresponding Masks" on page 393 for details about each flag. The mapping of the parameter block (PARMBLOCK) includes the mapping of the flags. REXX/VSE provides a mapping macro ARXPARMB for the parameter block. The mapping macro is in PRD1.BASE. The parameter after the flags is a *mask* field that works with the flags.

MASKS
This field is a fullword of bits that ARXINIT uses as a mask for setting the flag bits. See the preceding field for details about the flags field. 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. ARXINIT 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 1 the corresponding bit in the flags field is used. If it is 0, the corresponding bit in the flags field is ignored (that is, the bit is considered null).

SUBPOOL
This field is reserved.

ADDRSPN
This field specifies the name of the partition. This value is VSE.

X'FFFFFFFFFFFFFFFF'
X'FFFFFFFFFFFFFFFF' indicates the end of the parameter block.

Flags and Corresponding Masks

The following table summarizes the flags field.

Table 52. Summary of Each Flag Bit in the Parameters Module

Bit Position Number	Flag Name	Description
0	TSOFL	This bit is reserved and must be off.
1	Reserved	This bit is reserved.
2	CMDSOFL	Specifies the search order REXX/VSE uses for locating a command.
3	FUNCSOFL	Specifies the search order REXX/VSE uses for locating functions and subroutines.
4	NOSTKFL	Specifies whether REXX programs running in the environment can use data stack operations.
5	NOREADFL	Specifies whether REXX programs running in the environment can read from input files.
6	NOWRTFL	Specifies whether REXX programs running in the environment can write to output files.
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.

Table 52. Summary of Each Flag Bit in the Parameters Module (continued)

Bit Position Number	Flag Name	Description
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 member from which REXX programs are obtained is closed after a program is loaded or remains open.
13	NOESTAE	This bit is reserved.
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	This bit is reserved.
18	STORFL	Indicates whether REXX programs running in the environment can use the STORAGE function.
19	NOLOADDD	This bit is reserved.
20	NOMSGWTO	Indicates whether REXX error messages are issued.
21	NOMSGIO	Indicates whether REXX error messages with I/O are issued to the current output.
22	Reserved	The remaining bits are reserved.

TSOFL
This field is reserved and must be `0`.

CMDSOFL
The CMDSOFL flag specifies the search order REXX/VSE uses for locating a command that is issued from a program.

0
Indicates searching for a phase in the active PHASE chain, then for a program in the active PROC chain.

1
Indicates searching the PROC chain for a program first, then search for a phase in the active PHASE chain.

FUNCSOFL
The FUNCSOFL flag specifies the search order REXX/VSE uses for locating external functions and subroutines that a program calls.

0
Indicates searching for a phase in the active PHASE chain, then for a program in the active PROC chain.

1
Indicates searching the PROC chain for a program first, then search for a phase in the active PHASE chain.

NOSTKFL
The NOSTKFL flag specifies whether REXX programs running in the environment can use the data stack.

0
A REXX program 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 specifies whether REXX programs can read input files using the EXECIO command or the I/O replaceable routine ARXINOUT.

0
Permits reading from input files.

1
Prohibits reading from input files.

NOWRTFL
The NOWRTFL flag specifies whether REXX programs can write to output files using the EXECIO command or the supplied I/O replaceable routine ARXINOUT.

0
Permits writing to output files.

1
Prohibits writing to output files.

NEWSTKFL
The NEWSTKFL flag specifies whether ARXINIT should initialize a new data stack for the language processor environment. If ARXINIT creates a new data stack, any REXX program 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 accesses the data stack the NEWSTKFL flag most recently created. The first environment that is initialized on any chain of environments is always initialized as though the NEWSTKFL flag is on, that is, ARXINIT 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. (“Using the Data Stack” on page 422 describes the data stack in different environments. Note that NOSTKFL overrides the setting of the NEWSTKFL.)

0
ARXINIT does not create a new data stack. However, if this is the first environment being initialized on a chain, ARXINIT automatically initializes a data stack.

1
ARXINIT creates a new data stack during the initialization of the new language processor environment. The data stack is deleted when the environment is terminated.

USERPKFL
The USERPKFL flag specifies whether the user function packages that are defined for the previous language processor environment are also available to the new environment.

0
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 specifies whether local function packages 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 specifies whether the system function packages 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 specifies whether the environments for issuing host commands that are defined for the previous language processor environment are also available to programs 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 specifies whether the member from which REXX programs are fetched is closed after the program is loaded or remains open.

You need to close the member if you are editing REXX programs and then running the changed programs under the same language processor environment. If you do not close the member, results may be unpredictable.

0
The member is opened once and remains open.

1
The member is opened for each load and then closed.

NOESTAE
This bit is reserved.

RENTRANT
The RENTRANT flag specifies whether ARXINIT initializes the new environment as a reentrant or a non-reentrant environment. (For information about reentrant environments, see "Using the Environment Block for Reentrant Environments" on page 327.)

0
ARXINIT initializes a non-reentrant language processor environment.

1
ARXINIT initializes a reentrant language processor environment.

NOPMSGS
The NOPMSGS flag specifies whether REXX primary messages are printed in the environment.

0
Primary messages are printed.

1
Primary messages are not printed.

ALTMSGS
The ALTMSGS flag specifies whether REXX alternate messages are printed in the environment. (Alternate messages are also known as secondary messages.)

0
Alternate messages are not printed.

1
Alternate messages are printed.

SPSHARE
This bit is reserved.

STORFL
The STORFL flag specifies whether REXX programs running in the environment can use the STORAGE external function.

0
Programs can use the STORAGE external function.

1
Programs cannot use the STORAGE external function.

NOLOADDD
This bit is reserved.

NOMSGWTO and NOMSGIO
Together, these two flags control where REXX error messages are routed.

Table 53. Flag Settings for NOMSGWTO and NOMSGIO

NOMSGWTO	NOMSGIO	
0	0	ASSGN(STDOUT) returns the name of the current output stream. Error messages are written to the current output stream and SYSLOG. If the current output is SYSLOG, messages are written to SYSLOG only. This happens regardless of whether tracing is active.
1	0	REXX error messages are written to the current output. If the current output is SYSLOG, messages are suppressed. This happens regardless of whether REXX tracing is active.
0	1	REXX error messages cannot be written to the current output. Instead, error messages are written to SYSLOG. 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 are off (0) for both NOMSGWTO and NOMSGIO.

REXX error messages include all of the REXX messages numbered ARX*nnnn*E or ARX*nnnn*I, where *nnnn* is the message number. Error messages also include any text written to the error message output stream using the 'WRITEERR' function of ARXSAY.

Module Name Table

The module name table contains the names of:

- The file or device for reading and writing data
- 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 390).

Table 54 on page 398 shows the format of the module name table. Each field is described in detail following the table. Indicate the end of the table with X'FFFFFFFFFFFFFFFF'. REXX/VSE provides a mapping macro ARXMODNT for the module name table. The mapping macro is in PRD1.BASE.

Note: Each field name in the following table must include the prefix MODNAMET_.

Table 54. Format of the Module Name Table

Offset (Decimal)	Number of Bytes	Field Name	Description
0	8	INDD	The file or device from which PARSE EXTERNAL, PULL, PARSE PULL, and interactive debug pause read input data. **Note:** You receive an error if you do not provide your own I/O replaceable routine and are using a file name other than: • SYSLOG • SYSIPT • SYSLST • SYS*xxx* (where xxx is numeric) • Any other 7-character name. See “Input/Output Routine” on page 446 for details about supplying a replaceable routine. You need to open a SAM file (using EXECIO... (OPEN) before reading from or writing to the file. SYSIPT, SYSLST, and SAM files you have opened use the replaceable routine ARXINOUT.

Table 54. Format of the Module Name Table (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
8	8	OUTDD	The file or device to which data is written for either a SAY instruction, for REXX error messages, or when tracing is started. **Note:** You receive an error if you do not provide your own I/O replaceable routine and are using a file name other than: • SYSLOG • SYSIPT • SYSLST • SYS*xxx* (where xxx is numeric) • Any other 7-character name. See "Input/Output Routine" on page 446 for details about supplying a replaceable routine. You need to open a SAM file (using EXECIO... (OPEN) before reading from or writing to the file. SYSIPT, SYSLST, and SAM files you have opened use the replaceable routine ARXINOUT.
16	8	LOADDD	Reserved.
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	Reserved.
64	8	STACKRT	The name of the data stack replaceable routine.
72	8	IRXEXECX	The name of the exit routine for the ARXEXEC 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.
104	8	RXHLT	Name of the REXX halt exit.
112	8	—	The end of the module name table must be indicated by X'FFFFFFFFFFFFFFFF'.

Each field in the module name table is described in the following.

INDD
Specifies the name of the file or device from which PARSE EXTERNAL, PULL, PARSE PULL, and interactive debug pause read input data. ASSGN(STDIN) returns the name of the current input stream.

OUTDD
Specifies the name of the file or device to which data is written for a SAY instruction, for REXX error messages, or when tracing is started. ASSGN(STDOUT) returns the name of the current output stream.

LOADDD
This field is reserved.

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
- The PULL instruction
- Requests from the EXECIO command
- Issuing REXX error messages

For more information, see "Input/Output Routine" on page 446.

EXROUT
Specifies the name of the routine that is called to load and free a REXX program. The routine returns the structure that is described in "The In-Storage Control Block (INSTBLK)" on page 339. The specified routine is called to load and free this structure. For more information, see "Exec Load Routine" on page 442.

GETFREER
Specifies the name of the routine that is called when storage is to be obtained or freed. If this field is blank, storage routines handle storage requests and use the GETVIS and FREEVIS macros when larger amounts of storage must be handled. For more information, see "Storage Management Routine" on page 463.

EXECINIT
Specifies the name of an exit routine that gets control after REXX/VSE initializes the REXX variable pool for a REXX program, but before the language processor processes the first clause in the program. You provide the exit and specify the routine's name in the EXECINIT field. "REXX Exit Routines" on page 468 describes the exec initialization exit.

ATTNROUT
This field is reserved.

STACKRT
Specifies the name of the routine that REXX/VSE calls to handle all data stack requests. For more information, see "Data Stack Routine" on page 459.

IRXEXECX
Specifies the name of an exit routine that is invoked whenever the ARXEXEC routine is called to run a program. You can use the exit to check the parameters specified on the call to ARXEXEC, change the parameters, or decide whether or not ARXEXEC processing should continue.

You provide the exit and specify the routine's name in the IRXEXECX field. For more information, see "REXX Exit Routines" on page 468.

IDROUT
Specifies the name of a replaceable routine that REXX/VSE calls to obtain the user ID. The USERID built-in function returns the result that the replaceable routine obtains. For more information, see "User ID Routine" on page 465.

MSGIDRT
Specifies the name of a replaceable routine that determines whether REXX/VSE should include the message identifier (message ID) with a REXX error message. For more information, see "Message Identifier Routine" on page 467.

EXECTERM
Specifies the name of an exit routine that gets control after the language processor processes a REXX program, but before REXX/VSE terminates the REXX variable pool. You provide the exit and specify

the routine's name in the EXECTERM field. “REXX Exit Routines” on page 468 describes the exit in more detail.

RXHLT
Specifies the name of the halt exit. See “Halt Exit” on page 472 for more information about the halt exit.

X'FFFFFFFFFFFFFFFF'
Indicate the end of the module name table with X'FFFFFFFFFFFFFFFF'.

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 “Characteristics of a Language Processor Environment” on page 390).

The table contains the environment names (for example, VSE, POWER, LINK, LINKPGM, JCL, and CONSOLE) that are valid for programs that run in the language processor environment. The table also contains the names of the routines that REXX/VSE calls to handle "commands" for each host command environment.

You can add, delete, update, and query entries in the host command environment table using the ARXSUBCM routine. For more information, see “Maintain Entries in the Host Command Environment Table – ARXSUBCM” on page 357.

When a REXX program runs, the program has at least one active host command environment that processes host commands. When the REXX program begins processing, a default environment is available. The default is specified in the host command environment table. In the REXX program, 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 23 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 55 on page 401 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. One row in the table defines each host command environment entry. Each row contains the environment name, corresponding routine to handle the commands, and a user token. Table 56 on page 402 illustrates the rows of entries in the table. A mapping macro for the host command environment table, ARXSUBCT, is in PRD1.BASE.

Note: Each field name in the following table must include the prefix SUBCOMTB_.

Table 55. Format of the Host Command Environment Table Header

Offset (Decimal)	Number of Bytes	Field Name	Description
0	4	FIRST	Specifies the address of the first entry in the table. The address is a fullword binary number. Table 56 on page 402 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, another 8-byte field (ROUTINE) that contains the name of the corresponding routine, followed by a 16-byte field (TOKEN) that is a user token.

Table 55. Format of the Host Command Environment Table Header (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
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 used entries in the table. The number is a fullword binary number. All valid entries begin at the top of the table. Any unused entries follow these. 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.
16	4	INITIAL	Specifies the name of the initial host command environment. The default is VSE. This is the default environment for any REXX program that is not called as a function or subroutine. The INITIAL field is used only if you call the exec processing routine ARXEXEC to run a REXX program and you do not pass an initial host command environment on the call. “Calling REXX” on page 328 describes the ARXEXEC routine and its parameters.
20	8	—	Reserved. The field is set to blanks.
28	8	—	Indicate the end of the table header with X'FFFFFFFFFFFFFFFF'.

Table 56 on page 402 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.

Note: Each field name in the following table must include the prefix SUBCOMTB_.

Table 56. 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 REXX/VSE calls 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 REXX/VSE calls 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.

Table 56. Format of Entries in Host Command Environment Table (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
64	8	NAME	The name of the third environment (entry) in the table.
72	8	ROUTINE	The name of the routine that REXX/VSE calls 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 this row in the table defines. The string is 8 characters long, left justified, and padded with blanks.

If the REXX program 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 8 characters long, left justified, and 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 being unable 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 REXX/VSE 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 program is running in the language processor environment and a host command environment must be located, REXX/VSE 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), REXX/VSE 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 the active PHASE chain and active PROC chain (see “Search Order” on page 60).

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 344 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 390). 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 57 on page 404 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 8-character fields that are contiguous. Each 8-character field contains the name of a function package, which is the name of a phase 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 348 describes the format of the function package directory in detail.

Figure 24 on page 406 illustrates the 8-character fields that contain the function package directory names for user, local, and system function packages.

REXX/VSE provides a mapping macro for the function package table. The name of the mapping macro is ARXPACKT. The mapping macro is in PRD1.BASE.

Note: Each field name in the following table must include the prefix PACKTB_.

Table 57. 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 8-character fields that contain the names of the function package directories for user packages. Figure 24 on page 406 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.

Table 57. Function Package Table Header (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
8	4	USER_USED	Specifies the total number of user package table entries in use. 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 8-character fields that contain the names of the function package directories for local packages. Figure 24 on page 406 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 8-character fields that contain the names of the function package directories for system packages. Figure 24 on page 406 shows the series of directory names.
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.

Table 57. Function Package Table Header (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
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	—	Indicate the end of the table with X'FFFFFFFFFFFFFFFF'.

Figure 24 on page 406 shows the function package table entries that are the names of the directories for user, local, and system function packages.

Figure 24. Function Package Table Entries – Function Package Directories

The table entries are a series of 8-character fields. Each field contains the name of a function package directory. The directory is a phase that, when loaded, contains information about each external function and subroutine in the function package. "Directory for Function Packages" on page 348 describes the format of the function package directory in detail.

The function package directory names in each 8-character field must be left justified and padded with blanks.

Values in the ARXPARMS Default Parameters Module

Table 58 on page 407 shows the default values in ARXPARMS. "Characteristics of a Language Processor Environment" on page 390 describes the structure of the parameters module in detail.

In the figure, the LANGUAGE field contains the language code ENU for US English in mixed case (upper and lowercase). The default parameters module may contain a different language code depending on whether one of the language features has been installed. See page "LANGUAGE " on page 392 for information about the different language codes.

In the figure, the value of each flag setting is followed by the value of its corresponding mask setting, in parentheses.

Note: Table 58 on page 407 shows the default values in the parameters module. 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 390. The ARXPARMB mapping macro is for the parameter block and the ARXMODNT, ARXSUBCT, and ARXPACKT mapping macros are for the module name table, host command environment table, and function package table respectively.

Table 58. Values in ARXPARMS Default Parameters Module (1)

Field Name	ARXPARMS
ID	ARXPARMS
VERSION	0001
LANGUAGE	ENU
PARSETOK	
FLAGS (MASKS)	
TSOFL	0 (1) (This field is reserved.)
CMDSOFL	0 (1)
FUNCSOFL	0 (1)
NOSTKFL	0 (1)
NOREADFL	0 (1)
NOWRTFL	0 (1)
NEWSTKFL	0 (1)
USERPKFL	0 (1)
LOCPKFL	0 (1)
SYSPKFL	0 (1)
NEWSCFL	0 (1)
CLOSEXFL	0 (1)
NOESTAE	0 (1) (This field is reserved.)
RENTRANT	0 (1)
NOPMSGS	0 (1)
ALTMSGS	1 (1)
SPSHARE	0 (1) (This field is reserved.)
STORFL	0 (1)
NOLOADDD	0 (1) (This field is reserved.)
NOMSGWTO	0 (1)
NOMSGIO	0 (1)
SUBPOOL	0 (This field is reserved.)
ADDRSPN	VSE
—	FFFFFFFFFFFFFFFF

Table 59. Values in ARXPARMS Default Parameters Module (2)

Field Name in Module Name Table	ARXPARMS
INDD	SYSIPT
OUTDD	SYSLST
LOADDD	Reserved
IOROUT	

Table 59. Values in ARXPARMS Default Parameters Module (2) (continued)

Field Name in Module Name Table	ARXPARMS
EXROUT	
GETFREER	
EXECINIT	
ATTNROUT	Reserved
STACKRT	
IRXEXECX	
IDROUT	
MSGIDRT	
EXECTERM	
RXHLT	
—	FFFFFFFFFFFFFFFF

Table 60. Values in ARXPARMS Default Parameters Module (3)

Field Name in Host Command Environment Table	ARXPARMS
TOTAL	10
USED	6
LENGTH	32
INITIAL	VSE
—	FFFFFFFFFFFFFFFF
Entry 1	
NAME	VSE
ROUTINE	ARXSTAM
TOKEN	
Entry 2	
NAME	POWER
ROUTINE	ARXSTAM
TOKEN	
Entry 3	
NAME	LINK
ROUTINE	ARXSTAM
TOKEN	
Entry 4	
NAME	LINKPGM
ROUTINE	ARXSTAM
TOKEN	
Entry 5	

Table 60. Values in ARXPARMS Default Parameters Module (3) (continued)

Field Name in Host Command Environment Table	ARXPARMS
NAME	JCL
ROUTINE	ARXJCLAD
TOKEN	
Entry 6	
NAME	CONSOLE
ROUTINE	ARXCONAD
TOKEN	

How ARXINIT Determines What Values to Use for the Environment

When REXX/VSE calls ARXINIT to automatically initialize a language processor environment, ARXINIT must first determine what values to use for the environment. The following topics describe how ARXINIT determines the values for a new environment. “Chains of Environments and How Environments Are Located” on page 410 describes how a routine locates a previous environment.

Values ARXINIT Uses to Initialize Environments

When JCL or an application program needs to call a REXX program, ARXINIT automatically initializes an environment for the REXX program. (See also “Initialization Routine – ARXINIT” on page 427.) ARXINIT determines the values to use for defining the environment from:

1. The in-storage parameter list specified on the call to ARXINIT (On the call to ARXINIT, you can pass parameters that define the values for the environment. ARXINIT evaluates these.)
2. The parameters module specified on the call to ARXINIT
3. The previous language processing environment (“Chains of Environments and How Environments Are Located” on page 410 describes in detail how ARXINIT locates a previous environment.)
4. The ARXPARMS parameter module.

ARXINIT first checks the values in the in-storage parameter list specified on the call to ARXINIT. If the value is not null, ARXINIT uses that value. ARXINIT considers the following types of parameter values to be null:

- A character string containing only blanks or of length 0
- An address of 0
- A binary number with the value X'80000000'
- A bit setting with a corresponding mask of 0.

If the value in the parameters module is null, ARXINIT uses the value from the parameter module specified on the call to ARXINIT. If this value is null, ARXINIT uses the value from the previous language processor environment. If an environment does not exist, ARXINIT uses the value from the ARXPARMS parameters module. ARXINIT computes each individual value using this method and then initializes the environment.

For example, if the in-storage parameter list does not include a value for ADDRSPN, ARXINIT uses the value from the parameter module specified on the call to ARXINIT. Suppose the parameter module is not ARXPARMS. In this case, the value can be null, and ARXINIT would check the previous language processor environment. If there is no previous environment, ARXINIT checks ARXPARMS, finding the value VSE.

After ARXINIT determines all of the values, ARXINIT initializes the new environment.

Chains of Environments and How Environments Are Located

As described in previous topics, many language processor environments can be initialized in one partition. A language processor environment is associated with a task. Several language processor environments can be associated with a single task. This topic describes how non-reentrant environments are chained together in a partition.

Language processor environments 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 partition are independent.

Figure 25 on page 410 illustrates three language processor environments that form one chain.

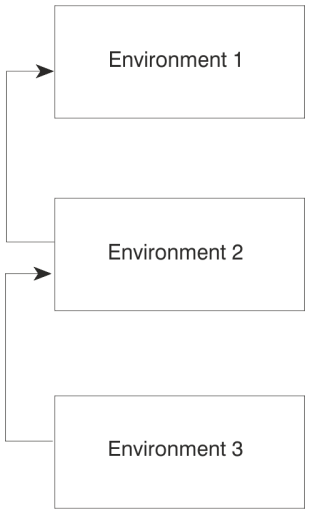


Figure 25. Three Language Processor Environments in a Chain

The first environment initialized was environment 1. When ARXINIT 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 ARXINIT 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 partition. Figure 26 on page 410 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.

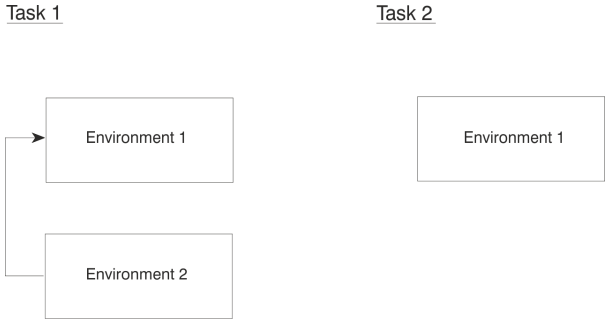


Figure 26. Separate Chains on Two Different Tasks

As discussed previously, language processor environments are associated with a task. Under a task, ARXINIT can initialize one or more language processor environments. The task can then attach another task. ARXINIT 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 27 on page 411 illustrates a task that has attached another task and how the language processor environments are chained together.

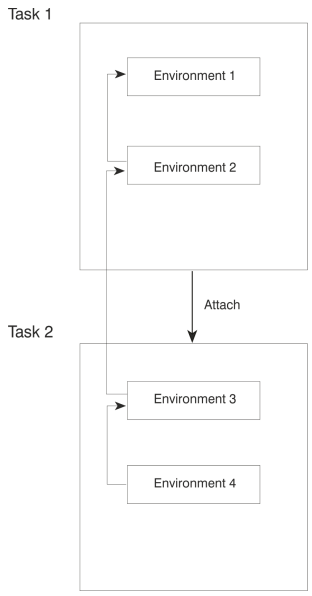


Figure 27. One Chain of Environments for Attached Tasks

As shown in Figure 27 on page 411, task 1 is started and ARXINIT initializes an environment (environment 1). ARXINIT is called 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 program within task 1, the program runs in environment 2.

Task 1 then attaches another task, task 2. ARXINIT is called to initialize an environment. ARXINIT locates the previous environment, which is environment 2, and chains the new environment (environment 3) to its parent (environment 2). When ARXINIT is called again, ARXINIT 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

Whenever you invoke a REXX program or routine, the program or routine must run in a language processor environment. The one exception is the initialization routine, ARXINIT, which initializes environments.

If you call a program using ARXEXEC or ARXJCL, a language processor environment may or may not already exist. If an environment does not exist on the current task, REXX/VSE calls the ARXINIT routine to initialize an environment before the program runs. Otherwise, REXX/VSE locates the current non-reentrant environment and the program runs in that environment.

ARXINIT always tries to locate a previous language processor environment. If an environment does not exist on the current task or on a parent task, ARXINIT uses the values in the ARXPARMS parameters module as the previous environment.

A language processor environment must already exist if you call any of the programming routines or replaceable routines. These routines do not invoke ARXINIT 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 17, “Programming Services,” on page 323 for information about the programming routines and Chapter 21, “Replaceable Routines and Exits,” on page 439 for information about the replaceable routines.

When ARXINIT initializes a new language processor environment, ARXINIT creates a number of control blocks that contain information about the environment and any REXX program 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 414 describes the control blocks that ARXINIT creates for each language processor environment.

The environment block represents its language processor environment and is the anchor that REXX/VSE 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 the language processor environment in which you want the routine to run. For example, suppose you call the initialization routine, ARXINIT. On return, ARXINIT 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 ARXINIT several times to initialize a total of four environments in that partition. If you then want to call a 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.

When a programming service routine is called, the programming service routine must determine in which environment to run. The routine checks 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. If register 0 does not contain the address of a valid environment block, the routine that is called searches for a non-reentrant environment on the current task. If the routine could not find an 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, ARXTERM.
- If ARXEXEC or ARXJCL was called, the routine invokes ARXINIT to initialize a new environment.
- If ARXINIT was called, ARXINIT uses the ARXPARMS parameters module as the previous environment.

The ARXINIT routine initializes a new language processor environment. Therefore, ARXINIT does not need to locate an environment in which to run. However, ARXINIT does locate a previous environment to determine what values to use when defining the new environment. The following summarizes the steps ARXINIT takes to locate the previous environment:

1. If register 0 contains the address of a valid environment block, ARXINIT uses that environment as the previous environment.
2. If a non-reentrant environment exists on the current task, ARXINIT uses the last non-reentrant environment on the task as the previous environment.
3. Otherwise, ARXINIT locates the parent task. If a non-reentrant environment exists on any of the parent tasks, ARXINIT uses the last non-reentrant environment on the task as the previous environment.
4. If ARXINIT cannot find an environment, ARXINIT uses the values in the default parameters module ARXPARMS as the previous environment.

“Initialization Routine – ARXINIT” on page 427 describes how the ARXINIT routine determines what values to use when you explicitly call ARXINIT.

Changing the Default Values for Initializing an Environment

The parameters module (phase) contains default values for initializing language processor environments. 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 phase that contains your values.

Note: You can also call the initialization routine, ARXINIT, 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, ARXINIT uses the value from the previous environment. See "Initialization Routine – ARXINIT" on page 427 for more information.

This topic describes how to create a phase containing parameter values for initializing an environment. You should also refer to "Characteristics of a Language Processor Environment" on page 390 for information about the format of the parameters module.

To change one or more default values that ARXINIT uses to initialize a language processor environment, you can provide a phase containing the values you want. You must first write the code for a parameters module. PRD1.BASE contains a sample that is assembler code for the default parameters module. The member name of the sample is: ARXPARMS.Z (for ARXPARMS).

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 ARXPARMS module. 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 in the ARXPARMS Default Parameters Module" on page 406 shows the default parameter values for ARXPARMS.

After you create the code, you must assemble the code and then link-edit the object code. The output is a member of a sublibrary with a member type of PHASE. You must then place the member in the active PHASE chain. The default parameters module is in PRD1.BASE. ARXPARMS.Z suggests putting the changed parameters module into PRD2.CONFIG. However, you may also place your phase somewhere ahead of the default parameters module in the active PHASE chain by using a LIBDEF statement.

The new values you specify in your own phase are not available until the current language processor environment is terminated and a new environment is initialized. For example, if you provide a phase (ARXPARMS), you must reinitialize the environment.

Providing Your Own Parameters Module

The sample ARXPARMS.Z is in PRD1.BASE. You can use this to code your own phases.

Changing Values

If you want to change a default parameter value, code a new ARXPARMS 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. See "Values in the ARXPARMS Default Parameters Module" on page 406 to review the defaults in the ARXPARMS parameters module. When you assemble the code and link-edit the object code, you must name the output member ARXPARMS. You must then place the phase with ARXPARMS in PRD2.CONFIG or in a sublibrary with type PHASE that precedes PRD1.BASE in the active PHASE chain. You can do this using JCL.

If you provide your own ARXPARMS module, ARXINIT locates the module when initializing a language processor environment. The values for the replaceable routines in the default parameters module are null. You can code your own ARXPARMS phase and specify the names of one or more replaceable routines.

For more information about the parameters you can use in different language processor environments, see "Specifying Values for Different Environments" on page 413.

Specifying Values for Different Environments

You can also call the initialization routine, ARXINIT, to initialize a new environment. When you call ARXINIT, you can pass parameter values on the call. Chapter 20, "Initialization and Termination Routines," on page 427 describes ARXINIT and its parameters and return codes.

Whether you provide your own phase or call ARXINIT directly you cannot change some parameters. There are also some restrictions on parameter values based on the values of other parameters in the same environment and parameters in the previous environment. This topic describes considerations for using

the parameters. For more information about the parameters and their descriptions, see "Characteristics of a Language Processor Environment" on page 390.

Parameters You Cannot Change

The following parameters have fixed values that you cannot change.

ID
The value must be ARXPARMS. If you provide your own phase, you must specify ARXPARMS for the ID. If you call ARXINIT, ARXINIT ignores any value you pass and uses the default ARXPARMS.

VERSION
The value must be 0001. If you provide your own phase or call ARXINIT, specify 0001 for the version.

The following parameters are reserved, and you should not attempt to change them:

- TSOFL
- NOLOADDD
- SPSHARE
- NOESTAE.

Control Blocks Created for a Language Processor Environment

When ARXINIT initializes a new language processor environment, ARXINIT 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 ARXINIT used to define the environment. The parameter block ARXINIT creates has the same format as the parameters module.
- The user field that was passed on the call to ARXINIT if a user explicitly called ARXINIT
- The work block extension, which is a control block that contains information about the REXX program that is currently running
- The REXX vector of external entry points, which contains the addresses of the REXX routines, such as ARXINIT, ARXTERM, REXX programming routines, and replaceable routines. For replaceable routines, the vector contains the addresses of both the routines that REXX/VSE supplies and any routines that users provide.
- The 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.

Note 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 61 on page 415 shows the format of the environment block. A mapping macro for the environment block, ARXENVB, is in PRD1.BASE.

When ARXINIT initializes a new language processor environment, ARXINIT 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.

Note: The following field names in the table must include the prefix ENVBLOCK_: ID, VERSION, LENGTH, PARMBLOCK, USERFIELD, WORKBLOK_EXT, IRXEXTE, COMPGMTB, ATTNROUT_PARMPTR, ECPTR.

Table 61. Format of the Environment Block

Offset (Decimal)	Number of Bytes	Field Name	Description
0	8	ID	An 8-character field that identifies the environment block. The field contains the characters 'ENVBLOCK'.
8	4	VERSION	A field that contains the character representation of the version number of the environment block. The version number is 0001.
12	4	LENGTH	The length of the environment block.
16	4	PARMBLOCK	The address of the parameter block (PARMBLOCK). See "Format of the Parameter Block (PARMBLOCK)" on page 416 for more information.
20	4	USERFIELD	The address of the user field that is passed to ARXINIT if you explicitly call ARXINIT. You pass the user field in parameter 4 (see "Initialization Routine – ARXINIT" on page 427 for information about the parameters). You can use this field for your own processing. The REXX/VSE services do not use this field.
24	4	WORKBLOK_EXT	The address of the current work block extension. If a program is not currently running in the environment, the address is 0. See "Format of the Work Block Extension" on page 416 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 418 for details about the vector.
32	4	ERROR_CALL@	The address of the routine that encountered the first error in the language processor environment and that issued the first error message. The error could have occurred while a program was running or when a particular service was requested in the environment.
36	4	—	Reserved.
40	8	ERROR_MSGID	An 8-character field that contains the message ID of the first error message REXX/VSE issued in the language processor environment. The message relates to the error the routine encountered and to which offset +32 points.

Table 61. Format of the Environment Block (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
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	**This field is a product-sensitive programming interface**. 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 "The Compiler Programming Table" on page 501.
292	4	ATTNROUT_PARMPTR	This field is reserved.
296	4	ECTPTR	This field is reserved.
300	4	—	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 Chapter 23, "ARXTERMA Routine," on page 495 for information about abnormal termination.)

Format of the Parameter Block (PARMBLOCK)

The parameter block (PARMBLOCK) contains information about the parameters that ARXINIT uses to define the environment. The environment block points to the parameter block. Table 51 on page 391 shows the format of the parameter block.

Format of the Work Block Extension

The work block extension contains information about the REXX program that is currently running. The environment block points to the work block extension.

When ARXINIT 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 program is not yet running in the environment. At this point, ARXINIT is only initializing the environment.

When a program starts running in the environment, the environment block is updated to point to the work block extension describing the program. If a program is running and calls another program, the environment block is updated to point to the work block extension for the second program. The work block extension for the first program still exists, but the environment block does not point to it. When the

second program completes and returns control to the first program, the environment block is changed again to point to the work block extension for the original program.

The work block extension contains the parameters that are passed to the ARXEXEC routine to invoke the program. You can call ARXEXEC explicitly to invoke a program and pass the parameters on the call. If you use ARXJCL and invoke a program, the ARXEXEC routine always gets control to run the program. "The ARXEXEC Routine" on page 334 describes the ARXEXEC routine in detail.

Table 62 on page 417 shows the format of the work block extension. A mapping macro for the work block extension, ARXWORKB, is in PRD1.BASE.

Note: Each field name in the following table must include the prefix WORKEXT_.

Table 62. 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 337 for a description of the control block.
4	4	ARGTABLE	The address of the arguments for the program. The arguments are arranged as a vector of address/length pairs followed by X'FFFFFFFFFFFFFFFF'. See "Format of Argument List" on page 338 for a description of the argument list.
8	4	FLAGS	A fullword of bits that ARXEXEC uses as flags. See "The ARXEXEC Routine" on page 334 for details.
12	4	INSTBLK	The address of the in-storage control block (INSTBLK). See "The In-Storage Control Block (INSTBLK)" on page 339 for a description of the control block.
16	4	CPPLPTR	This field is reserved.
20	4	EVALBLOCK	The address of the evaluation block (EVALBLOCK). See "The Evaluation Block (EVALBLOCK)" on page 341 for a description of the control block.
24	4	WORKAREA	The address of an 8-byte field that defines a work area for the ARXEXEC routine. See Table 14 on page 335 for more information about the work area.
28	4	USERFIELD	The address of the user field that is passed to ARXEXEC if you explicitly called ARXEXEC. You pass the address of the user field in parameter 8 (see "The ARXEXEC Routine" on page 334 for information about the parameters). You can use this field for your own processing. None of the REXX services use this field.

Table 62. Format of the Work Block Extension (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
32	4	RTPROC	**This field is a product-sensitive programming interface**. A fullword that is available for a REXX compiler runtime processor to use. This field lets a compiler runtime processor have an *anchor* that is unique for each compiled REXX program that runs within a language processor environment. A compiler runtime processor can use this field for its own purpose. The language processor does not check or change this field.
36	4	SOURCE_ADDRESS	The address of the PARSE SOURCE string for the program currently processing. This is the string that the PARSE SOURCE instruction would return.
40	4	SOURCE_LENGTH	The length of the PARSE SOURCE string to which the SOURCE_ADDRESS field at offset +36 (decimal) points.
44	4	--	This field is reserved.

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 63 on page 419 shows the format of the vector of external entry points. A mapping macro for the vector, ARXEXTE, is in PRD1.BASE.

The vector allows you to easily access the address of a particular REXX/VSE 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
ARXINIT	ARXINT
ARXLOAD	ARXLD
ARXSUBCM	ARXSUB
ARXEXEC	ARXEX
ARXINOUT	ARXIO
ARXJCL	ARXJCL (same)
ARXRLT	ARXRLT (same)
ARXSTK	ARXSTK (same)
ARXTERM	ARXTRM
ARXIC	ARXIC (same)
ARXUID	ARXUID (same)

Primary Entry Point Name	Alternate Entry Point Name
ARXTERMA	ARXTMA
ARXMSGID	ARXMID
ARXEXCOM	ARXEXC
ARXSAY	ARXSAY (same)
ARXERS	ARXERS (same)
ARXHST	ARXHST (same)
ARXHLT	ARXHLT (same)
ARXTXT	ARXTXT (same)
ARXLIN	ARXLIN (same)
ARXRTE	ARXRTE (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 provided for the language processor environment. If a user did not provide a replaceable routine, the address points to the default routine REXX/VSE supplies. The second address points to the default REXX/VSE routine. Chapter 21, “Replaceable Routines and Exits,” on page 439 describes replaceable routines in detail.

Note:

1. For compatibility with MVS, you can use IRX instead of ARX for the first three characters of field names in the following table that begin with ARX.
2. The ENTRY_COUNT field must include the prefix IRXEXTE_.

Table 63. 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	ARXINIT	The address of the initialization routine, ARXINIT.
8	4	LOAD_ROUTINE	The address of the user-supplied exec load replaceable routine for the language processor environment. This is the routine that the EXROUT field of the module name table specifies. If EXROUT does not specify a replaceable routine, the address points to the exec load routine that REXX/VSE supplies, ARXLOAD.
12	4	ARXLOAD	The address of the exec load routine REXX/VSE supplies, ARXLOAD.
16	4	ARXEXCOM	The address of the variable pool access interface, ARXEXCOM.
20	4	ARXEXEC	The address of the exec processing routine, ARXEXEC.

Table 63. 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 IO_ROUTINE does not specify a replaceable routine, the address points to the I/O routine REXX/VSE supplies, ARXINOUT.
28	4	ARXINOUT	The address of the I/O routine REXX/VSE supplies, ARXINOUT.
32	4	ARXJCL	The address of the ARXJCL routine.
36	4	ARXRLT	The address of the ARXRLT (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 the STACKRT field of the module name table specifies. If STACKRT does not specify a replaceable routine, the address points to the data stack routine REXX/VSE supplies, ARXSTK.
44	4	ARXSTK	The address of the data stack handling routine REXX/VSE supplies, ARXSTK.
48	4	ARXSUBCM	The address of the host command environment routine, ARXSUBCM.
52	4	ARXTERM	The address of the termination routine, ARXTERM.
56	4	ARXIC	The address of the trace and execution control routine, ARXIC.
60	4	MSGID_ROUTINE	The address of the user-supplied message ID replaceable routine for the language processor environment. This is the routine that the MSGIDRT field of the module name table specifies. If MSGID_ROUTINE does not specify a replaceable routine, the address points to the message ID routine REXX/VSE supplies, ARXMSGID.
64	4	ARXMSGID	The address of the message ID routine REXX/VSE supplies, ARXMSGID.
68	4	USERID_ROUTINE	The address of the user-supplied user ID replaceable routine for the language processor environment. This is the routine that the IDROUT field of the module name table specifies. If USERID_ROUTINE does not specify a replaceable routine, the address points to the user ID routine REXX/VSE supplies, ARXUID.

Table 63. Format of REXX Vector of External Entry Points (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
72	4	ARXUID	The address of the user ID routine REXX/VSE supplies, ARXUID.
76	4	ARXTERMA	The address of the termination routine, ARXTERMA.
80	4	ARXSAY	The address of the SAY instruction routine, ARXSAY.
84	4	ARXERS	**This field is a product-sensitive programming interface**. The address of the external routine search routine, ARXERS. The ARXERS routine is a REXX compiler programming routine. See "External Routine Search Routine (ARXERS)" on page 517 for a description.
88	4	ARXHST	**This field is a product-sensitive programming interface**. The address of the host command search routine, ARXHST. ARXHST is a REXX compiler programming routine. See "Host Command Search Routine (ARXHST)" on page 521 for a description.
92	4	ARXHLT	The address of the halt condition routine, ARXHLT.
96	4	ARXTXT	The address of the text retrieval routine, ARXTXT.
100	4	ARXLIN	The address of the LINESIZE built-in function routine, ARXLIN.
104	4	ARXRTE	**This field is a product-sensitive programming interface**. The address of the exit routing routine, ARXRTE. ARXRTE is a REXX compiler programming routine. See "Exit Routing Routine (ARXRTE)" on page 523 for a description.

Changing the Maximum Number of Environments in a Partition

Within a partition, 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 partition. There is a maximum number of environments that can be initialized at one time in a partition. The maximum is not a set number of environments. It depends on the number of chains of environments and the number of environments on each chain. The default maximum should be sufficient for any partition. However, if ARXINIT is initializing a new environment and this exceeds the maximum, ARXINIT 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.

The maximum number of environments REXX/VSE can initialize in a partition is defined in an environment table known as ARXANCHR. To change the number of environment table entries, you can use the ARXANCHR.Z sample in PRD1.BASE or you can create your own ARXANCHR phase.

If you create your own ARXANCHR phase, you must assemble the code and then link-edit the module as non-SVA eligible. You can place the phase in PRD2.CONFIG or in a sublibrary with type PHASE that precedes PRD1.BASE in the active PHASE chain. The phase cannot be in the SVA.

Table 64 on page 422 describes the environment table. A mapping macro for the environment table, ARXENVT, is in PRD1.BASE.

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.

Note: Each field name in the following table must include the prefix ENVTABLE_.

Table 64. Format of the Environment Table

Offset (Decimal)	Number of Bytes	Field Name	Description
0	8	ID	An 8-character field that identifies the environment table. The field contains the characters 'ARXANCHR'.
8	4	VERSION	The version of the environment table. The value must be 0001 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

The data stack is a repository for storing data for use by a REXX program. You can place elements on the data stack using the PUSH and QUEUE instructions, and take elements off of the data stack using the PULL instruction. You can also use REXX/VSE 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, “REXX/VSE Commands,” on page 143 describes the REXX commands for manipulating the data stack. *REXX/VSE User's Guide*, SC33-6641, 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 programs that run within a specific language processor environment.

A data stack may or may not be available to REXX programs that run in a particular language processor environment. Whether or not a data stack is available depends on the setting of the NOSTKFL flag (see page “NOSTKFL ” on page 395). When ARXINIT initializes an environment and the NOSTKFL flag is on, ARXINIT does not create a data stack or make a data stack available to the language processor environment. Programs that run in the environment cannot use a data stack.

If the NOSTKFL flag is off, either ARXINIT 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 ARXINIT initializes a new data stack for the new environment depends on:

- The setting of the NEWSTKFL (new data stack) flag, and
- Whether the environment is the first environment that ARXINIT is initializing on a chain.

Note: The NOSTKFL flag takes precedence over the NEWSTKFL flag. If the NOSTKFL flag is on, ARXINIT 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, ARXINIT automatically initializes a new data stack, regardless of the setting of the NEWSTKFL flag.

If the environment is not the first one on the chain, ARXINIT determines the setting of the NEWSTKFL flag. If the NEWSTKFL flag is off, ARXINIT 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, ARXINIT creates a new data stack for the language processor environment. Any REXX programs that run in the new environment can access only the new data stack for this environment. Programs cannot access any data stacks that ARXINIT created for any parent environment on the chain.

Environments can share only data stacks that environments higher on the chain initialized.

If ARXJCL calls ARXINIT to create a data stack when initializing an environment, REXX/VSE deletes the data stack when that environment is terminated. This occurs regardless of whether any elements are on the data stack. All elements on the data stack are lost.

Note: If you use the JCL EXEC command to call a REXX program, and the exit return code of the REXX program is zero when it is done, the language processor passes the data stack to Job Control before terminating the environment.

Figure 28 on page 423 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.

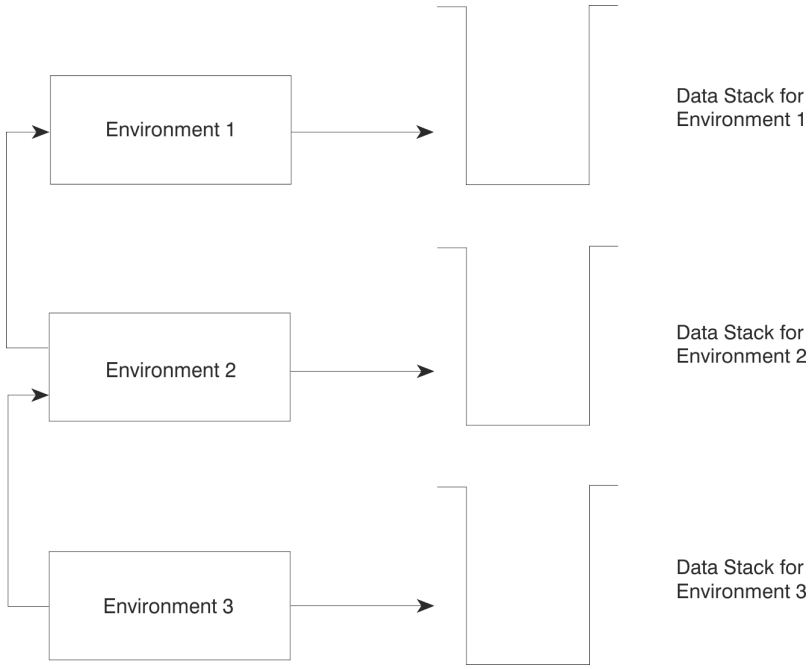


Figure 28. Separate Data Stacks for Each Environment

Environment 1 was the first environment on the chain. Therefore, REXX/VSE automatically created a data stack for environment 1. Any REXX programs that run 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 a different stack than for any of the other environments. A program runs in the most current environment (environment 3) and has access only to the data stack for environment 3.

Figure 29 on page 424 shows two environments that are initialized on one chain. The two environments share one data stack.

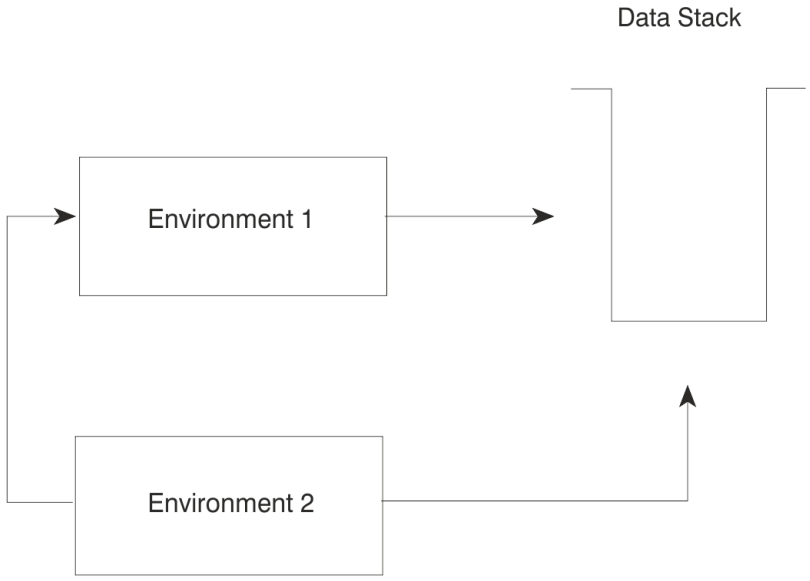


Figure 29. Sharing of the Data Stack between Environments

Environment 1 was the first environment on the chain. Therefore, REXX/VSE automatically created a data stack. The NEWSTKFL flag was off for initialization of environment 2. This indicates that a new data stack should not be created. Environment 2 shares the data stack that was created for environment 1. Any REXX programs 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 30 on page 425 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.

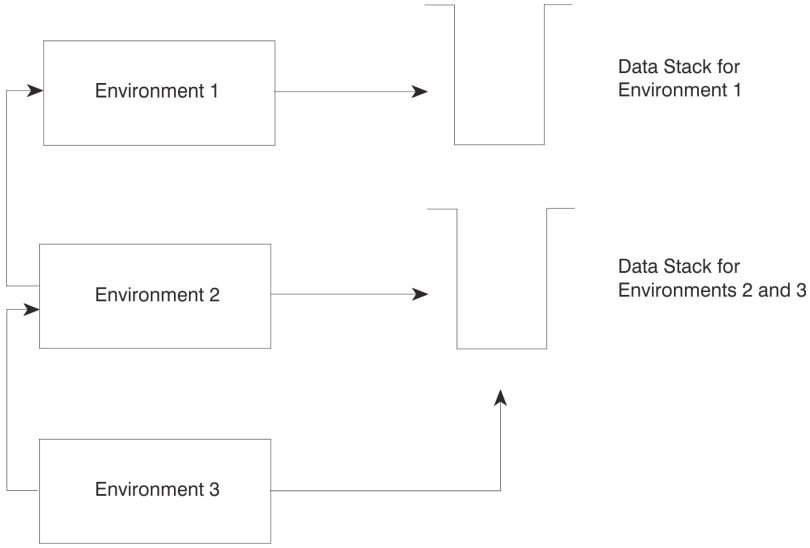


Figure 30. Separate Data Stack and Sharing of a Data Stack

Environments can be created without having a data stack, that is, the NOSTKFL flag is on. Referring to Figure 30 on page 425, 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 programs 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 program may need to use a data stack that is not shared with any other programs that are executing in different language processor environments. The NEWSTACK command creates a new data stack and 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 programs that are running in the first environment. The program in environment 2 may need to access its own data stack that is private. In the program, 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 NEWSTACK created 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 31 on page 426 shows two language processor environments 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 a program 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 programs executing in environment 2.

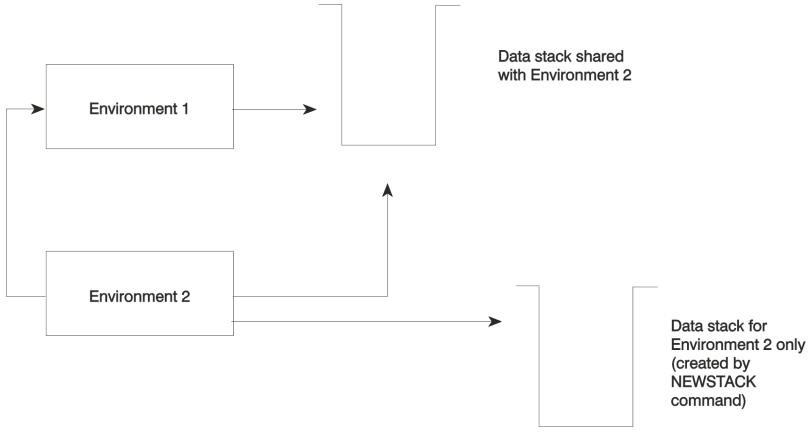


Figure 31. Creating a New Data Stack with the NEWSTACK Command

If a program executing in environment 1 issues the NEWSTACK command to create a secondary data stack, the secondary data stack is available only to REXX programs that execute in environment 1. Any programs that execute in environment 2 cannot access the new data stack created for environment 1.

You can use the DELSTACK command to delete any secondary data stacks that NEWSTACK created. When the secondary data stack is no longer required, the program can issue DELSTACK to delete the secondary stack. At this point, the primary data stack that is shared with environment 1 is accessible.

Several other commands perform data stack functions. For example, the QSTACK command finds out the number of data stacks that exist for the language processor environment. Chapter 10, “REXX/VSE Commands,” on page 143 describes stack-oriented commands, such as NEWSTACK and DELSTACK.

Chapter 20. Initialization and Termination Routines

This topic provides information about how to use the initialization routine, ARXINIT, and the termination routine, ARXTERM. ARXINIT, the initialization routine, initializes a language processor environment or obtains the address of the environment block for the current non-reentrant environment. ARXTERM, the termination routine, terminates a language processor environment. Chapter 8, “Using REXX,” on page 135 provides general information about how the initialization and termination of environments relates to REXX processing. Chapter 19, “Language Processor Environments,” on page 387 describes the concept of a language processor environment in detail. This includes the various characteristics you can specify when initializing an environment, the default parameters module, and information about the environment block and its format.

Language processor environments are created when they are needed. They are terminated when they are no longer needed, that is, when the job step is done.

Initialization Routine – ARXINIT

Use ARXINIT to initialize a new language processor environment or to obtain the address of the environment block for the current non-reentrant environment.

Note: To permit FORTRAN programs to call ARXINIT, there is an alternate entry point for the ARXINIT routine. The alternate entry point name is ARXINT.

If you use ARXINIT to obtain the address of the current environment block, ARXINIT returns the address in register 0 and also in the sixth parameter.

If you use ARXINIT 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, ARXINIT uses the value from the previous environment.

ARXINIT always locates a previous environment as follows. On the call to ARXINIT, you can pass the address of an environment block in register 0. ARXINIT 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, ARXINIT locates the previous environment. If ARXINIT locates a previous environment, ARXINIT uses that environment as the previous environment. If ARXINIT cannot locate an environment, ARXINIT uses the phase ARXPARMS as the previous environment. (“Values ARXINIT Uses to Initialize Environments” on page 409 describes in detail how ARXINIT locates a previous environment.)

A previous environment is always identified regardless of the parameters you specify on the call to ARXINIT.

Using ARXINIT, you can initialize a reentrant or a non-reentrant environment, as determined by the setting of the RENTRANT flag bit. If you use ARXINIT 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 ARXINIT to locate a previous environment, you can locate only the current non-reentrant environment. ARXINIT does not locate a reentrant environment.

Entry Specifications

For the ARXINIT initialization routine, the contents of the registers on entry are:

Register 0
Address of the current environment block (optional)

Register 1
Address of the parameter list the caller passes

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. To indicate the end of the parameter list, set the high-order bit of the last address to 1. If ARXINIT does not find the high-order bit set on in either the address for parameter 7 or the address for parameter 8 or 9, ARXINIT does not initialize the environment but returns with a return code of 20 and a reason code of 27. See "Output Parameters" for more information.

Table 65 on page 428 describes the parameters for ARXINIT. For general information about passing parameters, see “Parameter Lists for REXX/VSE Routines” on page 325.

Table 65. Parameters for ARXINIT

Parameter	Number of Bytes	Description
Parameter 1	8	The function ARXINIT is to perform, which can be: **INITENVB** To initialize a new environment. **FINDENVB** To obtain the address of the environment block for the current non-reentrant environment. ARXINIT does not initialize a new environment. ARXINIT returns the address of the environment block in register 0 and in parameter 6.
Parameter 2	8	The name of a parameters module that contains the values for initializing the new environment. (“Parameters Module and In-Storage Parameter List” on page 432 describes the module.) If the name of the parameters module is blank, ARXINIT assumes that all fields in the parameters module are null. ARXINIT provides two ways to pass parameter values: the parameters module and the address of an in-storage parameter list, which is parameter 3. “How ARXINIT Determines What Values to Use for the Environment” on page 432 describes how ARXINIT computes each parameter value and the flexibility of passing parameters.

Table 65. Parameters for ARXINIT (continued)

Parameter	Number of Bytes	Description
Parameter 3	4	The address of the *in-storage parameter list*, which is an area in storage containing parameters equivalent to those 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 432 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`, ARXINIT assumes that all fields in the in-storage parameter list are null.
Parameter 4	4	The address of a user field. ARXINIT does not use or check this pointer or the field. You can use this field for your own processing.
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	The address of the environment block. ARXINIT uses this parameter for output only. 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, ARXINIT returns the address of the environment block for the newly created environment in parameter 6. For either FINDENVB or INITENVB, ARXINIT 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.
Parameter 7	4	ARXINIT uses this parameter for output only. ARXINIT returns a reason code that indicates why processing was unsuccessful. Table 67 on page 435 describes the reason codes that ARXINIT returns.
Parameter 8	4	This parameter is optional. It lets you specify how REXX obtains storage in the language processor environment. Specify `0` if you want REXX/VSE to reserve a default amount of storage work area. If you want to pass a storage work area to ARXINIT, specify the address of an *extended parameter list*. The extended parameter list consists of the address (a fullword) of the storage work area and the length (a fullword) of the work area, followed by X'FFFFFFFFFFFFFFFF'. For more information about parameter 8 and storage, see "Specifying How REXX Obtains Storage in the Environment" on page 430. Although parameter 8 is optional, it is recommended that you specify an address of `0` if you do not want to pass a storage work area to ARXINIT.

Table 65. Parameters for ARXINIT (continued)

Parameter	Number of Bytes	Description
Parameter 9	4	This parameter is for output only, and it is optional. ARXINIT uses this for the return code. If you use this parameter, ARXINIT places the return code in the parameter and also in register 15. Otherwise, ARXINIT uses register 15 only. If the parameter list is incorrect, the return code is only in register 15. "Return Codes" describes the return codes.

Specifying How REXX Obtains Storage in the Environment

On the call to ARXINIT, parameter 8 is optional. You can use it to specify how REXX obtains storage in the language processor environment for the processing of REXX programs. If you specify 0 for parameter 8, during the initialization of the environment, REXX/VSE reserves a default amount of storage for the storage work area. If you have provided your own storage management replaceable routine, REXX/VSE calls your routine to obtain this storage work area. Otherwise, REXX/VSE obtains storage using GETVIS. When the environment that ARXINIT is initializing is terminated, REXX/VSE automatically frees the storage. REXX/VSE frees the storage by either calling your storage management replaceable routine or using FREEVIS, depending on how the storage was obtained.

You can also pass a storage work area to ARXINIT. 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 work area 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 32 on page 431 shows the extended parameter list.

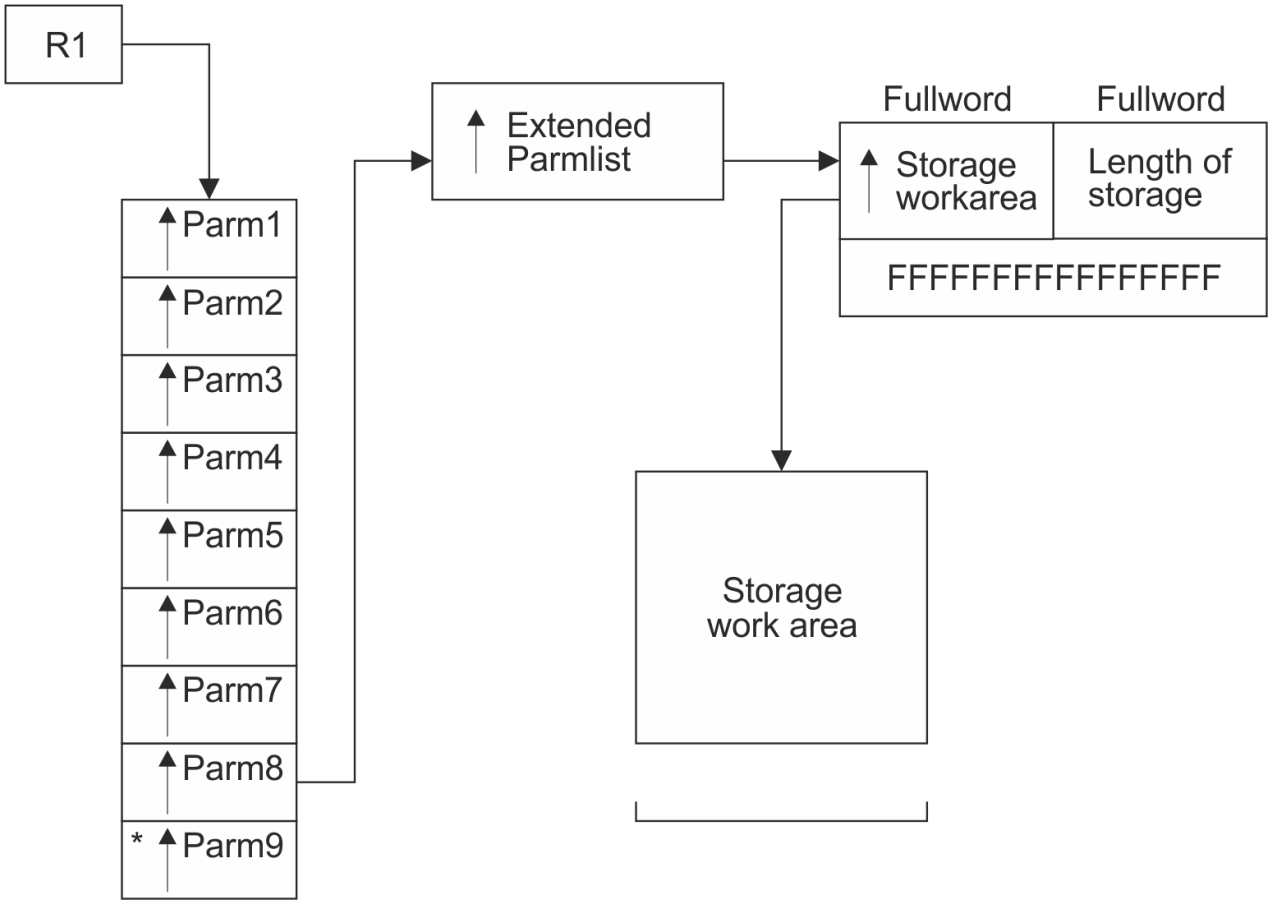


Figure 32. Extended Parameter List – Parameter 8

The storage work area you pass to ARXINIT is then available for REXX processing in the environment that you are initializing. The storage work area must remain available to the environment until the environment is terminated. After you terminate the language processor environment, you must also free the storage work area. REXX/VSE does not free the storage you pass to ARXINIT when you terminate the environment.

You can also specify that a reserved storage work area should not be initialized for the environment. REXX/VSE then obtains and frees storage whenever storage is required. To specify that a storage work area should not be initialized, for parameter 8, specify the address of the extended parameter list as previously described. In the extended parameter list, specify `0` for the address of the storage work area and `0` for the length of the storage work area. Again, X'FFFFFFFFFFFFFFFF' must follow the address/length pair to indicate the end of the extended parameter list. Specifying that REXX should run without a reserved storage work area is not recommended because of possible performance degradation. However, this option may be useful 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 work area and `-1` for the length of the work area. This is considered a null entry and ARXINIT ignores the extended parameter list entry. This is equivalent to specifying an address of `0` for parameter 8, and REXX/VSE reserves a default amount of work area storage.

In general, 3 pages (12K) of storage are needed for the storage work area for regular program processing, for each level of program nesting. If there is insufficient storage available in the storage work area, REXX calls the storage management routine to obtain additional storage if you provided a storage management replaceable routine. Otherwise, REXX/VSE uses GETVIS and FREEVIS to obtain and free storage. For more information about the replaceable routine, see “Storage Management Routine” on page 463.

How ARXINIT Determines What Values to Use for the Environment

ARXINIT first determines the values to use to initialize the environment. After all of the values are determined, ARXINIT initializes the new environment using the values.

On the call to ARXINIT, you can pass parameters that define the environment in two ways. You can specify the name of a parameters module (a phase) that contains the values ARXINIT 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 ARXINIT. 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 ARXINIT computes the values to use to initialize the environment, ARXINIT takes values from four sources using the following hierarchical search order:

1. The in-storage list of parameters that you pass on the call to ARXINIT.

   If you pass an in-storage parameter list and the value in the list is not null, ARXINIT uses this value. Otherwise, ARXINIT continues.
2. The parameters module whose name you pass on the call to ARXINIT.

   If you pass a parameters module and the value in the module is not null, ARXINIT uses this value. Otherwise, ARXINIT continues.
3. The previous language processor environment.

   ARXINIT copies the value from the previous environment.
4. The ARXPARMS parameters module if a previous environment does not exist.

If a parameter has a null value, ARXINIT continues to search until it finds a non-null value. The following types of parameters are defined to be null:

- A character string containing only blanks or having a length of 0
- An address if its value is 0
- A binary number with the value X'80000000'
- A bit setting with a corresponding mask of 0.

On the call to ARXINIT, 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, ARXINIT 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 phase that contains the values you want ARXINIT to use to initialize a new language processor environment. The default parameters module for initializing environments is ARXPARMS. “Characteristics of a Language Processor Environment” on page 390 describes the parameters module.

On the call to the ARXINIT, you can optionally pass the name of a parameters module that you have created. The parameters module contains the values you want ARXINIT 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 66 on page 433 shows the format of a parameters module and in-storage list. The format of the parameters module is identical to the default module. “Characteristics of a Language Processor

Environment" on page 390 describes the parameters module and each field in detail. Indicate the end of the table with X'FFFFFFFFFFFFFFFF'.

Table 66. 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 0001.
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 files or devices for reading and writing data, 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 programs 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.
44	4	SUBPOOL	This field is reserved.
48	8	ADDRSPN	Name of the partition (VSE).
56	8	—	The end of the parameter block must be X'FFFFFFFFFFFFFFFF'.

Specifying Values for the New Environment

For more information about parameters, see "Specifying Values for Different Environments" on page 413.

When you call ARXINIT, you cannot specify the ID and VERSION. If you pass values for the ID or VERSION parameters, ARXINIT ignores the value and uses the default.

At offset +36 in the parameters module, the field is a fullword of bits that ARXINIT uses as flags. The flags define certain characteristics for the new language processor environment and how the environment and programs running in the environment operate. The parameter following the flags is a mask field that works 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. ARXINIT uses the mask field to determine whether it should use or ignore the corresponding flag bit.

See page “MASKS ” on page 393 for details about the bit settings for the mask field. Table 52 on page 393 summarizes each flag. “Flags and Corresponding Masks” on page 393 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 — ARXINIT ignores the corresponding bit in the flags field (that is, ARXINIT considers the bit to be null)
- 1 – ARXINIT uses the corresponding bit in the flags field.

When you call ARXINIT, the flag settings that ARXINIT 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 ARXPARMS if a previous environment does not exist.

ARXINIT uses the following order to determine what value to use for **each** flag bit:

- ARXINIT first checks the mask setting in the in-storage parameter list. If the mask is 1, ARXINIT uses the flag value from the in-storage parameter list.
- If the mask in the in-storage parameter list is 0, ARXINIT then checks the mask setting in the parameters module. If the mask in the parameters module is 1, ARXINIT uses the flag value from the parameters module.
- If the mask in the parameters module is 0, ARXINIT uses the flag value defined for the previous environment.
- If a previous environment does not exist, ARXINIT uses the flag setting from ARXPARMS.

For detailed information about the parameters you can specify for initializing a language processor environment, see “Specifying Values for Different Environments” on page 413.

Indicate the end of the parameter block with X'FFFFFFFFFFFFFFFF'.

Return Specifications: For the ARXINIT initialization routine, the contents of the registers on return are:

Register 0
Contains the address of the new environment block if ARXINIT initialized a new environment, or the address of the environment block for the current non-reentrant environment that ARXINIT located.

If you call ARXINIT to initialize a new REXX environment and this is successful, register 0 and Parameter 6 contain the address of the new environment block (ENVBLOCK). Otherwise, register 0 is restored, and Parameter 6 contains 0.

If you call ARXINIT to find the current non-reentrant REXX environment and this is successful, then register 0 and Parameter 6 contain the address of the current, non-reentrant environment block (ENVBLOCK). Otherwise, register 0 and Parameter 6 contain 0.

Register 1
Address of the parameter list.

ARXINIT uses three parameters (parameters 6, 7, and 9) for output only (see Table 65 on page 428). "Output Parameters" describes the three output parameters.

Registers 2-14
Same as on entry

Register 15
Return code

Output Parameters: The parameter list for ARXINIT contains three parameters that ARXINIT uses for output only (parameters 6, 7, and 9). Parameter 6 contains the address of the environment block. If you called ARXINIT to locate an environment, parameter 6 contains the address of the environment block for the current non-reentrant environment. If you called ARXINIT 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, ARXINIT returns the return code in parameter 9 and also in register 15.

Parameter 7 contains a reason code for ARXINIT processing. The reason code indicates whether or not ARXINIT completed successfully. If ARXINIT processing was not successful, the reason code indicates the error. Table 67 on page 435 describes the reason codes ARXINIT returns.

Table 67. Reason Codes for ARXINIT Processing

Reason Code	Description
0	Successful processing.
1	Unsuccessful processing. The type of function (Parameter 1) was not valid. Valid functions are INITENVB and FINDENVB.
2	Unsuccessful processing. Attempted to use TSOFL flag.
3	Reserved
4	Reserved
5	Unsuccessful processing. The value specified in the MODNAMET_GETFREER field in the module name table does not match the MODNAMET_GETFREER value in the current REXX 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 entry length field in the table. See “Host Command Environment Table” on page 401 for information about the table.
7	Reserved
8	Reserved
9	Reserved
10	Unsuccessful processing. The ARXINITX exit routine returned a nonzero return code. ARXINIT stops initialization.
11	Reserved
12	Unsuccessful processing. REXX/VSE initialization was unsuccessful. The ARXITMV exit routine returned a nonzero return code. ARXINIT stops initialization.
13	Unsuccessful processing. The REXX I/O routine or the replaceable I/O routine is called to initialize I/O when ARXINIT is initializing a new language processor environment. The I/O routine returned a nonzero return code.
14	Unsuccessful processing. The REXX data stack routine or the replaceable data stack routine is called to initialize the data stack when ARXINIT is initializing a new language processor environment. The data stack routine returned a nonzero return code.
15	Unsuccessful processing. The REXX exec load routine or the replaceable exec load routine is called to initialize exec loading when ARXINIT is initializing a new language processor environment. The exec load routine returned a nonzero return code.
16	Reserved
17	Reserved

Table 67. Reason Codes for ARXINIT Processing (continued)

Reason Code	Description
20	Unsuccessful processing. Storage could not be obtained.
21	Unsuccessful processing. A module could not be loaded into storage.
22	Unsuccessful processing. A lock could not be obtained.
23	Reserved
24	Unsuccessful processing. The environment table (ENVTABLE) is full. The maximum number of environments has already been initialized. See "Changing the Maximum Number of Environments in a Partition" on page 421 for more information about the environment table.
25	Unsuccessful processing. The extended parameter list (parameter 8) passed to ARXINIT is incorrect. The end of the extended parameter list must be indicated with X'FFFFFFFFFFFFFFFF'.
26	Unsuccessful processing. The values specified in the extended parameter list (parameter 8) are incorrect. Either the address or the length of the storage work area (but not both) was 0, or the length was negative. Reason code 26 is not returned if: • Both the address and length of the storage work area are 0, which are valid values. • The address of the storage work area is 0 and the length is -1, which is considered a valid null entry.
27	Unsuccessful processing. An incorrect number of parameters was passed to ARXINIT. Setting on the high-order bit in parameter 7 or in optional parameters 8 or 9 marks the end of the parameter list. ARXINIT returns reason code 27 if it cannot find the high-order bit on in the last address of the parameter list. ARXINIT does not return reason code 27 if the caller passes fewer than seven parameters (that is, sets on the high-order bit in a parameter prior to parameter 7). If ARXINIT 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, ARXINIT returns only a return code of 20 in register 15 to indicate an error.
28	Unsuccessful processing. Attempted use of SPSHARE flag.
29	Unsuccessful processing. Attempted use of NOLOADDD flag.
30	Unsuccessful processing. REXX is not installed in the SVA.
31	Reserved

Return Codes: ARXINIT returns different return codes for finding an environment and for initializing an environment. ARXINIT returns the return code in register 15. If you specify the return code parameter (parameter 9), ARXINIT also returns the return code in the parameter.

Table 68 on page 436 shows the return codes if you call ARXINIT to find an environment (FINDENVB function).

Table 68. ARXINIT Return Codes for Finding an Environment (FINDENVB)

Return Code	Description
0	Processing was successful. ARXINIT located and initialized the current non-reentrant REXX environment under the current task.
4	Processing was successful. ARXINIT located and initialized the current non-reentrant REXX environment under a previous task.

Table 68. ARXINIT Return Codes for Finding an Environment (FINDENVB) (continued)

Return Code	Description
20	Processing was not successful. An error occurred. Check the reason code that ARXINIT returns in parameter 7.
28	Processing was successful. There is no current non-reentrant REXX environment.

Table 69 on page 437 shows the return codes if you call ARXINIT to initialize an environment (INITENVB function).

Table 69. ARXINIT Return Codes for Initializing an Environment (INITENVB)

Return Code	Description
0	Processing was successful. ARXINIT initialized a new language processor environment. The new environment is not the first REXX environment under the current task.
4	Processing was successful. ARXINIT initialized a new REXX language processor environment, which is the first environment under the current task.
20	Processing was not successful. An error occurred. Check the reason code that ARXINIT returns in the parameter list.

Termination Routine – ARXTERM

When an application is done with a language processor environment, the application is responsible for terminating the language processor environment. Only the application that called ARXINIT to create the language processor environment should terminate the language processor environment.

The ARXTERM routine terminates a language processor environment.

Note: To permit FORTRAN programs to call ARXTERM, there is an alternate entry point for the ARXTERM routine. The alternate entry point name is ARXTRM.

Note: Another way to terminate a language processor environment is calling ARXTERMA, which is for abnormal terminations but works in the general case as well. ARXTERMA terminates language processor environments that still contain active programs; ARXTERM does not do this.

In register 0, you can optionally pass the address of the environment block (ENVBLOCK) for the environment you want to terminate. ARXTERM then terminates the language processor environment register 0 points to. The environment must have been initialized on the current task.

If you do not specify an environment block address in register 0, ARXTERM locates the last environment that was created under the current task and terminates that environment.

When ARXTERM terminates the environment, ARXTERM closes all open members and files that were opened under that environment. ARXTERM also deletes any data stacks that you created under the environment using the NEWSTACK command.

ARXTERM does not terminate an environment under any one of the following conditions:

- The environment was not initialized under the current task
- An active program 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 ARXTERM to terminate

the first environment and other environments on the task still exist, ARXTERM does not terminate the environment and returns with a return code of 20.

Entry Specifications: For the ARXTERM termination routine, the contents of the registers on entry are:

Register 0
Address of an environment block (ENVBLOCK). (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 ARXTERM.

Return Specifications: For the ARXTERM termination routine, the contents of the registers on return are:

Register 0
If you pass the address of an environment block and ARXTERM terminates the environment, ARXTERM returns the address of the environment block for the previous environment or 0 if there is no previous environment. If you do not pass an address, register 0 contains the same value as on entry.

Registers 1-14
Same as on entry

Register 15
Return code

Return Codes: Table 70 on page 438 shows the return codes for the ARXTERM routine.

Table 70. Return Codes for ARXTERM

Return Code	Description
0	ARXTERM successfully terminated the environment. The terminated environment was not the last REXX environment on the task.
4	ARXTERM successfully terminated the environment. The terminated environment was the last REXX environment on the task.
20	ARXTERM could not terminate the environment.
28	The environment could not be found.

Chapter 21. Replaceable Routines and Exits

When a REXX program runs, various system services obtain and free storage, handle data stack requests, load and free the program, and perform I/O. REXX/VSE provides routines for these system services. The routines are called *replaceable routines* because you can provide your own routines that replace the REXX/VSE routines.

Besides defining your own replaceable routines to replace the routines that REXX/VSE provides, you can use the interfaces as described in this chapter to call any of the supplied routines to perform system services. 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 to perform that particular service.

REXX/VSE also provides several exits you can use to customize REXX processing. The exits let you customize the initialization and termination of language processor environments and exec processing itself.

This topic describes each of the replaceable routines and the exits.

Replaceable Routines: If you replace the REXX/VSE-supplied routine, your routine can perform some pre-processing and then call the REXX/VSE routine to actually perform the service request. If the replaceable routine you provide calls the REXX/VSE routine, your replaceable routine must act as a *filter* between the call to your routine and its call to the REXX/VSE 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 without calling the REXX/VSE routine.

The following summarizes the routines you can replace and the functions your routine must perform, if you replace the REXX/VSE routine. “Replaceable Routines” on page 440 describes each routine in more detail.

Exec Load
: Called to load a program into storage and free a program when the program completes processing. The exec load routine is also called to determine whether a program is currently loaded and to close a specified member.

I/O
: Called to read a record from or write a record to a specified file. The I/O routine is also called to open or close a specified file. For example, the routine is called for the SAY and PULL instructions 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) accompanies 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 REXX/VSE 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. You can provide your own ARXPARMS parameters module that ARXINIT uses instead of the default ARXPARMS module. In your module, specify the names of your replaceable routines. You can also call ARXINIT to initialize an environment and pass the name of your module name table that includes the names of your replaceable routines.

  "Changing the Default Values for Initializing an Environment" on page 412 describes how to provide your own parameters module. "Initialization Routine – ARXINIT" on page 427 describes ARXINIT.

You can also call any of the REXX/VSE replaceable routines from a program to perform a system service. You can also write your own routine to perform a service. This topic describes the interfaces to the REXX/VSE routines.

Exit Routines: You can use several exits to customize REXX processing. Some exits have fixed names. Others do not have fixed names. You name the exit yourself and then specify the name in the module name table. The following briefly describes exits. "REXX Exit Routines" on page 468 describes each exit in more detail.

- Pre-environment initialization – customizes processing before the ARXINIT initialization routine initializes a language processor environment.
- Post-environment initialization – customizes processing after the ARXINIT initialization routine has initialized an environment, but before ARXINIT completes processing.
- Environment termination – customizes processing when a language processor environment is terminated.
- Exec Initialization—customizes processing after the variable pool has been created and before the program begins processing.
- Exec Termination—customizes processing after a program completes processing and before the variable pool is deleted.
- Exec Processing— customizes exec processing before a program is loaded and runs.
- RXHLT – raises the halt condition (see "REXX Exit Data Areas and Parameters" on page 473).

See "REXX Exit Routines" on page 468 for more information about the exits.

Replaceable Routines

The following topics describe each of the REXX/VSE replaceable routines. The documentation describes how the REXX/VSE routines work, the input they receive, and the output they return. If you provide your own routine that replaces the REXX/VSE routine, your routine must handle all of the functions that the REXX/VSE routine handles.

The replaceable routines are programming routines that you can call from a program. The only requirement for calling one of the REXX/VSE routines is that a language processor environment must exist in which the routine runs.

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 a program before calling ARXEXEC to call the REXX program. Similarly, if you write your own routine, an application program can call your routine as long as a language processor environment exists in which the routine can run.

You could also write your own routine that application programs can call to perform a system service and have your routine call the REXX/VSE routine. Your routine could act as a *filter* between the call to your routine and its call to the REXX/VSE routine. For example, you could write your own exec load routine that verifies a request and then calls the REXX/VSE exec load routine to actually load the program.

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 REXX/VSE routine. Your routine can perform pre-processing, such as checking or changing the request or parameters, and then call the

corresponding REXX/VSE routine. If your routine calls the REXX/VSE routine to actually perform the request, your routine must call the system routine in 31 bit addressing mode also.

- When REXX/VSE calls your replaceable routine, your routine can use any of the REXX/VSE replaceable routines to request system services.
- The addresses of the REXX/VSE-supplied replaceable routines and any replaceable routines you provide are stored in the REXX vector of external entry points (see page "Format of the REXX Vector of External Entry Points" on page 418). This allows a caller external to REXX to call any of the replaceable routines, either your routines or the supplied ones. For example, if you want to preload a REXX program in storage before using the ARXEXEC routine to call the program, you can call the ARXLOAD routine to load the program. ARXLOAD is the supplied exec load routine. If you provide your own exec load routine, you can also use your routine to preload the program.
- When REXX/VSE or an application program calls a replaceable routine, 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 441.

Using the Environment Block Address

If you provide your own routine to replace a supplied one, when REXX/VSE calls your routine, it passes the address of the environment block for the current environment in register 0. If your routine then calls the supplied one, it is recommended that you pass the environment block address you received to the supplied one. When you call the 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 routine passes the environment block address in the parameter list, the supplied routine uses the address you specify and ignores register 0. The supplied routine does not validate the address you pass. Ensure that your routine passes the same address it received in register 0 when it got control.

If your routine does not specify an address in the environment block address parameter or the replaceable routine does not support the parameter, the supplied routine checks register 0 for the environment block address. If register 0 contains the address of a valid environment block, the supplied routine runs in that environment. If the address in register 0 is not valid, the supplied routine locates and runs in the current non-reentrant environment.

If your routine does not pass the environment block address it received to the supplied routine, the 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 recommended that you pass the environment block address when your routine calls the supplied routine.

An application program can call a supplied replaceable routine or one that you provide 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 call replaceable routines:

- If an application program calls a supplied replaceable routine and does not pass an environment block address, the supplied routine locates the current non-reentrant environment and runs in that environment.
- If an application program calls a routine you provide, either the application program must provide the environment block address or your 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 phase. You can link-edit all your replaceable routines in a separate sublibrary or in an existing library that contains other routines. The routines can reside in a phase in a sublibrary in the active PHASE chain.

The replaceable routines must be reentrant, refreshable, and reusable. The characteristics for the routines are:

- State: Problem program
- Not authorized
- AMODE(31), RMODE(ANY)

Exec Load Routine

REXX/VSE calls the exec load routine to load and free REXX programs and:

- To close any input files from which programs are loaded
- To check whether a program is currently loaded in storage
- When a language processor environment is initialized and terminated.

The name of the supplied exec load routine is ARXLOAD.

Note: To permit FORTRAN programs to call ARXLOAD, REXX/VSE provides an alternate entry point for the ARXLOAD routine. The alternate entry point name is ARXLD.

When the exec load routine is called to load a program, the routine reads the program from the member of a sublibrary in the active PROC chain and then places the program into a data structure called the *in-storage control block* (INSTBLK). "The In-Storage Control Block" on page 445 describes the format of the in-storage control block. When the exec load routine is called to free a program, the program frees the storage that the previously loaded program 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 398 describes the format of the module name table.

REXX/VSE calls the exec load routine when:

- A language processor environment is initialized. During environment initialization, the exec load routine initializes the REXX program load environment.
- The ARXEXEC routine is called and the program is not preloaded. See "The ARXEXEC Routine" on page 334 for information about using ARXEXEC.
- The program that is currently running calls an external function or subroutine and the function or subroutine is a REXX program. (This is an internal call to the ARXEXEC routine.)
- A program that was loaded needs to be freed.
- The language processor environment that originally opened the member of the sublibrary from which programs are loaded is terminating and all files associated with the environment must be closed.
- You use the EXEC command (page "EXEC" on page 145) to run a REXX program.

The supplied load routine, ARXLOAD, tests for numbered records in the file. If the records of a file are numbered, the routine removes the numbers when it loads the program. A record is considered to be numbered if the last 8 characters of the first record are numeric.

If the first record of the file is not numbered, the routine loads the program without making any changes.

Any user-written program can call ARXLOAD to perform the functions that ARXLOAD supports. You can also write your own exec load routine and call the routine from an application program. For example, if you have an application program that calls the ARXEXEC routine to run a REXX program, you may want to preload the program into storage before calling ARXEXEC. To preload the program, your application program can call ARXLOAD. The program can also call your own exec load routine.

If you are writing an exec load routine that will be used in environments in which compiled REXX programs run, note that your exec load routine may want to call a compiler interface load routine. For information about the compiler interface load routine and when it can be called, see Chapter 24, "Support for the Library for REXX/370 in REXX/VSE," on page 499.

Entry Specifications: For the exec load replaceable routine, the contents of the registers on entry are described in the following. You can specify the address of the environment block 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 441.

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 5-word parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. To indicate the end of the parameter list, set the high-order bit of the last address to 1. For more information about passing parameters, see “Parameter Lists for REXX/VSE Routines” on page 325.

Table 71 on page 443 describes the parameters for the exec load routine.

Table 71. 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 end of the field with blanks. The valid functions are: • INIT • LOAD • FREE • STATUS • CLOSEDD • TERM. The functions are described in "Functions You Can Specify...".
Parameter 2	4	Specifies the address of the exec block (EXECBLK). The exec block is a control block that describes the program to load (LOAD) or check (STATUS) or the member to close (CLOSEDD). “The Exec Block” on page 445 describes the exec block. For the LOAD, STATUS, and CLOSEDD functions, this parameter must contain a valid exec block address. For the other functions, this parameter is ignored.

Table 71. 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 program in storage. The in-storage control block contains pointers to each record in the program and the length of each record. “The In-Storage Control Block” on page 445 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, 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 program to be freed. As an output parameter for the FREE function, the parameter contains a `0` indicating the program was freed. If the program could not be freed, the return code in register 15 or the return code parameter (parameter 5) or both indicate the error condition. "Return Codes" describes the return codes. As an output parameter for the LOAD or STATUS functions, the parameter returns the address of the in-storage control block that represents the program that was: • Just loaded (LOAD function) • Previously loaded (STATUS function). For the LOAD and STATUS functions, the routine returns a value of `0` if the program is not loaded.
Parameter 4	4	This parameter is optional. It is the address of the environment block that represents the environment in which you want the exec load replaceable routine to run. If you specify a nonzero value, 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, ensure the address you specify is correct or unpredictable results can occur. For more information, see “Using the Environment Block Address” on page 441.
Parameter 5	4	This parameter is optional. It is a field that the exec load replaceable routine uses to return the return code. 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 incorrect, the return code is returned in register 15 only. "Return Codes" describes the return codes.

Functions You Can Specify for Parameter 1: The functions you can specify in parameter 1 are:

INIT
The routine performs any initialization that is required. During the initialization of a language processor environment, REXX/VSE calls the exec load routine to initialize load processing.

LOAD
The routine loads the specified program in the exec block from the member of a sublibrary specified in the exec block. “The Exec Block” on page 445 describes the exec block.

The routine returns the address of the in-storage control block (parameter 3) that represents the loaded program. “The In-Storage Control Block” on page 445 shows the format of the in-storage control block.

Note: The ARXLOAD routine reuses an existing copy of a previously loaded program if it appears that the program has not changed. However, if the CLOSEXFL flag is on, indicating the member should be closed after each program is loaded, ARXLOAD does not reuse a previously loaded program. Instead, a new copy of the program is read into storage for each load request. For more information about the CLOSEXFL flag, see page “CLOSEXFL ” on page 396.

FREE
The routine frees the program represented by the in-storage control block to which parameter 3 points.

Note: If a user-written load routine calls ARXLOAD to load a program, the user-written load routine must also call ARXLOAD to free the program.

STATUS
The routine determines whether the program specified in the exec block is currently loaded in storage from the member of a sublibrary specified in the exec block. If the program 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 program into storage.

TERM
The routine performs any cleanup prior to termination of the language processor environment. When the last language processor environment under the task that originally opened the member terminates, all files associated with the environment are closed. When ARXLOAD is terminating the last language processor environment under a task, it frees any programs that were loaded by any language processor environment under the task but were not yet freed.

CLOSEDD
The routine closes the member specified in the exec block. It does not free any programs that have been loaded.

The Exec Block

The exec block (EXECBLK) is a control block that describes the:

- Member to load (LOAD function)
- Member to check (STATUS function)
- Member to close (CLOSEDD function).

If a user-written program calls ARXLOAD or your own exec load routine, the program must build the exec block and pass the address of the exec block on the call. REXX/VSE provides a mapping macro, ARXEXECB, for the exec block. The mapping macro is in PRD1.BASE. See Table 15 on page 337 for the format of the exec block.

The In-Storage Control Block

The in-storage control block defines the structure of a program in storage. It contains pointers to each record in the program and the length of each record.

Table 17 on page 339 shows the format of the in-storage control block. Table 18 on page 340 shows the format of the vector of records.

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 72 on page 446 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 72. Return Codes for the Exec Load Replaceable Routine

Return Code	Description
-3	The program could not be located. The program is not loaded.
0	Processing was successful. The requested function completed.
4	The specified program 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 MEMBER was required but not specified (LOAD, STATUS, and CLOSEDD functions). • The MEMBER was specified, but a LIBDEF specifying the sublibrary did not precede the attempt to load the member. • An error occurred during processing. REXX/VSE 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 incorrect. It contains too few or too many parameters, or the high-order bit of the last address is not 1 to indicate the end of the parameter list.

Input/Output Routine

REXX/VSE has two kinds of input and output:

- REXX/VSE runs in batch only, so console input and output consists of line mode input from and output to the default input and output streams.
- EXECIO commands read and write data on disk.

ARXINOUT, the input/output (I/O) replaceable routine, is also called the read input/write output data routine.

The input/output replaceable routine operates on the following types of files:

- Sublibrary members of any type (use the fully-qualified name)
- SYSIPT or SYSLST
- SAM files. (Only SAM files on disk are supported. You need to use DLBL to associate a SAM file with a file name.)

The default input/output routine operates only on:

- SYSIPT
- SYSLST
- SYS*xxx* (where xxx is numeric)
- Any other 7-character name.

REXX/VSE calls the I/O routine to:

- Read a record
- Write a record
- Open a file

- Close a file. (You can open a file in a user routine and change the DSIB_LRECL field.)

Note: To permit FORTRAN programs to call ARXINOUT, REXX/VSE provides an alternate entry point for the ARXINOUT routine. The alternate entry point name is ARXIO.

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 ARXDSIB mapping macro to map this area. The mapping macro is in PRD1.BASE.

Specify the name of the I/O routine in the IOROUT field in the module name table. "Module Name Table" on page 398 describes the format of the module name table. I/O processing is based on the Librarian and SAM access methods.

The I/O routine is called for:

- Initialization. When ARXINIT initializes a language processor environment, REXX/VSE calls the I/O replaceable routine to initialize I/O processing.
- Open:
  - When you use the LINESIZE built-in function in a program
  - Before the language processor does any input or output.
- For input, when:
  - A PULL or a PARSE PULL instruction is processed, and the data stack is empty
  - A PARSE EXTERNAL instruction is processed
  - Input during pauses in interactive debug is processed
  - The EXECIO command reads data on disk
  - A program outside of REXX calls the I/O replaceable routine for input of a record.
- For output, when:
  - A SAY instruction is processed
  - Error messages must be written
  - Trace (interactive debug facility) messages must be written
  - The EXECIO command writes data to disk
  - A program outside of REXX calls the I/O replaceable routine for output of a record.
- Termination. When REXX/VSE terminates a language processor environment, the I/O replaceable routine is called to clean up I/O.

Entry Specifications: This section describes the contents of the registers on entry for the I/O replaceable routine. You can specify the address of the environment block either in register 0 or in the environment block address parameter in the parameter list. For more information, see "Using the Environment Block Address" on page 441.

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. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for REXX/VSE Routines" on page 325.

Table 73 on page 448 describes the parameters for the I/O routine.

Table 73. Input Parameters for the I/O Replaceable Routine

Parameter	Number of Bytes	Description
Parameter 1	8	The function to perform. The function name is left justified, in uppercase, and padded to the right with blanks. Valid functions are: • CLOSE • INIT • OPENR • OPENW • OPENX • READ • READX • TERM • WRITE. "Functions Supported for the I/O Routine" on page 450 describes these functions.
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 page "Data Set Information Block (DSIB)" on page 453). This field is not used as an input parameter for the CLOSE, INIT, READ, READX, or TERM functions.
Parameter 3	4	Specifies the length of the data in the buffer to which parameter 2 points. On output for an open request, parameter 3 may contain the length of the data set information block. "Buffer and Buffer Length Parameters" on page 451 describes the buffer and buffer length in more detail.
Parameter 4	8	This is the name of a SAM file or SYSIPT or SYSLST. (If you are using a sublibrary member, this parameter must be blank; see Parameter 8. The name must be: • SYSIPT • SYSLST • SYS*xxx* (where xxx is numeric) • Any other 7-character name. Otherwise, you receive an error.) The name is uppercase, left justified, and padded with blanks on the right. For READ, READX, and WRITE, it is the name of file from which to read or to which to write the data. For CLOSE, it is the name of the file to close. For OPEN, OPENX, and OPENW, it is the name of the file to open. INIT and TERM functions do not use this field. If the input or output file is not a SAM file or SYSIPT or SYSLST, and if parameter 8 is also blank, the return code from the I/O routine is 20.

Table 73. Input Parameters for the I/O Replaceable Routine (continued)

Parameter	Number of Bytes	Description
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 file that is opened for update, it provides a record number to verify the number of the record to update. Specify 0 to bypass verification of the record number. This parameter is not used for the CLOSE, INIT, OPENR, OPENW, OPENX, or TERM functions. See “Line Number Parameter” on page 452 for more information.
Parameter 6	4	This parameter is optional. It is the address of the environment block that represents the environment in which you want the I/O replaceable routine to run. If you specify a nonzero 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, ensure the address you specify is correct or unpredictable results can occur. For more information, see “Using the Environment Block Address” on page 441.
Parameter 7	4	This parameter is optional. It is a field that the I/O replaceable routine uses for the return code. If you use this parameter, the I/O routine puts the return code in the parameter and also in register 15. Otherwise, the routine uses register 15 only. If the parameter list is incorrect, the return code is returned in register 15 only. "Return Codes" describes the return codes.
Parameter 8	34	This parameter is optional. It is a fully qualified sublibrary name (7 characters for the library name, 8 for the sublibrary, 8 for the member name, 8 for the type). The name is uppercase, left justified, and padded with blanks on the right. For READ, READX, and WRITE, it is the name of file from which to read or to which to write the data. For CLOSE, it is the name of the file to close. For OPEN, OPENX, and OPENW, it is the name of the file to open. INIT and TERM functions do not use this field. If this parameter and parameter 4 are both blank, the return code form the I/O routine is 20.

Table 73. Input Parameters for the I/O Replaceable Routine (continued)

Parameter	Number of Bytes	Description
Parameter 9	4	This parameter is optional. It specifies the address of the control block for ARXINOUT. (Table 74 on page 452 shows the control block.) To explicitly or implicitly open a SAM file, you must specify Parameter 9. For other types of files, if you do not specify this parameter, the language processor uses default information. The control block is used for input and output with all types of files. For a member of a sublibrary, the control block is needed only for writing information. It contains an indication of whether the sublibrary member contains SYSIPT data. The default for a new file indicates no SYSIPT data. For an old file, the default is the same as specified on opening the original file. For SYSLST and SYSIPT, the control block is for reading or writing information. It contains information about block size, record format, and record size. It also contains carriage control information for SYSLST. The defaults for SYSLST are: block size of 120, record format of FIXUNB, record size of 121, and no carriage control data. The defaults for SYSIPT are: block size of 80, record format of FIXUNB, record size of 80.

Functions Supported for the I/O Routine

Parameter 1 specifies the function the I/O routine performs. Valid functions are:

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 file for a read operation if it is not already open. Parameter 4 or parameter 8 specifies the file name.

The I/O routine returns the address of the data set information block in parameter 2. “Data Set Information Block (DSIB)” on page 453 describes the block in more detail.

OPENW
The routine opens the specified file for a write operation if it is not already open. Parameter 4 or parameter 8 specifies the file name.

The I/O routine returns the address of the data set information block in parameter 2. “Data Set Information Block (DSIB)” on page 453 describes the block in more detail.

OPENX
The routine opens the specified file for update if it is not already open. Parameter 4 or parameter 8 specifies the file name.

The I/O routine returns the address of the data set information block in parameter 2. “Data Set Information Block (DSIB)” on page 453 describes the block in more detail.

READ
The routine reads data from the file that parameter 4 or parameter 8 specifies. It returns the data in the buffer to which the address in parameter 2 points. It also returns the number of the record that was read in the line number parameter (parameter 5).

READ and READX are equivalent, except that the file is opened differently. You can do subsequent read operations to the same file using either READ or READX because they do not reopen the file.

If the file to read from is closed, the routine opens it for input and then performs the read.

READX
The routine reads data from the file that parameter 4 or parameter 8 specifies. It returns the data in the buffer to which the address in parameter 2 points. It also returns the number of the record that was read in the line number parameter (parameter 5).

If the file to read from is closed, the routine opens it for update and then performs the read.

READ and READX are equivalent, except that the file is opened differently. You can do subsequent read operations to the same file using either READ or READX because they do not reopen the file.

WRITE
The routine writes data from the specified buffer to the specified file. The address in parameter 2 points to the buffer. Parameter 4 or parameter 8 specifies the file name.

If the file is closed, the routine first opens it for output and then writes the record. For a member of a sublibrary, the record is written at the end of the file. For SAM files residing in VSAM-managed space, a disposition of NEW indicates writing the first record, and a disposition of OLD indicates writing at the end of the file.

When a file is opened for update, the WRITE function rewrites the last record that READ or READX retrieved. 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 open files.

CLOSE
The routine closes the file that parameter 4 or parameter 8 specifies. The CLOSE function permits files to be freed.

CLOSE is allowed only from the task under which the file 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. Only the WRITE function uses these parameters for input. (CLOSE, INIT, OPENR, OPENX, OPENW, READ, READX, and TERM do not use these parameters for input.) READ, READX, OPENR, OPENX, and OPENW use parameter 2 for output, and the same functions plus WRITE use parameter 3 for output. (CLOSE, INIT, TERM, and WRITE do not use parameter 2 for output, and CLOSE, INIT, and TERM do not use parameter 3 for output.)

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. “Data Set Information Block (DSIB)” on page 453 describes the format of this area. REXX/VSE provides a mapping macro, ARXDSIB, 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 the flags field indicates (see "Data Set Information Block (DSIB)" on page 453).

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 this case, the I/O buffers are freed and the storage is made available to REXX/VSE.

Any replaceable I/O routine must conform to this procedure to ensure that the program 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 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 formats all of its output lines to the width the LRECL field specifies. If your routine specifies a LRECL (DSIB_LRECL field) of 0, the language processor formats its output using a width of 80 characters, the default.

When the I/O routine is called with the TERM function, all buffers are freed.

Line Number Parameter

Parameter 5, the line number parameter, is an input parameter for the WRITE function and an output parameter for the READ and READX functions. (It is not used for input for the CLOSE, INIT, OPENR, OPENW, OPENX, READ, READX, or TERM functions. It is not used for output for CLOSE, INIT, OPENR, OPENX, OPENW, or TERM.)

If you are writing to a file that is opened for update, you can use this parameter to verify the record being updated. The parameter must be either:

- A nonzero 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 file 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.

I/O Control Block

You can use the I/O control block to identify the file you want to read or to which you want to write. You can use this control block only in the I/O replaceable routine (ARXINOUT). A mapping macro for the I/O control block, ARXIOPTS, is in PRD1.BASE. The following table shows the I/O control block.

Note: Each field name in the table must include the prefix IOPTS_.

Table 74. I/O Control Block

Offset (Decimal)	Number of Bytes	Field Name	Description
0	8	ID	An 8-character string that identifies the information block. It contains the characters ARXIOPTS.
8	2	LENGTH	Length of the control block.

Table 74. I/O Control Block (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
10	2	LIB_OPTS	Library option flags. Only the first 4 bits are used. The flag bits are: • LIB_DATA - SYSIPT data • LIB_NODATA - no SYSIPT data • LIB_FORMAT_F - fixed record format • LIB_FORMAT_S - string record format
12	4	DTF_BLKSIZE	The block size of the file.
16	4	DTF_RECSIZE	The record size of the file.
24	8	DTF_RECFORM	The record format of the file. This is one of the following: • 'FIXUNB ' - fixed unblocked • 'FIXBLK ' - fixed blocked • 'VARUNB ' - variable unblocked • 'VARBLK ' - variable blocked.
32	2	DTF_FLAGS	SAM options flag. Only the first 4 bits are used. The flag bits are: • DTF_BLKSIZE_FLAG block size passed • DTF_RECSIZE_FLAG record size passed • DTF_RECFORM_FLAG record format passed • DTF_CC_FLAG carriage control • DTF_ASA_FLAG carriage control • DTF_MCC_FLAG carriage control
34	2	(reserved)	none
36	4	LIB_BYTES	Part length if library members in string record format are to be read or written in smaller pieces.

Data Set Information Block (DSIB)

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 in parameter 2. REXX/VSE provides a mapping macro ARXDSIB you can use to map the block. The mapping macro is in PRD1.BASE.

Table 75 on page 453 shows the format of the control block.

Note: Each field name in the following table must include the prefix DSIB_.

Table 75. Format of the Data Set Information Block

Offset (Decimal)	Number of Bytes	Field Name	Description
0	8	ID	An 8-character string that identifies the information block. It contains the characters ARXDSIB.

Table 75. Format of the Data Set Information Block (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
8	2	LENGTH	The length of the data set information block.
10	2	---	Reserved.
12	8	DDNAME	An 8-character string that specifies the name of the file for which REXX/VSE returns information. This is the file that the I/O routine opened. This is blank for a member of a sublibrary.
20	4	FLAGS	A fullword of bits that are used as flags. Only the first 8 bits are used. The remaining bits are reserved. The flag bits indicate whether or not information is returned in the fields at offset +26 through offset +56. 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 follows the table.
24	2	---	Reserved.
26	2	BLKSZ	The block size (BLKSIZE) of the file.
28	2	DSORG	The organization of the file: • '0800' - This is a Librarian file. • '4000' - This is a SAM file.
30	2	RECFM	The record format (RECFM) of the file. • 'F ' - Fixed • 'FB' - Fixed blocked • 'V ' - Variable • 'VB' - Variable blocked.
32	4	GET_CNT	The total number of records read.
36	4	PUT_CNT	The total number of records written.
40	1	IO_MODE	The mode in which the file was opened. • 'R' - open for READ • 'X' - open for READX • 'W' - open for WRITE • 'L' - open for exec load.
41	1	CC	Carriage control information. • 'A' - ANSI carriage control • 'M' - machine carriage control • ' ' - no carriage control.
42	1	TRC	Reserved.
43	1	---	Reserved.

Table 75. Format of the Data Set Information Block (continued)

Offset (Decimal)	Number of Bytes	Field Name	Description
44	12	---	Reserved.
56	4	LRECL	The logical record length (LRECL) of the file. 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.
60	20	---	Reserved.
80	34	LIBRNAME	A string that specifies the name of sublibrary member for which REXX/VSE returns information. This is the member that the I/O routine opened. This is blank for a SAM file, SYSIPT, or SYSLST.

At offset +20 in the data set information block, there is a fullword of bits that are used as flags. Only the first eight bits are used. The remaining bits are reserved. The bits indicate whether information is returned in each field in the control block starting at offset +26. A bit must be on if its corresponding field is returning a value. If the bit is 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 +56. The logical record length is the only file attribute that is required. The remaining 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 file 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 file was opened at offset +40.
- The CC flag. This bit must be set on if you are returning carriage control information at offset +41.

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 76 on page 456 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 76. 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 file 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 ARXDSIB mapping macro to map this area.
4	Processing was successful. For a READ, READX, or WRITE, the file was opened. For an OPENR, OPENW, or OPENX, the file 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 ARXDSIB 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 file 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 ARXDSIB 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 you did not specify a file name. An error message that describes the error is also issued.
24	Processing was not successful. 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. It contains too few or too many parameters, or the high-order bit of the last address is not 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 page “The VSE Host Command Environment” on page 25 for the definition of host commands). A REXX program 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 a program, 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 ROUTINE field of the host command environment table specifies the names of the routines that are called for each host command environment. (“Host Command Environment Table” on page 401 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 you can provide a routine that processes the commands for that environment. "Host Command Environment Table" on page 401 describes the table.

Entry Specifications: For a host command environment routine, the contents of the registers on entry are described in the following. For more information about register 0, see "Using the Environment Block Address" on page 441.

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. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. (Table 77 on page 457 describes the parameters for a host command environment replaceable routine.)

Table 77. 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 401 describes the user token field.

Table 77. 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 program that called 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, programs 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 REXX/VSE 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 a command that is not valid, it is recommended that you return X'FFFFFFFD' as the return code. This means the REXX special variable RC contains -3.

Note: If a host command processor abnormally terminates, the entire batch job abends. For information about what happened to your job, see the description of $ABEND in the *VSE/ESA System Control Statements*, SC33-6713.

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. (The return code is also contained in parameter 5.)

Return Codes: Table 78 on page 458 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 129 for information about ERROR and FAILURE conditions and condition traps.

Table 78. Return Codes for the Host Command Environment Routine

Return Code	Description
≤-13	If the value of the return code is -13 or less than -13, the routine requested turning on the HOSTFAIL flag. This is a TRACE NEGATIVE condition and a FAILURE condition is trapped in the program.

Table 78. Return Codes for the Host Command Environment Routine (continued)

Return Code	Description
-1 — -12	If the value of the return code is from -1 to -12 inclusive, the routine requested turning on the HOSTERR flag. This is a TRACE ERROR condition and an ERROR condition is trapped in the program.
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 turning on the HOSTERR flag. This is a TRACE ERROR condition and an ERROR condition is trapped in the program.
≥13	If the value of the return code is 13 or greater than 13, the routine requested turning on the HOSTFAIL flag. This is a TRACE NEGATIVE condition and a FAILURE condition is trapped in the program.

Data Stack Routine

The data stack routine is called to handle any requests for data stack services. The routine is called when a program 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.

The name of the data stack routine that REXX/VSE supplies is ARXSTK. 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 routine REXX/VSE supplies, ARXSTK. If your routine handles the data stack requests without calling the routine REXX/VSE supplies, your routine must manipulate its own data stack.

If your data stack routine performs pre-processing and then calls ARXSTK, your routine must pass the address of the environment block for the language processor environment to ARXSTK.

An application can call ARXSTK 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.

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 is not created and any subsequent data stack functions operate as if the data stack is permanently empty.

The maximum string that can be placed on the data stack is 1 byte less than 16 megabytes. 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 programs 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" on page 422 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 398 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 in the following. You can specify the address of the environment block 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 441.

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. To indicate the end of the parameter list, set the high-order bit of the last address to 1. For more information about passing parameters, see "Parameter Lists for REXX/VSE Routines" on page 325.

Table 79 on page 460 describes the parameters for the data stack routine.

Table 79. 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. Valid functions are: `PUSH QUEUE MAKEBUF NEWSTACK QSTACK QELEM PULL PULLEXTR QUEUED DROPBUF DELSTACK QBUF` "Functions Supported for the Data Stack Routine" on page 461 describes the functions in more detail.

Table 79. Parameters for the Data Stack Routine (continued)

Parameter	Number of Bytes	Description
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 current input stream. It is recommended that you do not change the original string and that you copy the original string into your own dynamic storage. Also, the original string may no longer be valid when another data stack operation is performed.
Parameter 3	4	The length of the string to which the address in parameter 2 points. This is 0 if there is no string or element. The maximum length is 16 million. A string longer than 16 million characters is truncated to 16 million with no error indication.
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 that follow and the individual descriptions of the data stack commands in this book.
Parameter 5	4	This parameter is optional. It is the address of the environment block that represents the environment in which you want the data stack replaceable routine to run. If you specify a nonzero 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, ensure the address you specify is correct or unpredictable results can occur. For more information, see “Using the Environment Block Address” on page 441.
Parameter 6	4	This parameter is optional. It is a field that the data stack replaceable routine uses to return the return code. 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 incorrect, the return code is returned in register 15 only. "Return Codes" describes the return codes.

Functions Supported for the Data Stack Routine

Parameter 1 contains the name of the function that the data stack routine is to perform. The functions operate on the currently active data stack. Valid functions are:

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 current input stream. ASSGN(STDIN) returns the name of the current input stream.

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. (For more information, see PARSE EXTERNAL on page "PARSE EXTERNAL " on page 44.)

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 you can use MAKEBUF 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 you do not specify *n*, the last buffer that was created and all subsequently added elements are deleted.

For example, if MAKEBUF is issued six times (that is, the last return code from the MAKEBUF function is 6), and

```
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 cannot 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 program.

QBUF
Returns the number of buffers on the active data stack. If the data stack contains no buffers, `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.

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 80 on page 463 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 REXX/VSE commands that are issued, such as NEWSTACK, DELSTACK, or QBUF. The command's return code is placed into the REXX special variable RC, which the program 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 80. 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 only for the PULL function.
20	Processing was not successful. An error condition occurred. The requested function is not performed. You may have specified a function name that is incorrect. An error message describing the error may be issued.
28	Processing was not successful. A language processor environment could not be located.
32	Processing was not successful. The parameter list is incorrect. It contains too few or too many parameters, or the high-order bit of the last address is not 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 call your storage management routine for a storage pool. If you do not provide your own storage management routine, GETVIS and FREEVIS handle storage pool requests. The storage that GETVIS and FREEVIS obtain and free is accessible in only a single partition. No storage is shared between partitions. Providing your own storage management routine gives you an alternative to this.

The storage management routine is called to obtain or free a storage pool for REXX processing. The routine supplies a storage pool that the REXX storage routines manage.

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 398 describes the format of the module name table. Note that an application may replace this routine.

Entry Specifications: The following describes the contents of the registers on entry for the storage management replaceable routine. For more information about register 0, see “Using the Environment Block Address” on page 441.

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. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. Table 81 on page 464 describes the parameters for the storage management routine.

Table 81. 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 megabytes in virtual storage **GETLOW** Obtain a storage pool below 16 megabytes 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.
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 free. This is the size of the storage pool to which parameter 2 points. All requests for the FREE function are for a single storage pool that GET or GETLOW previously obtained. On output for the GET and GETLOW functions, the parameter specifies the size of the storage pool the routine obtained. This size must be at least the size that was requested in parameter 4. The REXX/VSE storage routines use the size returned in parameter 3.
Parameter 4	4	Specifies in bytes the size of the storage pool to obtain. This parameter is an input parameter for GET and GETLOW. 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.

Table 81. Parameters for the Storage Management Replaceable Routine (continued)

Parameter	Number of Bytes	Description
Parameter 5	4	This field is reserved.

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 82 on page 465 shows the return codes for the storage management routine.

Table 82. 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. REXX/VSE calls the user ID replaceable routine whenever the USERID built-in function is issued in a language processor environment. The name of the user ID routine REXX/VSE supplies is ARXUID.

The name of the user ID replaceable routine is specified in the IDROUT field in the module name table. "Module Name Table" on page 398 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 441.

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. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. For more information about passing parameters, see "Parameter Lists for REXX/VSE Routines" on page 325.

Table 83 on page 466 describes the parameters for the user ID routine.

Table 83. 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 only valid function is USERID. This must be in uppercase, left justified, and padded to the right with blanks. “Function Supported for the User ID Routine” on page 466 describes function 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 to which the address in parameter 2 points. 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 are unpredictable.
Parameter 4	4	This parameter is optional. It is the address of the environment block that represents the environment in which you want the user ID replaceable routine to run. If you specify a nonzero value, 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, ensure the address you specify is correct or unpredictable results can occur. For more information, see “Using the Environment Block Address” on page 441.
Parameter 5	4	This parameter is optional. It is a field the user ID replaceable routine uses to return the return code. 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 only register 15. If the parameter list is incorrect, the return code is returned only in register 15. "Return Codes" describes the return codes.

Function Supported for the User ID Routine

Specify the function the user ID routine is to perform in parameter 1. The only valid function is USERID.

USERID
Returns the same value that the USERID built-in function would return. The value returned can be:

1. the user ID from the last SETUID command
2. the user ID of the calling REXX program, if one REXX program calls another
3. the user ID under which the job is running
4. the job name.

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 84 on page 467 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 84. 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 incorrect. It contains too few or too many parameters, or the high-order bit of the last address is not 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 accompany an error message. The name of the message identifier routine that REXX/VSE supplies is ARXMSGID.

Note: To permit FORTRAN programs to call ARXMSGID, REXX/VSE provides an alternate entry point for the ARXMSGID routine. The alternate entry point name is ARXMID.

The routine is called whenever a message is to be written when a REXX program or REXX routine (for example, ARXEXCOM or ARXIC) is running.

The name of the message identifier replaceable routine is specified in the MSGIDRT field in the module name table. “Module Name Table” on page 398 describes the format of the module name table.

Entry Specifications: The following describes the contents of the registers on entry for the message identifier routine. For more information about register 0, see “Using the Environment Block Address” on page 441.

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 85 on page 468 shows the return codes for the message identifier routine.

Table 85. Return Codes for the Message Identifier Replaceable Routine

Return Code	Description
0	Include the message identifier (message ID) with the message.
Nonzero	Do not include the message identifier (message ID) with the message.

REXX Exit Routines

You can use exit routines to customize REXX processing.

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.

This topic describes the following types of exits:

- Exits for initialization and termination (ARXINITX, ARXITMV, and ARXTERMX)
- The SAA-defined halt exit (with Halt Test and Halt Clear functions)
- Installation-supplied exits (for exec processing, exec initialization, and exec termination).

Some exits receive parameters on entry, and others receive no parameters. Some of the REXX exits, such as the exits for initialization and termination, have fixed names. Others, such as the exec processing, exec initialization, exec termination, and halt exit, do not. You supply the name yourself and then define the name in the appropriate fields in the module name table.

Exits for Language Processor Environment Initialization and Termination

The supplied exits are default exits. There are three exits you can use to customize the initialization and termination of language processor environments. The names of these exits are fixed. If you do not wish to use the default exits, you can provide your own exits. If you provide one or more exits, the exit is called whenever the ARXINIT and ARXTERM routines are called. This occurs whenever a user explicitly calls ARXINIT and ARXTERM or when the system automatically calls the routines to initialize and terminate a language processor environment. See Chapter 20, “Initialization and Termination Routines,” on page 427 for a description of ARXINIT and ARXTERM and their parameters.

ARXINITX

This is the pre-environment initialization exit routine. You can use ARXINITX to:

- Prevent the initialization of a language processor environment
- Change parameters for initializing a language processor environment
- Perform special pre-environment processing.

By default, ARXINIT sets a return code of 0 and returns.

ARXINITX performs exit processing before a new language processor environment is initialized. ARXINIT calls ARXINITX. ARXINITX receives control before an environment is initialized and before ARXINIT evaluates any parameters. ARXINITX receives the same parameters that ARXINIT receives.

ARXINIT uses register 0 to locate the previous environment block, reentrant or non-reentrant. Therefore, changing register 0 controls how ARXINIT locates the previous environment block. If you change register 0 and do not restore it, REXX/VSE uses the new value to locate the previous environment block.

The following shows the contents of the registers on entry for the ARXINITX exit.

Register 0
Same as on entry to ARXINIT initialization routine (address of environment block)

Register 1
Address of the parameter list passed to ARXINIT

Registers 2–12
Unpredictable

Register 13
Address of register save area

Register 14
Return address

Register 15
Entry point address

The following table shows the parameters for ARXINITX.

Table 86. Parameters for ARXINITX

Parameter	Number of Bytes	Description
Parameter 1	8	This parameter specifies the function to be performed: **INITENVB** Initializes a new environment. **FINDENVB** Obtains the address of the environment block for the current non-reentrant environment. FINDENVB returns the address of the environment block in register 0 and in parameter 6. It does not initialize a new environment.
Parameter 2	8	The name of the parameters module, which contains the values for initializing the new environment. On the call to the ARXINIT initialization routine, the caller may have passed a blank in this field. Therefore, ARXINIT assumes that all the fields in the parameters module are null. ARXINIT 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.
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. This parameter may be `0`. If the address is `0`, ARXINIT assumes that all fields in the in-storage parameter list are null.
Parameter 4	4	The address of a user field. ARXINIT does not use or check this pointer or the field. You can use this field for your own processing.
Parameter 5	4	A 4-byte field that is reserved.

Table 86. Parameters for ARXINITX (continued)

Parameter	Number of Bytes	Description
Parameter 6	4	Only ARXINIT uses this parameter for output, and the exit should not alter this parameter. The parameter contains the address of the environment block. If you use the FINDENVB parameter to locate an environment, this parameter contains the address of the environment block for the current non-reentrant environment. If you use INITENVB to initialize a new environment, ARXINIT returns the address of the environment block for the newly created environment in this parameter. For either FINDENVB or INITENVB, ARXINIT also returns the address of the environment block in register 0. This parameter lets higher level languages obtain the environment block address in order to examine information in the environment block.
Parameter 7	4	Only ARXINIT uses this parameter for output, and the exit should not alter this parameter. In this field ARXINIT returns a reason code, which indicates why the requested function did not complete successfully.
Parameter 8	4	This is an optional parameter that lets you specify how REXX obtains storage in the language processor environment. Specify 0 if you want REXX/VSE to reserve a default amount of storage work area. If you want to pass a storage work area to ARXINIT, specify the address of an extended parameter list. The extended parameter list consists of a fullword that is the address of the storage work area and a fullword that is the length of the work area, followed by X'FFFFFFFFFFFFFFFF'.
Parameter 9	4	Only ARXINIT uses this parameter for output, and the exit should not alter this parameter. It is a 4-byte field that ARXINIT uses to return the return code.

The following shows return specifications from ARXINITX.

Register 0
Same values passed to the ARXINIT initialization routine

Registers 1-14
Same as on entry

Register 15
Return code

The following table shows return codes for ARXINITX.

Table 87. Return Codes for ARXINITX

Return Code	Meaning
0	Exit processing was successful. REXX processing continues.
Nonzero	Exit processing was not successful. REXX processing sets register 15 to 20 and terminates. (The program is not executed. REXX sends a message that indicates a failure in a system service.)

ARXITMV

ARXIMTV is the post-environment initialization exit routine. It performs exit processing after a language processor environment is initialized. You can use ARXITMV to perform special processing for a newly initialized language processor environment.

ARXINIT calls ARXITMV after the environment is initialized and after the control blocks, such as the environment block, are set up. By default, ARXITMV does not prevent the initialization of a language processor environment and does not perform any special initialization processing. It sets a return code of 0 and returns.

ARXITMV does not receive any parameters. ARXITMV has the same return codes as ARXINITX; see Table 87 on page 470.

The following shows entry specifications for ARXITMV.

Register 0
Address of new environment block

Registers 1-12
Unpredictable

Register 13
Address of register save area

Register 14
Return address

Register 15
Exit entry point address

The following shows return specifications.

Register 0
Same as on entry

Registers 1–14
Same as on entry

Register 15
Return code

ARXTERMX

ARXTERMX is the environment termination exit routine. You can use ARXTERMX to prevent the termination of a language processor environment or to perform special termination processing for a language processor environment. By default, ARXTERMX sets a return code of 0 and returns.

ARXTERM calls ARXTERMX. ARXTERMX performs exit processing before a language processor environment is terminated. ARXTERMX does not receive any parameters. See the list that follows for entry specifications. ARXTERMX receives control before ARXTERM terminates the environment. ARXTERMX has the same return specifications as ARXITMV; see “ARXITMV” on page 470. ARXITMV has the same return codes as ARXINITX; see Table 87 on page 470.

The following shows entry specifications for ARXTERMX.

Register 0
Address of terminating environment block

Registers 1–12
Unpredictable

Register 13
Address of register save area

Register 14
Return address

Register 15
Exit entry point address

Installing ARXINITX, ARXITMV, and ARXTERMX

To install the ARXINITX, ARXITMV, or ARXTERMX exit, you need to link-edit the exit with the ARXINIT initialization routine. You cannot change the names of these default exit routines.

Halt Exit

The halt exit has two functions: test and clear. The halt test function is called at each clause boundary. The halt clear function is called when a halt condition is raised.

The following describes the contents of the registers on entry:

Register 0
The address of the language processor environment (ENVBLOCK) under which the program is running

Register 1
The address of a list that contains the addresses of the REXX exit parameters

Registers 2-12
Unpredictable

Register 13
The address of a 72-byte register save area

Register 14
The return address

Register 15
The exit entry point address.

The halt test exit parameter list consists of the following 6 fullword fields:

Table 88. Parameter List for Halt Exit

Parameter	Length	Type	Description
RXIT_EXIT	4	Supplied	Exit identifier code, a unique integer value in binary that identifies the exit. The value is 7.
RXIT_SUBFN	4	Supplied	Exit function subcode. The value is 1 for clear, 2 for test.
RXIT_ENVB@	4	Supplied	Address of the ENVBLOCK under which the REXX program was running when this exit was called.
RXIT_USER	4	Supplied	This is the ENVBLOCK_USER field that was established when ARXINIT created the environment block.
RXIT_EXITRC	4	Output	The exit must store its return code here.
RXHLT_FLAGS	4	Returned	This parameter is for halt test only. For the test function, having the first bit on raises halt. This is equivalent to setting this field to the decimal value 2147483648.

When you do not want to raise a halt condition, set RC and RXHLT_FLAGS to 0.

The following shows return specifications.

Register 1-14
Same as on entry

Register 15
Ignored

The halt exit may return one of the three following values as a return code in RXIT_EXITRC:

Table 89. Return Codes for Halt Exit

Return Code	Meaning
0	Successful handling of the service. The parameter has been updated as appropriate for the exit. The bit setting of the flags determines the action.
1	Exit chooses not to handle the service request. The language processor should handle the request by the default means. The continuation processing of the language processor is the same as would occur if this exit had not been specified.
-1	A severe error occurred while processing this request. REXX ends the program with error 48 (Failure in system service).

Any other value specified is not supported and is treated as a -1.

REXX Exit Data Areas and Parameters

The ARXXITDF macro assigns the correct integer values to the symbols identifying the REXX exit and the associated function subcodes. To include it in an assembler language program, call the macro with:

```
         ARXXITDF     Include symbols for REXX exits and subcodes
```

The macro declares the following symbol and subcodes:

```
*  RXHLT: Exit for HALT processing
*
RXHLT    EQU   7      Halt processing
RXHLTCLR EQU   1      ... Clear HALT indicator
RXHLTTST EQU   2      ... Test HALT indicator
```

Installing a Halt Exit

The halt exit routine must be a separate phase that ARXINIT loads before the start of processing of a REXX program. The name of this phase is in the module name table, field RXHLT.

Installation-Supplied Exits

There are default exit routines for exec initialization, exec termination, and exec processing. You can use these routines or provide and use your own exit routines. To do so, you specify the name you have chosen in the appropriate field of the module name table.

Exec Initialization and Termination Exits

You can use these exits to update and access REXX variables. ARXEXEC or a compiler runtime processor calls these exits. The exec initialization exit gets control after initialization of the REXX variable pool but before the processing of the first clause in the program. The exec termination exit is called after a REXX program has completed, but before the termination of the variable pool for the program.

The exec initialization and termination exits do not have fixed names. For initialization, specify the name you have chosen in the EXECINIT field in the module name table. For termination, specify the name you have chosen in the EXECTERM field in the module name table. You can do this by providing your own parameters module that replaces the default module. Or you can call ARXINIT to initialize a language processor environment and pass the module name table on the call.

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.

See “Changing the Default Values for Initializing an Environment” on page 412 for a description of how to provide your own parameters module. See Chapter 20, “Initialization and Termination Routines,” on page 427 for a description of ARXINIT.

The following shows entry specifications for both exits:

Register 0
Address of current environment block

Registers 1-12
Unpredictable

Register 13
Address of register save area

Register 14
Return address

Register 15
Exit entry point address

When an environment is initialized, REXX/VSE creates the environment block (ENVBLOCK) that contains pointers to several other control blocks. Together, these control blocks define all the characteristics of the environment. The address of the environment block is passed in register 0 in all calls to REXX exits and routines, and in all calls to the REXX compiler runtime processor and compiler interface routines. You can read only information from the environment block or the control blocks to which the environment block points. If you change the values, results are unpredictable.

The exec initialization and termination exits have the same return specifications as ARXITMV. See “ARXITMV” on page 470.

The following table shows return codes.

Table 90. Return Codes

Return Code	Meaning
0	Exit processing was successful. REXX processing continues.
Nonzero	Exit processing was not successful. The program is not run. REXX issues a message that indicates a failure in a system service.

Exec Processing (ARXEXEC) Exit Routine

You can use an exec processing exit to prevent the running of a REXX program or to perform special processing before a REXX program runs.

The ARXEXEC routine calls an exec processing exit. (If you provide an exec processing exit, it is called whenever the ARXEXEC routine is called to invoke a REXX program. You can explicitly call ARXEXEC or REXX/VSE can call ARXEXEC to invoke a program. REXX/VSE always calls ARXEXEC to handle exec processing. For example, if you run a REXX program using the EXEC command, the ARXEXEC routine is called to invoke the program.) The exec processing exit gets control before the program is loaded, if the program was not pre-loaded, and before ARXEXEC evaluates any parameters on the call.

The exec processing exit does not have a fixed name. Specify the name of the exit in the IRXEXECX field in the module name table. You can do this by providing your own parameters module that replaces the default module. Or you can call ARXINIT to initialize a language processor environment and pass the module name table on the call.

The exit is used on a language processor environment basis. You can provide an exec processing exit in any type of environment.

See “Changing the Default Values for Initializing an Environment” on page 412 for a description of how to provide your own parameters module. See Chapter 20, “Initialization and Termination Routines,” on page 427 for a description of ARXINIT.

The following shows entry specifications for the exec processing exit.

Register 0
Address of the current environment block

Register 1
Address of the parameter list passed to ARXEXEC

Registers 2–12
Unpredictable

Register 13
Address of register save area

Register 14
Return address

Register 15
Entry point address

The following table shows parameters for the exec processing exit.

Table 91. Parameters for Exec Processing Exit

Parameter	Number of Bytes	Description
Parameter 1	4	The address of the exec block (EXECBLK). The exec block is a control block that describes the program to load. It contains information needed to process the program, such as the member from which the program is to be loaded and the name of the initial host command environment when the program starts running. This parameter can be `0` if the program is pre-loaded and the address of the pre-loaded program is passed in parameter 4. If you specify both this parameter and parameter 4, the value in parameter 4 is used and this parameter is ignored.
Parameter 2	4	The address of the arguments for the program. The arguments are arranged as a vector of address/length pairs followed by X'FFFFFFFFFFFFFFFF'. There is no limit to the number of arguments passed to the program.

Table 91. Parameters for Exec Processing Exit (continued)

Parameter	Number of Bytes	Description
Parameter 3	4	A fullword of flag bits. Only the first 4 bits are used. The remaining bits are reserved. Bits 0, 1, and 2 are mutually exclusive. **Bit 0** If the bit is set on, the program was called as a *command*, that is, another program did not call it as an external function or subroutine. **Bit 1** If the bit is set on, the program was called as an external function (a function call). **Bit 2** If the bit is set on, the program was called as a subroutine. **Bit 3** Set this bit on if you want ARXEXEC to return extended return codes in the range 20001-20099. If a syntax error occurs, ARXEXEC 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, ARXEXEC 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, ARXEXEC returns with return code 0.
Parameter 4	4	The address of the in-storage control block (INSTBLK). The in-storage control block defines the structure of a pre-loaded program in storage. It contains pointers to each record in the program and the length of each record. This parameter is specified if the caller of the ARXEXEC routine has pre-loaded the program. Otherwise, this parameter is `0`.
Parameter 5	4	Reserved, must be `0`.
Parameter 6	4	The address of an evaluation block (EVALBLOCK). ARXEXEC uses the evaluation block to return the result from the program that was specified on either the RETURN or EXIT instruction. The value may be `0`, if the program does not return a result or the caller of ARXEXEC plans to use the ARXRLT (get result) routine to get the result or the result is to be ignored.
Parameter 7	4	The address of an 8-byte field that defines a work area. In the 8-byte field: • The first 4 bytes contain the address of the work area • The second 4 bytes contain the length of the work area. The work area is passed to the language processor to use for running the program. If the work area is too small, ARXEXEC returns with a return code of 20 and a message 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`. ARXEXEC obtains storage for its work area or calls the replaceable storage routine specified in the GETFREER field (in the module name table) for the environment, if you provided a storage routine.

Table 91. Parameters for Exec Processing Exit (continued)

Parameter	Number of Bytes	Description
Parameter 8	4	The address of a user field. ARXEXEC 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`.

The exec processing exit has the same return specifications as ARXITMV. See “ARXITMV” on page 470.
The exec processing exit has the same return codes as ARXINITX; see Table 87 on page 470.

Installing the Exec Processing, Exec Initialization, and Exec Termination

For the exec initialization exit, specify the exit's name in the EXECINIT field in the module name table. For the exec termination exit, specify the exit's name in the EXECTERM field. For the exec processing exit, specify the exit's name in the IRXEXECX field. Link-edit these exits as separate phases.

Chapter 22. 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 59. 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 page "OPTIONS" on page 43 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 479.

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 Characters in Symbols and Comments

To enable the use of DBCS characters in symbols and comments, use the ETMODE option of the OPTIONS instruction. For more information, see “OPTIONS” on page 43.

The following are some ways that DBCS names can be used:

- as variables or labels within your program
- as constant symbols
- to pass parameters on the LINKPGM host command environment.
- as a STEM name on EXECIO or as a trapping variable for the OUTTRAP function
- in functions such as SYMBOL and DATATYPE
- in arguments of functions (such as LENGTH)
- in the variable access routine ARXEXCOM.

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  */
<.S.Y.M.D> = 10          /* Variable with DBCS characters between */
                         /* shift-out (<) and shift-in (>)        */
y.<.S.Y.M.D> = JUNK
CALL <.D.B.C.S.R.T.N>     /* Call subroutine with DBCS name        */
EXIT
<.D.B.C.S.R.T.N>:         /* Subroutine with DBCS name             */
DO i = 1 TO 10
  IF y.i = JUNK THEN      /* Does y.i match the DBCS variable's
                              value?                               */
    SAY 'Value of the DBCS variable is : ' <.S.Y.M.D>
END
RETURN
```

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 the current output stream. ASSGN(STDOUT) returns the name of the current output stream. 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 output stream or device.

When the data is split up in shorter lengths, again the DBCS data integrity is kept under OPTIONS EXMODE. In EBCDIC, if the default output width 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 59.

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
```

Note: 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.

LEFT, RIGHT, and CENTER

In EBCDIC:

```
LEFT('<.A.B.C.D.E>',4)     ->   '<.A.B.C.D>'
LEFT('a<>',2)              ->   'a<> '
LEFT('<.A>',2,'*')         ->   '<.A>*'
RIGHT('<.A.B.C.D.E>',4)    ->   '<.B.C.D.E>'
RIGHT('a<>',2)             ->   ' a'
CENTER('<.A.B>',10,'<.E>') ->   '<.E.E.E.E.A.B.E.E.E.E>'
CENTER('<.A.B>',11,'<.E>') ->   '<.E.E.E.E.A.B.E.E.E.E.E>'
CENTER('<.A.B>',10,'e')    ->   'eeee<.A.B>eeee'
```

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.

REVERSE

In EBCDIC:

```
REVERSE('<.A.B><.C.D><>')      ->   '<><.D.C><.B.A>'
```

Applying the character extraction from a string and character concatenation rules.

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.

Note: When used with DBCS functions, *length* is always measured in bytes (as opposed to LENGTH(*string*), which is measured in characters).

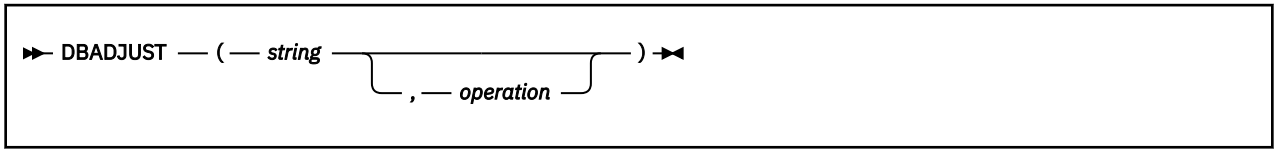
Counting Option

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>'
```

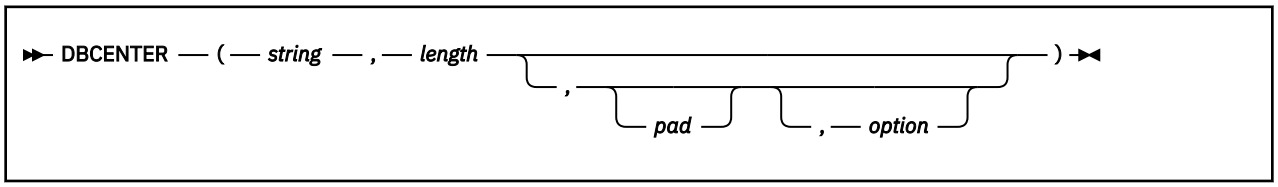
DBBRACKET

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



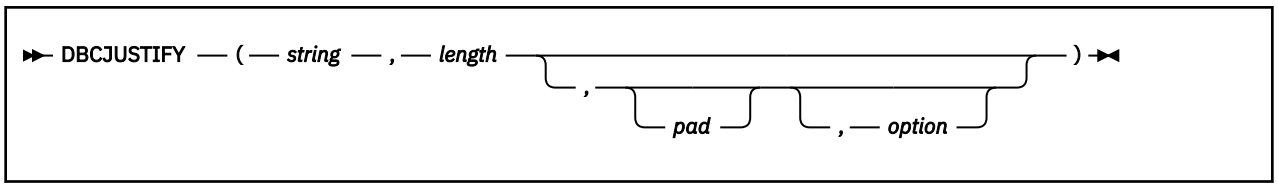
returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up length. The default *pad* character is a blank. If *string* is longer than *length*, it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand 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.

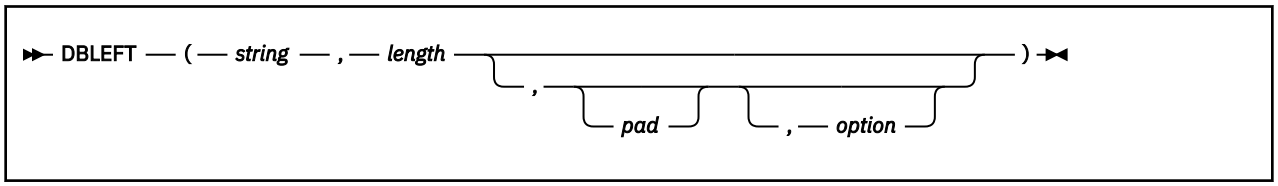
The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

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



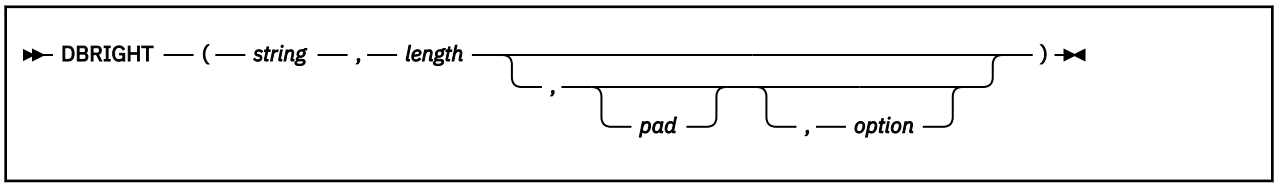
returns a string of length *length* containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank.

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



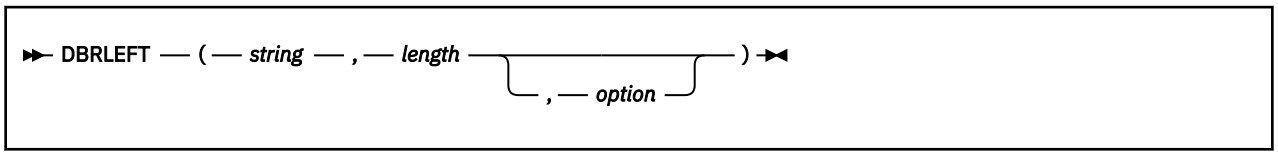
returns a string of length *length* containing the rightmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank.

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



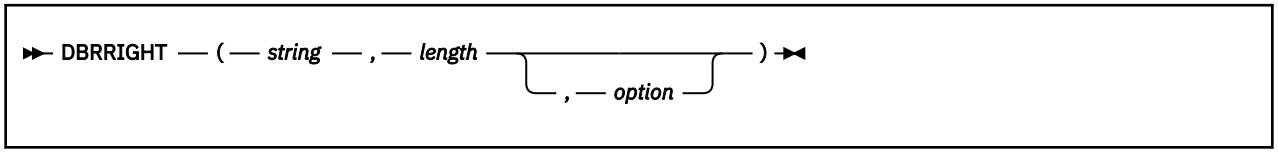
returns the remainder from the DBLEFT function of *string*. If *length* is greater than the length of *string*, returns a null string.

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>'
```

Note: 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>'
```

Note: 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.

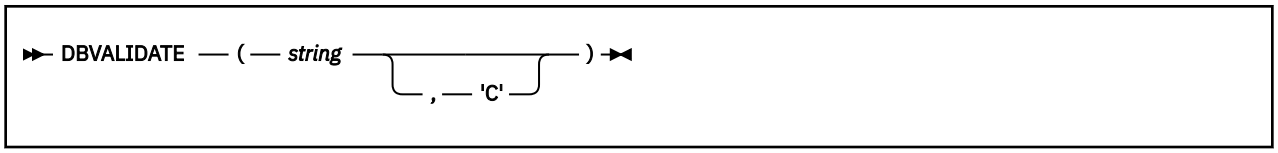
DBUNBRACKET

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:

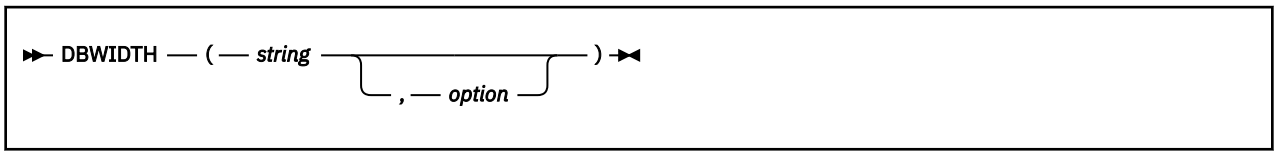
1. Only valid DBCS character codes
2. DBCS string is an even number of bytes in length
3. EBCDIC only — Proper SO and SI pairing.

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
```

Chapter 23. ARXTERMA Routine

The ARXTERMA routine terminates a language processor environment. ARXTERMA differs from the ARXTERM termination routine. ARXTERM terminates a language processor environment only if no active REXX programs are currently running in the environment. ARXTERMA terminates all active REXX programs under a language processor environment, and optionally terminates the environment. If you customize REXX processing and initialize a language processor environment using the ARXINIT initialization routine, when you terminate the environment, you are recommended to use the ARXTERM termination routine. "Termination Routine – ARXTERM" on page 437 describes ARXTERM.

Note: To permit FORTRAN programs to call ARXTERMA, REXX/VSE provides an alternate entry point for the ARXTERMA routine. The alternate entry point name is ARXTMA.

On the call to ARXTERMA, you specify whether ARXTERMA should terminate the environment in addition to terminating all active programs 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 ARXTERMA to run. You can pass the address either in parameter 2 or in register 0. If you do not pass an environment block address, ARXTERMA locates the current non-reentrant environment that was created at the same task level and runs in that environment.

ARXTERMA 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, ARXTERMA does terminate all active programs running in the environment.

ARXTERMA invokes the exec load routine to free each program in the environment. The exec load routine is the routine the EXROUT field in the module name table identifies, which is one of the parameters for the initialization routine, ARXINIT. All programs in the environment are freed regardless of whether they were pre-loaded before the ARXEXEC routine was called. ARXTERMA also frees the storage for each program in the environment.

ARXTERMA sets the ENVBLOCK_TERMA_CLEANUP flag to indicate that ARXTERMA is cleaning up the environment. ARXTERMA frees all active programs and optionally terminates the environment itself. The replaceable routines can use this ENVBLOCK_TERMA_CLEANUP flag to allow special processing during abnormal termination. If ARXTERMA does not terminate the environment, the flag is cleared upon exit from ARXTERMA.

Entry Specifications

For the ARXTERMA 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 the caller passes

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. Set the high-order bit of the last address to 1 to indicate the end of the parameter list. For more information about passing parameters, see “Parameter Lists for REXX/VSE Routines” on page 325.

Table 93 on page 496 shows the parameters for ARXTERMA.

Table 93. Parameters for ARXTERMA

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 programs running in the environment. Specify one of the following: • 0 — terminates all programs and the environment • X'80000000' — terminates all programs, but does not terminate the environment.
Parameter 2	4	This parameter is optional. It is the address of the environment block that represents the environment you want ARXTERMA to terminate. If you do not want to use this parameter, set the high-order bit in the address that points to parameter 1 to 1 to end the parameter list. (You cannot simply specify an address of 0 because ARXTERMA tries to use 0 as a valid address and fails with a return code of 28.) If you specify an environment block address, ARXTERMA uses the value you specify and ignores register 0. However, ARXTERMA does not check whether the address is valid. Therefore, ensure the address you specify is correct or unpredictable results can occur. If you use register 0 to specify the address of an environment block, ARXTERMA checks whether the address is valid. If the address is valid, ARXTERMA terminates that environment. Otherwise, ARXTERMA locates the current non-reentrant environment that was created at the same task level and terminates that environment.

Return Specifications

For the ARXTERMA termination routine, the contents of the registers on return are:

Register 0
If you passed the address of an environment block in register 0, ARXTERMA 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.

Registers 1-14
Same as on entry

Register 15
Return code

Return Codes

Table 94 on page 497 shows the return codes for the ARXTERMA routine.

Table 94. Return Codes for ARXTERMA

Return Code	Description
0	Processing was successful. If ARXTERMA also terminated the environment, the environment was not the last environment on the task.
4	Processing was successful. If ARXTERMA also terminated the environment, the environment was the last environment on the task.
20	Processing was not successful. ARXTERMA could not terminate the environment.
28	Processing was not successful. The environment could not be found.

Chapter 24. Support for the Library for REXX/370 in REXX/VSE

Before using the Library for REXX/370 in REXX/VSE to execute a compiled REXX program, you need to compile the program on VM CMS or MVS. This appendix introduces the compiler and describes the support for the Library for REXX/370 in REXX/VSE.

Benefits of Using a Compiler

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

- 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 the host takes to process the commands.

Reduced System Load

Compiled REXX programs run faster than interpreted programs. Because a program has to be compiled only once, 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 on the interpreter. Running the program 60 times 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.

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 compiled REXX program can run under other operating systems, such as MVS/ESA* or VM CMS. A REXX program compiled under VM CMS or MVS/ESA can run under REXX/VSE.

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 set of common elements of the REXX language. The REXX/VSE interpreter supports these elements.

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

Compiler Publications

For more information about the compiler, see the following books:

- *IBM Compiler and Library for SAA REXX/370 Release 2: Introducing the Next Step in REXX Programming*, G511-1430
- IBM Compiler and Library for SAA REXX/370 3 User's Guide and Reference
- IBM Compiler and Library for SAA REXX/370 Diagnosis Guide

Routines and Interfaces for the Library for REXX/370 in REXX/VSE

REXX/VSE provides routines and interfaces it uses during the execution of compiled programs under a compiler runtime processor.

Central to compiler support is the compiler programming table. REXX/VSE uses the compiler runtime processor name stored in the compiled REXX program to locate the entry for the compiler runtime processor in the compiler programming table. The compiler programming table entry contains the name of the compiler runtime processor and the names of up to four optional compiler interface routines. REXX/VSE uses the compiler runtime processor to run compiled programs. During the execution of a compiled program, REXX/VSE invokes compiler interface routines to perform specialized processing.

The following information, to the end of the chapter, is product-sensitive programming interface information.

Programming Routines for a REXX Compiler Runtime Processor

REXX/VSE provides various programming routines that support a REXX compiler runtime processor. These routines are:

- ARXERS - a REXX compiler programming routine that searches for and runs an external routine. For more information on the search order for external routines, see “Search Order” on page 60.
- ARXHST - a REXX compiler programming routine that searches for and runs a host command. For more information on locating host commands, see “Commands” on page 23.
- ARXRTE - a REXX compiler programming routine that searches for and invokes a REXX exit routine. For more information on REXX exit routines, see “REXX Exit Routines” on page 468

In addition, you can use the GETEVAL function of the ARXRLT programming service to obtain the evaluation block for an external function or subroutine. These routines and the GETEVAL function of ARXRLT are intended for use only by a compiler runtime processor. For more information on the ARXRLT programming service, see “Programming Services” on page 135.

Routines and Interfaces to Support a REXX Compiler

This section discusses the characteristics of a compiled REXX program and the routines and interfaces to support a REXX compiler, including:

- The compiler programming table
- The compiler runtime processor

- The four compiler interface routines:
  - Compiler interface initialization routine
  - Compiler interface termination routine
  - Compiler interface load routine
  - Compiler interface variable handling routine.

Overview

REXX/VSE defines a format for compiled REXX programs so that REXX/VSE can distinguish between compiled and interpreted programs. REXX/VSE also provides a defined interface for installing a REXX compiler runtime processor.

A compiler runtime processor executes compiled programs. To initiate runtime processing of a compiled REXX program, REXX/VSE uses a compiler programming table to identify the runtime processor and up to four interface routines. You can modify the compiler programming table to identify routines for a compiler runtime processor, if a compiler runtime processor is installed. Each of the four compiler interface routines are optional and can provide special processing for initializing and terminating the compiler runtime processor, loading compiled REXX programs, and accessing REXX variables.

How REXX Identifies a Compiled Program

During REXX program processing, REXX/VSE determines whether a program is compiled or interpreted. REXX/VSE recognizes a program as compiled if the program meets the following three criteria:

- The length of the first record is at least 20 bytes
- The string 'EXECPROC' is in columns 5–12 of first record
- The first non-blank in columns 1–4 of the first record is not a comment delimiter.

If a program meets these criteria, REXX/VSE determines the name of a compiler runtime processor from columns 13–20 of the first record.

You might find that some interpreted programs meet these criteria and are, therefore, incorrectly executed as compiled programs. There are several ways to correct this problem, including:

- Shift everything in the first record one column to the right. This leaves the string 'EXECPROC' in the first record, but not in the expected position (columns 5–12) for a compiled program.
- Add a comment as the first record of the REXX program. The record that contains 'EXECPROC' remains intact as the second record.

The Compiler Programming Table

The compiler programming table is a control block that REXX/VSE uses to obtain information about a compiler runtime processor. This information includes the names of up to four optional compiler interface routines. Before REXX/VSE runs the first compiled program in the first language processor environment REXX/VSE loads ARXCMPTM as the compiler programming table. Once the compiler programming table is loaded, it is used for all compiled programs in the current and any subsequent language processor environments.

The ARXCMPTM module is in PRD1.BASE. Source for a sample compiler programming table is in PRD1.BASE member ARXCMPTM.Z. If you want to install another REXX compiler runtime processor, you can create your own compiler programming table using this member as a model. After you create the source for the compiler programming table, assemble and link-edit the table as phase ARXCMPTM. You must place ARXCMPTM in the SVA.

A mapping macro, ARXCMPTB, for the compiler programming table is in PRD1.BASE. See Table 95 on page 502 and Table 96 on page 502 for the format of the compiler programming table.

Note: Each field name in the following tables must include the prefix COMPGMTB_.

Table 95. Compiler Programming Table Header Information

Offset (Decimal)	Number of Bytes	Field Name	Description
0	4	FIRST	Address of the first entry
4	4	TOTAL	Total number of entries
8	4	USED	Number of entries used
12	4	LENGTH	Length of each entry
16	8	--	Reserved
24	8	_	X'FFFFFFFFFFFFFFFF'.

Table 96. Compiler Programming Table Entry Information

Offset (Decimal)	Number of Bytes	Field Name	Description
0	8	RTPROC	Name of the Compiler Runtime Processor
8	8	COMPINIT	Name of the Compiler Interface Initialization Routine
16	8	COMPTERM	Name of the Compiler Interface Termination Routine
24	8	COMPLOAD	Name of the Compiler Interface Load Routine
32	8	COMPVAR	Name of the Compiler Interface Variable Handling Routine
40	16	STORAGE	Four words of storage that a REXX compiler runtime processor can use. For example, a REXX compiler runtime processor might use these storage words as anchors for its control block structure.

Figure 33 on page 503 shows the sample compiler programming table shipped in PRD1.BASE member ARXCMPTM.Z. EAGRTXIN is the name of the compiler interface initialization routine. EAGRTXTR is the name of the compiler interface termination routine. EAGRTPRC is the name of the compiler runtime processor. EAGRTXLD is the name of the compiler interface load routine. EAGRTXVH is the name of the compiler interface variable handling routine.

```
ARXCMPTM CSECT ,
ARXCMPTM AMODE 31
ARXCMPTM RMODE ANY

ARXCMPTB_HEADER        DS    0CL32
ARXCMPTB_FIRST         DC    AL4(FIRST_ENTRY)
ARXCMPTB_TOTAL         DC    F'1'
ARXCMPTB_USED          DC    F'1'
ARXCMPTB_LENGTH        DC    F'56'
                       DC    X'0000000000000000'
ARXCMPTB_FFFF          DC    X'FFFFFFFFFFFFFFFF'
FIRST_ENTRY            DS    0CL56
FIRST_ENTRY_RTPROC     DC    C'EAGRTPRC'
FIRST_ENTRY_COMPINIT   DC    C'EAGRTXIN'
FIRST_ENTRY_COMPTERM   DC    C'EAGRTXTR'
FIRST_ENTRY_COMPLOAD   DC    C'EAGRTXLD'
FIRST_ENTRY_COMPVAR    DC    C'EAGRTXVH'
FIRST_ENTRY_STORAGE    DC    4F'0'
         END ARXCMPTM
```

Figure 33. Sample Compiler Programming Table

The Compiler Runtime Processor

When REXX/VSE encounters a compiled REXX program, REXX/VSE passes control to the appropriate compiler runtime processor to run the program. Before the first invocation of a compiled REXX program in the first language processor environment, REXX/VSE loads the appropriate compiler runtime processor, saves the location of the compiler runtime processor, then invokes the compiler runtime processor. On subsequent invocations of compiled REXX programs, and in subsequent language processor environments, REXX/VSE uses the saved location of the loaded compiler runtime processor to pass control to the compiler runtime processor.

The compiler runtime processor must issue all messages relating to language processing. This includes those *VSE/ESA Messages and Codes* contains.

When the compiler runtime processor receives control, it must pass control to the exec initialization routine (EXECINIT) and exec termination routine (EXECTERM) at the appropriate times. The programming routine ARXRTE must pass control to these routines.

Table 97 on page 504 describes the results required from a compiler runtime processor. The results vary according to how the compiled program was invoked under the compiler runtime processor.

Table 97. Compiler Runtime Processor Expected Results

Method of Invocation (Compiled Program)	**Returned Results (Compiled Program)**			
	EXIT/RETURN Without Expression	**EXIT/RETURN With Expression**	**Language Error**	**Processing Error**
Subroutine	Set return code to `0`. The compiler runtime processor must not obtain or complete an EVALBLOK.	Set return code to `0`. The compiler runtime processor must use the GETEVAL function of ARXRLT to obtain an EVALBLOK. The compiler runtime processor must then use the results from the execution of the compiled program to complete the EVALBLOK.	Set return code to `200`*nn*, where `1`≤*nn*≤`99`. The compiler runtime processor must not obtain or complete an EVALBLOK.	Set return code to `20`. The compiler runtime processor must not obtain or complete an EVALBLOK.
Function	For a RETURN without expression, set the return code to `20045`. Return code `20045` is a special case of return code `200`*nn*. For an EXIT without expression, set the return code to `0`.	Set return code to `0`. The compiler runtime processor must use the GETEVAL function of ARXRLT to obtain an EVALBLOK. The compiler runtime processor must then use the results from the execution of the compiled program to complete the EVALBLOK.	Set return code to `200`*nn*, where `1`≤*nn*≤`99`. The compiler runtime processor must not obtain or complete an EVALBLOK.	Set return code to `20`. The compiler runtime processor must not obtain or complete an EVALBLOK.

Table 97. Compiler Runtime Processor Expected Results (continued)

Method of Invocation (Compiled Program)	Returned Results (Compiled Program)			
	EXIT/RETURN Without Expression	**EXIT/RETURN With Expression**	**Language Error**	**Processing Error**
Command	Set return code to 0. The compiler runtime processor must use the GETEVAL function of ARXRLT to obtain an EVALBLOK. The compiler runtime processor must then complete the EVALBLOK with a result of 0.	Set return code to 0. The compiler runtime processor must represent the results from the compiled program execution as a number in string format. If the result string fits in a fullword, the compiler runtime processor must use the GETEVAL function of ARXRLT to obtain an EVALBLOK. The compiler runtime processor must then complete the EVALBLOK with the result string. If the result string does not fit in a fullword, then the compiler runtime processor must set the return code to 20026 and must not obtain or modify an EVALBLOK.	Set return code to 200*nn*, where 1≤*nn*≤99. The compiler runtime processor must not obtain or complete an EVALBLOK.	Set return code to 20. The compiler runtime processor must not obtain or complete an EVALBLOK.

Entry Specifications

The contents of the registers on entry to the compiler runtime processor are:

Register 0
Address of an 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 for the Compiler Runtime Processor

In register 1, REXX/VSE passes the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. REXX/VSE passes all parameters on the call. REXX/VSE sets the high-order bit of the last address in the parameter list to 1. Table 98 on page 506 lists the parameters for the compiler runtime processor.

Table 98. Parameters for a Compiler Runtime Processor

Parameter	Number of Bytes	Description
Parameter 1	4	EXECBLK address. On entry to the compiler runtime processor, this parameter contains the address of the REXX exec block (EXECBLK) that ARXLOAD uses. The exec block is a control block that describes the program to be loaded. For more information on the exec block parameter for ARXLOAD, see "The Exec Block" on page 445.
Parameter 2	4	Compiled Program arguments. On entry to the compiler runtime processor, this parameter contains the address of a series of address/length pairs that describe the arguments for the program. A double word of X'FFFFFFFFFFFFFFFF' delineates the end of the pairs. For more information on REXX program arguments, see "Format of Argument List" on page 338.
Parameter 3	4	A fullword of flag bits. For more information on flag bits, see the ARXEXEC parameters on page Table 14 on page 335.
Parameter 4	4	In-storage control block address. The in-storage control block contains a series of address/length pairs that REXX uses to describe the structure of a loaded program in storage. ARXLOAD or the compiler interface load routine initializes the in-storage control block before a compiler runtime processor receives control. For more information on the in-storage control block, see "The In-Storage Control Block (INSTBLK)" on page 339.
Parameter 5	4	This is reserved.
Parameter 6	4	Address of a user field. When a program calls ARXEXEC to invoke a compiled REXX program, the program can pass the address of a user field. ARXEXEC passes the user field address to the compiler runtime processor in this parameter. For more information on the user field, see Table 14 on page 335.
Parameter 7	4	Environment block address. On entry, this parameter contains the address of the REXX environment block with which the compiler programming table is associated. This parameter is identical to the address in register 0. For more information on the REXX environment block, see "Format of the Environment Block (ENVBLOCK)" on page 414.
Parameter 8	4	Compiler runtime processor entry address. Specifies the address of the entry in the compiler programming table for the compiler runtime processor.
Parameter 9	4	Compiler runtime processor return code. On exit, the compiler runtime processor must set this parameter to a return code that indicates the completion status of the compiler runtime processor. Table 99 on page 507 lists the return codes for the compiler runtime processor.

Table 98. Parameters for a Compiler Runtime Processor (continued)

Parameter	Number of Bytes	Description
Parameter 10	4	Compiler runtime processor abend and reason codes. The abend and reason codes are the same as those ARXEXEC returns. For more information on abend and reason codes for ARXEXEC, see page "Return Codes".

Return Specifications

On return from the compiler runtime processor, the contents of registers 0–14 must be the same as on entry.

Return Codes

Table 99 on page 507 lists the return codes the compiler runtime processor issues.

Table 99. Return Codes from a REXX Compiler Runtime Processor

Return Code (Decimal)	Description
0	Processing was successful. Table 97 on page 504 shows the expected results from the compiler runtime processor.
20	Processing was not successful. The compiler runtime processor issued an error message that describes the error.
20001–20099	Processing was successful. However, the compiler runtime processor detected a syntax error in the compiled program. The return code value is 20000 plus the value of the REXX error number. See z/VSE Messages and Codes.

Programming Considerations

The compiler runtime processor must follow standard linkage conventions. It must save the registers on entry and restore the registers when it returns. The compiler runtime processor must be reentrant.

Environment

The attributes for the compiler runtime processor are:

- State: Problem Program
- Key: 8
- AMODE(31)/RMODE(ANY)
- ASC mode: Primary
- Task Mode
- Reentrant.

Compiler Interface Routines

During various stages of processing a compiled REXX program, REXX/VSE invokes a compiler interface routine, if installed, to perform special processing. The compiler runtime processor is not required to use the compiler interface routines. However, you must install those compiler interface routines that the compiler runtime processor requires. To indicate to REXX/VSE that a compiler interface routine is not

required, specify a module name of eight blanks in the appropriate field of the compiler programming table entry. The four compiler interface routines are:

Compiler interface initialization routine
Initializes a compiler runtime processor

Compiler interface termination routine
Terminates a compiler runtime processor

Compiler interface load routine
Performs specialized processing to service a request to load or free a compiled program

Compiler interface variable handling routine
Performs specialized processing to service a request to access REXX variables.

Compiler Interface Initialization Routine

This routine, if installed, receives control to initialize a compiler runtime processor before the compiler runtime processor is invoked for the first time. REXX/VSE invokes a compiler interface initialization routine once for each compiler runtime processor that runs in a REXX language processor environment.

Entry Specifications

The contents of the registers on entry to the compiler interface initialization routine are:

Register 0
Address of an 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

Parameter List

In register 1, REXX/VSE passes the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. REXX/VSE passes all parameters on the call and sets the high-order bit of the last address in the parameter list to 1. The following table lists the parameters for the compiler interface initialization routine.

Table 100. Parameter List for the Compiler Interface Initialization Routine

Parameter	Number of Bytes	Description
Parameter 1	4	Environment block address. On entry, this parameter contains the address of the REXX environment block with which the compiler programming table is associated. This parameter is identical to the address in register 0. For more information on the REXX environment block, see “Format of the Environment Block (ENVBLOCK)” on page 414.

Table 100. Parameter List for the Compiler Interface Initialization Routine (continued)

Parameter	Number of Bytes	Description
Parameter 2	4	Compiler runtime processor entry address. Specifies the address of the entry in the compiler programming table for the compiler runtime processor.
Parameter 3	4	Compiler interface initialization routine return code. On exit, the compiler interface initialization routine must set this parameter to a return code that indicates the completion status of the compiler interface initialization routine. Table 101 on page 509 lists the return codes for the compiler interface initialization routine.

Return Specifications

On return from the compiler interface initialization routine, the contents of registers 0–14 must be the same as on entry.

Return Codes

Table 101 on page 509 lists the return codes the compiler interface initialization routine issues.

Table 101. Return Codes from the Compiler Interface Initialization Routine

Return Code (Decimal)	Description
0	Processing was successful. REXX/VSE can now pass control to the compiler runtime processor.
20	Processing was not successful. REXX/VSE does not give control to the associated compiler runtime processor. REXX/VSE does not execute any compiled REXX program that uses the associated compiler runtime processor.

Programming Considerations

The compiler interface initialization routine must follow standard linkage conventions. It must save the registers on entry and restore the registers when it returns. The compiler interface initialization routine must be reentrant.

Environment

The attributes for the compiler interface initialization routine are:

- State: Problem Program
- Key: 8
- AMODE(31)/RMODE(ANY)
- ASC mode: Primary
- Task Mode
- Reentrant.

Compiler Interface Termination Routine

This routine, if installed, receives control at the termination of a REXX language processor environment.

Entry Specifications

The contents of the registers on entry to the compiler interface termination routine are:

Register 0
Address of an 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

Parameter List

In register 1, REXX/VSE passes the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. REXX/VSE passes all parameters on the call and sets the high-order bit of the last address in the parameter list to 1. The following table lists the parameters for the compiler interface termination routine.

Table 102. Parameter List for the Compiler Interface Termination Routine

Parameter	Number of Bytes	Description
Parameter 1	4	Environment block address. On entry, this parameter contains the address of the REXX environment block with which the compiler programming table is associated. This parameter is identical to the address in register 0. For more information on the REXX environment block, see "Format of the Environment Block (ENVBLOCK)" on page 414.
Parameter 2	4	Compiler runtime processor entry address. Specifies the address of the entry in the compiler programming table for the compiler runtime processor.
Parameter 3	4	Compiler interface termination routine return code. This parameter is reserved for future use. REXX/VSE initializes this parameter to 0 and does not inspect the parameter on return from the compiler interface termination routine.

Return Specifications

On return from the compiler interface termination routine, the contents of registers 0–14 must be the same as on entry.

Return Codes

The return code parameter in the compiler interface termination routine is reserved for future use. The compiler interface termination routine must not modify the return code parameter; REXX/VSE does not inspect the return code parameter.

Programming Considerations

The compiler interface termination routine must follow standard linkage conventions. It must save the registers on entry and restore the registers when it returns. The compiler interface termination routine must be reentrant.

Environment

The attributes for the compiler interface termination routine are:

- State: Problem Program
- Key: 8
- AMODE(31)/RMODE(ANY)
- ASC mode: Primary
- Task Mode
- Reentrant.

Compiler Interface Load Routine

ARXLOAD passes control to the compiler interface load routine in either of two cases:

- After the REXX language processor reads a compiled REXX program into storage.
- When the REXX language processor makes a request to free the in-storage control block that an earlier request to the compiler interface load routine created.

Note: This section discusses the interaction between the compiler interface load routine and the IBM-supplied ARXLOAD routine.

For compiled programs, ARXLOAD calls the compiler interface load routine, if installed, before ARXLOAD builds the in-storage control block and after ARXLOAD has obtained all information the compiler interface load routine requires.

One of the inputs (parameter 5) to the compiler interface load routine is a group of blocks containing the compiled REXX program. The compiler interface load routine must create and initialize an in-storage control block from the group of blocks, preferably above 16 megabytes in virtual storage. For more information about the in-storage control block, see “The In-Storage Control Block (INSTBLK)” on page 339.

Entry Specifications

The contents of the registers on entry to the compiler interface load routine are:

Register 0
Address of an 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

Parameter List

In register 1, the calling program (ARXLOAD) passes the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. ARXLOAD passes all parameters on the call and sets the high-order bit of the last address in the parameter list to 1. The following table lists the parameters for the compiler interface load routine.

Table 103. Parameter List for the Compiler Interface Load Routine

Parameter	Number of Bytes	Description
Parameter 1	8	Function requested. On entry, this parameter contains the function requested of the compiler interface load routine The function specification is in uppercase, left-justified, and padded on the right with blanks. Acceptable values are: **"LOAD "** Specifies that the compiler interface load routine is to load a program into storage. **"FREE "** Specifies that the compiler interface load routine is to free the program represented by the in-storage control block specified in parameter 8. For more information on the LOAD and FREE functions, see "Functions You Can Specify...".
Parameter 2	4	EXECBLK address. On entry to the compiler interface load routine, this parameter contains the address of the REXX exec block (EXECBLK) that ARXLOAD uses. The exec block is a control block that describes the program to be loaded. For more information on the exec block parameter for ARXLOAD, see “The Exec Block” on page 445.
Parameter 3	4	Record format. On entry, this parameter specifies the format of records in the blocks passed to this routine in parameter 5. Possible values for this parameter are 'F ' for fixed-length records and 'V ' for variable-length records. Variable-length records do not span across blocks.
Parameter 4	4	Record length. On entry, this parameter specifies the length of each record for fixed-length records, or the maximum record length for variable-length records. Each variable-length record contains a record descriptor word (RDW). The first two bytes of the RDW indicate the actual length of the record, including the RDW.
Parameter 5	4	Address of a vector of address/length pairs. Each address/length pair contains the address and length of a block of data that contains the statements of the program. A double word of X'FFFFFFFFFFFFFFFF' indicates the end of the pairs.
Parameter 6	4	Environment block address. On entry, this parameter contains the address of the REXX environment block with which the compiler programming table is associated. This parameter is identical to the address in register 0. For more information on the REXX environment block, see “Format of the Environment Block (ENVBLOCK)” on page 414.
Parameter 7	4	Compiler runtime processor entry address. Specifies the address of the entry in the compiler programming table for the compiler runtime processor.

Table 103. Parameter List for the Compiler Interface Load Routine (continued)

Parameter	Number of Bytes	Description
Parameter 8	4	In-storage control block address. The in-storage control block contains a series of address/length pairs that REXX uses to describe the structure of a loaded program in storage. For more information on the in-storage control block, see "The In-Storage Control Block (INSTBLK)" on page 339. When ARXLOAD invokes the compiler interface load routine to load a compiled program, the compiler interface load routine should create an in-storage control block and place the control block address in this parameter. ARXLOAD considers this parameter to be valid only when the return code from the compiler interface load routine is 0. When ARXLOAD invokes the compiler interface load routine to free storage for the REXX program, this parameter contains the address of the in-storage control block that the compiler interface load routine previously created and is to free. For complete details on in-storage control blocks, see "The In-Storage Control Block (INSTBLK)" on page 339.
Parameter 9	4	Compiler interface load routine return code. On exit, the compiler interface load routine must set this parameter to a return code that indicates the completion status of the compiler interface load routine. Table 104 on page 513 lists the return codes issued by the compiler interface load routine.

Return Specifications

On return from the compiler interface load routine, the contents of registers 0–14 must be the same as on entry.

Return Codes

Table 104 on page 513 lists the return codes issued by the compiler interface load routine.

Table 104. Return Codes from the Compiler Interface Load Routine

Return Code (Decimal)	Description
0	Processing was successful. If the requested function was LOAD, parameter 8 contains the address of the created in-storage control block. If the requested function was FREE, the in-storage control block specified in parameter 8 has been freed.
4	Processing was successful. However, the compiler interface load routine did not create an in-storage control block. ARXLOAD will create an in-storage control block.
20	Processing was not successful. A severe error has occurred. The compiler interface load routine should issue a message to accompany this return code. ARXLOAD propagates a return code of 20 to the caller of ARXLOAD.

Programming Considerations

The compiler interface load routine must follow standard linkage conventions. It must save the registers on entry and restore the registers when it returns. The compiler interface load routine must be reentrant.

Environment

The attributes for the compiler interface load routine are:

- State: Problem Program
- Key: 8
- AMODE(31)/RMODE(ANY)
- ASC mode: Primary
- Task Mode
- Reentrant.

Compiler Interface Variable Handling Routine

The compiler interface variable handling routine, if installed, receives control whenever an external routine or host command requests access to REXX variables using ARXEXCOM.

Entry Specifications

The contents of the registers on entry to the compiler interface variable handling routine are:

Register 0
Address of an 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

Parameter List for the Compiler Interface Variable Handling Routine

In register 1, the calling program passes the address of a parameter list, which consists of a list of addresses. Each address in the parameter list points to a parameter. REXX/VSE sets the high-order bit of the last address in the parameter list to 1. Table 105 on page 515 lists the parameters for the compiler interface variable handling routine.

Table 105. Parameter List for the Compiler Interface Variable Handling Routine

Parameter	Number of Bytes	Description
Parameter 1	1	Variable handling function request. On entry to the compiler interface variable handling routine, this parameter contains a 1-character field corresponding to the shared variable request code (SHVCODE) used by ARXEXCOM. For more information on shared variable request codes, see "SHVCODE". This routine must also support the function 'n'—Fetch Next with Mask. The Fetch Next with Mask function must search through all variables known to the language processor. These variables include stem variables that have been assigned a value. The output from this function is expected to be the next variable that begins with the specified mask.
Parameter 2	4	The address of the variable name to be manipulated. This is an input parameter for the following functions: **Function** **SHVCODE** **Set Variable** 'S','s' **Fetch Variable** 'F','f' **Drop Variable** 'D','d' **Fetch Private** 'P' For the Fetch Next ('N') and Fetch Next with Mask ('n') functions, this parameter must be set on output to the address of the next variable name.
Parameter 3	4	Length of variable name. Specifies the length of the string to which the address in parameter 2 points.
Parameter 4	4	Address of the value for the variable. This is an input parameter for the Set Variable function ('S','s') and an output parameter for the following functions: **Function** **SHVCODE** **Fetch Variable** 'F','f' **Fetch Next** 'N' **Fetch Next with Mask** 'n' **Fetch Private** 'P' This parameter is not used for the Drop Variable ('D','d') function.
Parameter 5	4	Length of the value for the variable. Specifies the length of the value to which the address in parameter 4 points.

Table 105. Parameter List for the Compiler Interface Variable Handling Routine (continued)

Parameter	Number of Bytes	Description
Parameter 6	4	Work block extension address. On entry, this parameter contains the address of the work block extension. The work block extension contains the WORKEXT_RTPROC field, which the compiler runtime processor can use as an anchor for resources that are specific to a particular compiled program.
Parameter 7	4	Compiler runtime processor entry address. Specifies the address of the entry in the compiler programming table for the compiler runtime processor.
Parameter 8	4	Environment block address. On entry, this parameter contains the address of the REXX environment block with which the compiler programming table is associated. This parameter is identical to the address in register 0. For more information on the REXX environment block, see “Format of the Environment Block (ENVBLOCK)” on page 414.
Parameter 9	1	Shared variable function return code (SHVRET). On output, the compiler interface variable handling routine must set this parameter to the appropriate value for the SHVRET field. The values returned in this parameter for the Fetch Next with Mask function must be identical to those returned for the Fetch Next function. For a list of appropriate values for the SHVRET field, see "SHVBLOCK".
Parameter 10	4	Compiler interface variable handling routine return code. On exit, the compiler interface variable handling routine must set this parameter to a return code that indicates the completion status of the compiler interface variable handling routine. Table 106 on page 517 lists the return codes for the compiler interface variable handling routine.
Parameter 11	4	Fetch next mask. This parameter is optional and used only with the Fetch Next with Mask function ('n'). When the language processor provides this parameter, it specifies an address of a mask used to search for the next variable or stem. The mask can be a character string that meets the naming conventions for simple variables or variable stems. The mask cannot identify a compound variable. The compiler interface variable handling routine must return a variable whose name begins with the mask provided. A parameter value of 0 indicates that no mask is provided.
Parameter 12	4	Fetch next mask length. This parameter is optional and may be used only in conjunction with parameter 11. This value is the length of the mask provided in parameter 11. This parameter is ignored if the value in parameter 11 is 0.

Return Specifications

On return from the compiler interface variable handling routine, the contents of registers 0–14 must be the same as on entry.

Return Codes

Table 106 on page 517 lists the return codes the compiler interface variable handling routine issues.

Table 106. Return Codes from the Compiler Interface Variable Handling Routine

Return Code (Decimal)	Description
0	Processing was successful.
4	Processing was not successful. Insufficient storage was available.
8	Processing was not successful. The name that was passed in parameter 2, or created by a symbolic substitution on parameter 2, is too long.
12	Processing was not successful. The name that was passed in parameter 2, or created by a symbolic substitution on parameter 2, is incorrect because it begins with a character that is not valid.
20	Processing was not successful.

Programming Considerations

The compiler interface variable handling routine must follow standard linkage conventions. It must save the registers on entry and restore the registers when it returns. The compiler interface variable handling routine must be reentrant.

Environment

The attributes for the compiler interface variable handling routine are:

- State: Problem Program
- Key: 8
- AMODE(31)/RMODE(ANY)
- ASC mode: Primary
- Task Mode
- Reentrant.

Environment for the Programming Routines

The ARXERS, ARXHST, and ARXRTE programming routines must run in an environment with the following characteristics:

- State: Problem Program
- Key: 8
- AMODE(31)/RMODE(ANY)
- ASC mode: Primary
- Task mode.

External Routine Search Routine (ARXERS)

ARXERS is a programming routine that searches for and runs an external routine. ARXERS allows a compiler runtime processor to pass control to an external routine by a direct interface. A compiler runtime processor that uses ARXERS leaves the implementation of the external routine search and invocation to REXX/VSE. For more information on the search order for REXX external routines, see “Search Order” on page 60.

Entry Specifications

The contents of the registers on entry to ARXERS are:

Register 0
Address of an environment block (optional)

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 for ARXERS

You can pass the address of an environment block in register 0. In register 1, the compiler runtime processor must 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 five parameters are required. The addresses that point to parameter 6 and parameter 7 are optional. If ARXERS does not find the high-order bit set on in the address for parameter 5 or (optional parameters) 6 or 7, ARXERS does not invoke the specified routine and returns with a return code of 32 in register 15. See Table 108 on page 520 for more information on return codes. Set the high-order bit of the last address in the parameter list to 1. Table 107 on page 519 lists the parameters for the external routine search routine.

Table 107. Parameters for the External Routine Search Routine

Parameter	Number of Bytes	Description
Parameter 1	8	Function requested. On entry to ARXERS, this parameter contains the function requested of the external routine search routine. The function specification must be in uppercase, left-justified, and padded on the right with blanks. Acceptable values are: **"EXTSUB "** Specifies that the external routine that is being requested is a subroutine. The subroutine is not required to return an EVALBLOK. For a successfully run subroutine that does not return an EVALBLOK, the EVALBLOK address is set to 0 and the return code is set to 0. **"EXTBRSUB"** Specifies that the external routine that is being requested is a subroutine and receives control through a branch instruction. The external routine is invoked using standard register linkage. See Table 21 on page 345 for more information. **"EXTFCT "** Specifies that the external routine that is being requested is a function. The function is required to return an EVALBLOK. For a successfully run function that does not return an EVALBLOK, the EVALBLOK address is set to 0 and the return code is set to 4. **"EXTBRFCT"** Specifies that the external routine that is being requested is a function and receives control through a branch instruction. The external routine is called using standard register linkage conventions. See Table 21 on page 345 for more information.
Parameter 2	4	Address of the external routine name. For the "EXTFCT " and "EXTSUB " functions, this parameter specifies the address of the external routine name for the requested external routine. The name must not include the opening left parenthesis that identifies the routine as a function, if that is the type of routine being called. For the "EXTBRFCT" and "EXTBRSUB" functions, this parameter specifies the address of the external routine that is to be given control. ARXERS branches to this address after building the parameter list for the specified routine.
Parameter 3	4	Length of the external routine name. Specifies the length of the external routine name to which parameter 2 points. ARXERS ignores this parameter if parameter 1 is "EXTBRFCT" or "EXTBRSUB".
Parameter 4	4	Address of the arguments for the external routine. Specifies the address of a set of address/length pairs that hold the arguments for the external routine. These arguments must be in the format an external routine expects. (See “Interface for Writing External Function and Subroutine Code” on page 344 for a description of the argument list format.)
Parameter 5	4	Address of an EVALBLOK. On return from ARXERS, this parameter contains the address of an EVALBLOK (if any) that ARXERS returned after an external routine successfully completed. An address of 0 indicates that ARXERS did not receive an EVALBLOK.

Table 107. Parameters for the External Routine Search Routine (continued)

Parameter	Number of Bytes	Description
Parameter 6	4	The address of a REXX environment block is optional. This is the address of the REXX environment block under which the request is to be performed. If the compiler runtime processor supplies a nonzero parameter, ARXERS considers this parameter to be a valid environment block address. If you omit this parameter or it is 0, ARXERS obtains the environment block address from register 0. See "Using the Environment Block Address" on page 441 for more information about this.
Parameter 7	4	Return code. The return code parameter is optional. Upon return from ARXERS, this parameter contains the return code for ARXERS. Register 15 contains the same value as this parameter (if used).

Return Specifications

On return from the external routine search routine, the contents of the registers are:

Registers 0-14
Same as on entry

Register 15
Return code.

Return Codes

Table 108 on page 520 lists the return codes issued by the external routine search routine.

Table 108. Return Codes from the External Routine Search Routine

Return Code (Decimal)	Description
0	Processing was successful. ARXERS located the external routine, and the external routine returned control with a return code of 0 in register 15. If you specified EXTFCT or EXTBRFCT, the address of the EVALBLOK is available in parameter 5.
4	Processing was successful. ARXERS located the external routine, and the external routine returned control with a return code of 0 in register 15. However, you specified EXTFCT or EXTBRFCT, and the external routine returned no EVALBLOK.
8	Processing was successful. ARXERS located the external routine, and the external routine returned with a nonzero return code in register 15.
12	Processing was not successful. ARXERS attempted to create an EVALBLOK, but insufficient virtual storage was available.
16	Processing was not successful. ARXERS could not locate the specified routine.
20	Processing was not successful. An error message may accompany this return code.
28	Processing was not successful. ARXERS was unable to locate a language processor environment. Verify that you passed a valid environment block address.
32	Processing was not successful. The parameter list is incorrect. The parameter list contains either too few or too many parameters, or the high-order bit of the last address in the list is not set to 1 to indicate the end of the parameter list.

Host Command Search Routine (ARXHST)

ARXHST is a programming routine that searches for and runs a host command. ARXHST allows a compiler runtime processor to pass control to a host command through a direct interface. A compiler runtime processor that uses ARXHST leaves the implementation of the host command search and invocation to REXX/VSE. For more information on the search order for REXX external routines, see “Search Order” on page 60.

ARXHST also allows a compiler runtime processor to set and clear the ETMODE flag, based on the OPTIONS ETMODE or OPTIONS NOETMODE instruction. The "ETMODE" function of ARXHST sets the ETMODE flag. The "NOETMODE" function of ARXHST clears the ETMODE flag. For more information on OPTIONS ETMODE and OPTIONS NOETMODE, see the OPTIONS instruction on page “OPTIONS” on page 43.

Entry Specifications

The contents of the registers on entry to ARXHST are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters for ARXHST

In register 1, the compiler runtime processor must 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 six parameters are required. The addresses that point to parameter 7 and parameter 8 are optional. If ARXHST does not find the high-order bit set on in the address for parameter 6 or (optional parameters) 7 or 8, ARXHST does not invoke the specified routine and returns with a return code of 32 in register 15. See Table 110 on page 523 for more information on return codes. The high-order bit of the last address in the parameter list must be set to 1. Table 109 on page 521 lists the parameters for the host command search routine.

Table 109. Parameters for the Host Command Search Routine

Parameter	Number of Bytes	Description
Parameter 1	8	Function requested. On entry to ARXHST, this parameter contains the function requested of the host command search routine. The function name must be in uppercase, left-justified, and padded on the right with blanks. Acceptable values are: **"HOSTCMD "** Specifies ARXHST searches for and invokes a host command. **"ETMODE"** Specifies that ARXHST sets the ETMODE flag. **"NOETMODE"** Specifies that ARXHST clears the ETMODE flag.

Table 109. Parameters for the Host Command Search Routine (continued)

Parameter	Number of Bytes	Description
Parameter 2	8	Host command environment name. Specifies the name of the host command environment that is in effect for the compiled REXX program that is running. The name must be in uppercase, left-justified, and padded on the right with blanks. The name should correspond to an entry in the host command environment table. Use this parameter only for the "HOSTCMD" function. For the "ETMODE" or "NOETMODE" functions, set this parameter to blanks.
Parameter 3	4	Address of host command string. Specifies the address of a string for the host command environment to run. ARXHST passes the string as is to the host command environment routine that corresponds to the host command environment specified in parameter 2. The program that calls ARXHST must manage (allocate and free) storage for the command buffer. Use this parameter only for the "HOSTCMD" function. For the "ETMODE" or "NOETMODE" functions, set this parameter to 0.
Parameter 4	4	Host command string length. Specifies the length of the command string to which the address in parameter 3 points. Use this parameter only for the "HOSTCMD" function. For the "ETMODE" or "NOETMODE" functions, set this parameter to 0.
Parameter 5	4	Command output buffer address. Specifies the address of an area to hold the result of the command. This result is a character representation of the binary return code the host command issues. It is recommended that this area be 20 bytes. If parameter 2 is not defined in the host command environment table, ARXHST returns the character representation of -3. The compiler runtime processor that calls ARXHST should properly set the REXX special variable RC. Use this parameter only for the "HOSTCMD" function. For the "ETMODE" or "NOETMODE" functions, set this parameter to 0.
Parameter 6	4	Output area length. Specifies the length of the output area to which the address in parameter 5 points. Use this parameter only for the "HOSTCMD" function. For the "ETMODE" or "NOETMODE" functions, set this parameter to 0.
Parameter 7	4	The address of a REXX environment block is optional. This is the address of the REXX environment block under which the request is to be performed. If the compiler runtime processor supplies a nonzero parameter, ARXHST considers this parameter to be a valid environment block address. If you omit this parameter or it is 0, ARXHST obtains the environment block address from register 0. See “Using the Environment Block Address” on page 441 for more information about this.
Parameter 8	4	The requested function return code is optional. On return from ARXHST, this parameter contains the return code for ARXHST. See Table 110 on page 523 for information on return codes the host command search routine issues. Register 15 contains the same value as this parameter (if used).

Return Specifications

On return from the host command search routine, the contents of the registers are:

Registers 0-14
Same as on entry

Register 15
Return code.

Return Codes

Table 110 on page 523 lists the return codes the host command search routine issues.

Table 110. Return Codes from the Host Command Search Routine

Return Code (Decimal)	Description
0	Processing was successful. For the "HOSTCMD" function, ARXHST located the host command and the host command returned with a return code of 0 in register 15. For the "ETMODE" and "NOETMODE" functions, ARXHST set or cleared the ETMODE flag successfully.
20	Processing was not successful. For the "HOSTCMD" function, ARXHST could not locate the specified host command. ARXHST returns -3 in the command output buffer. The command string specified in parameters 3 and 4 is incorrect, the requested function could not be located in the search order, or the host command environment table did not define the host command environment routine. This return code could also indicate passing an incorrect function (parameter 1) to ARXHST. Valid functions are "HOSTCMD", "ETMODE", and "NOETMODE".
28	Processing was not successful. ARXHST could not locate a language processor environment. The command output area is not modified. Verify that you passed a valid environment block address.
32	Processing was not successful. The parameter list is incorrect. The parameter list contains too few or too many parameters, or the high-order bit of the last address in the list is not set to 1 to indicate the end of the parameter list. The command output buffer is not modified.
nn	Processing was successful. The specified host command environment routine returned a nonzero return code. The return code from the host command environment routine is *nn*.

Exit Routing Routine (ARXRTE)

ARXRTE is a programming routine that locates and invokes a REXX exit. ARXRTE provides a way for a compiler runtime processor to invoke REXX exit routines. A compiler runtime processor that uses ARXRTE leaves the implementation of exit routing to REXX/VSE. For information about REXX exit routines, see “REXX Exit Routines” on page 468.

Entry Specifications

The contents of the registers on entry to ARXRTE are:

Register 0
Address of an environment block (optional)

Register 1
Address of the parameter list the caller passes

Registers 2-12
Unpredictable

Register 13
Address of a register save area

Register 14
Return address

Register 15
Entry point address

Parameters for ARXRTE

In register 1, the compiler runtime processor must 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 three parameters are required. The addresses that point to parameter 4 and parameter 5 are optional. If ARXRTE does not find the high-order bit set on in either the address for parameter 3 or (optional parameters) 4 or 5, ARXRTE does not invoke the specified routine and returns with a return code of 32 in register 15. See Table 112 on page 525 for more information on return codes. To end the parameter list, set the high-order bit of the last address to 1. Table 111 on page 524 lists the parameters for the exit routing routine.

Table 111. Parameters for the Exit Routing Routine

Parameter	Number of Bytes	Description
Parameter 1	8	Function requested. On entry to ARXRTE, this parameter contains the function requested of the exit routing routine. The function name must be in uppercase, left-justified, and padded on the right with blanks. Acceptable values are: **"EXECINIT"** Specifies running the EXECINIT exit routine. For more information on the EXECINIT exit, see "Exec Initialization and Termination Exits" on page 473. **"EXECTERM"** Specifies running the EXECTERM exit routine. For more information on the EXECTERM exit, see "Exec Initialization and Termination Exits" on page 473.
Parameter 2	8	Exit routine parameter list address. Specifies the address of the parameter list for the requested exit routine. If the exit does not require parameters, the address in this parameter must be set to 0. For a discussion of the parameters for the specified exit, see "REXX Exit Routines" on page 468.
Parameter 3	4	Exit routine return code. On return from ARXRTE, this parameter contains the return code value from the requested exit. This value has meaning only if the return code from ARXRTE is 0.
Parameter 4	4	The address of a REXX environment block is optional. This is the address of the REXX environment block under which the request is to be performed. If the compiler runtime processor supplies a nonzero parameter, ARXRTE considers this parameter to be a valid environment block address. If you omit this parameter or it is 0, ARXRTE obtains the environment block address from register 0. (See Chapter 20, "Initialization and Termination Routines," on page 427.)
Parameter 5	4	Return code. The return code parameter is optional. On return from ARXRTE, this parameter contains the return code for ARXRTE. Register 15 will contain the same value as this parameter (if used).

Return Specifications

On return from the exit routing routine, the contents of the registers are:

Registers 0-14
Same as on entry

Register 15
Return code.

Return Codes

Table 112 on page 525 lists the return codes the exit routing routine issues.

Table 112. Return Codes from the Exit Routing Routine

Return Code (Decimal)	Description
0	Processing was successful. ARXRTE located the exit, passed control to the exit, and the exit ran to completion. The return code from the exit is available in parameter 3.
4	Processing was not successful. The module name table for the current environment did not have an entry for the requested exit. Verify that the environment block address specified in parameter 4 is correct and that the module name table contains the name of the exit you specified.
20	Processing was not successful. The error may occur because: • A compiled program is not executing • The requested function is not supported.
28	Processing was not successful. A language processor environment could not be located. Verify that the environment block address specified in parameter 4 is correct.
32	Processing was not successful. The parameter list contained too few or too many parameters, or the high-order bit of the last parameter was not set to 1 to indicate the end of the parameter list.

Appendix A. List of the Names of Macros Intended for Customers' Use

The macros identified in this appendix are provided as programming interfaces for customers of REXX/VSE.

Warning: Do not use as programming interfaces any REXX/VSE macros other than those identified in this appendix.

General-Use Programming Interfaces

The macros listed in this topic are general-use programming interfaces intended for customer use. Some macros have keywords, fields, or parameters that are designed for IBM internal use only. Such keywords, fields, or parameters are not part of the programming interfaces for use by customers in writing programs that request or receive the services of REXX/VSE. Please refer to the appropriate product documentation for the correct classification and use of these keywords, fields, or parameters.

Mapping Macros

This section lists the general-use programming interface mapping macros for REXX/VSE. The data areas are programming interfaces or contain fields that are programming interfaces. (The macro ID is the member name in PRD1.BASE; the acronym identifies the control block and is typically the prefix of each field in the control block.)

Table 113. Mapping Macros

Macro ID	Acronym
ARXARGTB	ARGTABLE
ARXDSIB	DSIB
ARXEFPL	EFPL
ARXENVB	ENVBLOCK
ARXEVALB	EVALBLOCK
ARXEXECB	EXECBLK
ARXEXTE	ARXEXTE
ARXFPDIR	FPCKDIR
ARXINSTB	INSTBLK
ARXMODNT	MODNAMET
ARXPACKT	PACKTB
ARXPARMB	PARMBLOCK
ARXSHVB	SHVBLOCK
ARXSUBCT	SUBCOMTB
ARXWORKB	WORKBLOK

Product-Sensitive Programming Interfaces

Macros listed in this topic are product-sensitive programming interfaces intended for customer use. Macros can have keywords, fields, or parameters that are designed for IBM internal use only. Such keywords, fields, or parameters are not part of the programming interfaces for use by customers in writing programs that request or receive the services of REXX/VSE. Please refer to the appropriate product documentation for the correct classification and use of these keywords, fields, or parameters.

Mapping Macros

This section lists the product-sensitive programming interface mapping macros for REXX/VSE. The data areas are programming interfaces or contain fields that are programming interfaces.

Table 114. Mapping Macros

Macro ID	**Acronym**
ARXCMPTB	—
ARXENVB	ENVBLOCK
ARXEXTE	ARXEXTE
ARXWORKB	WORKBLOK

Appendix B. Servicing REXX/VSE

When applying a Program Temporary Fix (PTF), check if mandatory or recommended phases are affected which had been loaded into the SVA during IPL.

Mandatory phases are:

Table 115. Mandatory Phases

Phase Name	Approximate Size	Residency Mode
ARXREXX	2080	ANY
ARXINIT	361.088	ANY
ARXRXVEC	920	ANY
EAGRTXIN	274.832	ANY
EAGRTXLD	304	ANY
EAGRTPRC	304	ANY
EAGRTXTR	312	ANY
EAGRTXVH	304	ANY

Recommended phases are

Table 116. Recommended Phases

Phase Name	Approximate Size	Residency Mode
ARXIOLAR	7424	24
ARXSTO00	23104	ANY
ARXSTAM	17096	ANY
ARXRX24.	5344	24

If these phases have been affected by a PTF they can be made active by running job ARXINST.Z. This job calls loadlist $SVAREXX and loads the mandatory and recommended phases listed above into the SVA. Make sure to have **600KB** of SVA storage.

```
* $$ JOB JNM=ARXINST,DISP=D,CLASS=0
// JOB ARXINST       LOAD REXX INTO THE SVA
* *-------------------------------------------------------------*
* * PART 1:  Load REXX/VSE via $SVAREXX                        *
* *-------------------------------------------------------------*
// LIBDEF PHASE,SEARCH=PRD1.BASE
SET SDL
LIST=$SVAREXX
/*
* *-------------------------------------------------------------*
* * PART 2:  VERIFY THAT THE MANDATORY AND RECOMMENDED         *
* *          REXX/VSE PHASES HAVE BEEN LOADED INTO THE SVA     *
* *-------------------------------------------------------------*
// EXEC ARXVERFY
* *-------------------------------------------------------------*
* * PART 3:  INITIALIZE REXX/VSE TABLES                        *
* *-------------------------------------------------------------*
// EXEC ARXLINK
/&
* $$ EOJ
```

Figure 34. Initializing REXX/VSE using ARXINST.Z

In order to load single phases into the SVA, replace *LIST=$SVAREXX* by the name of the affected phase and run **part 1** of ARXINST.Z.

To replace, for example, phase EAGRTXIN you would code the following:

```
* $$ JOB JNM=RELOAD,DISP=D,CLASS=0
// JOB RELOAD        LOAD EAGRTXIN INTO THE SVA
// LIBDEF PHASE,SEARCH=PRD1.BASE
SET SDL
EAGRTXIN,SVA
/*
/&
* $$ EOJ
```

Figure 35. Loading Single Phases into the SVA

Appendix C. REXX Supplied Link Books

REXX allows you to write exit routines that may replace IBM supplied programs. In order to activate your exit routines you need to relink ARXINIT together with your object decks.

The sample job ARXSKLNK.Z in PRD1.BASE shows you how to link the REXX phase ARXINIT. The job includes a link book with the name ARXINLNK.OBJ. You can also use the skeleton ARXSKLNK.Z to link other REXX phases. The following is the list of supplied link books:

LINKBOOK	PHASE
ARXCONA1:	ARXCONAD
ARXCPRO1:	ARXCPROF
ARXDILK1:	ARXDI01
ARXDILNK:	ARXDI02, ARXDI03, ARXDI04, ARXDI05, ARXDI06, ARXDI07, ARXDI08, ARXDI09, ARXDI0A, ARXDI0B, ARXDI0C, ARXDI0D, ARXDI0E
ARXEFCO1:	ARXEFCO
ARXEFSO1:	ARXEFSO
ARXEFSN1:	ARXEFSN
ARXENPL0:	ARX00ENP
ARXENPL1:	ARX01ENP
ARXENPL2:	ARX02ENP
ARXENPL3:	ARX03ENP
ARXENPL4:	ARX04ENP
ARXENPL5:	ARX05ENP
ARXENPL6:	ARX06ENP
ARXENPL7:	ARX07ENP
ARXENPL8:	ARX08ENP
ARXENPL9:	ARX09ENP
ARXENUL1:	ARX01ENU
ARXENUL2:	ARX02ENU
ARXENUL3:	ARX03ENU
ARXENUL4:	ARX04ENU
ARXENUL5:	ARX05ENU
ARXENUL6:	ARX06ENU
ARXENUL7:	ARX07ENU
ARXENUL8:	ARX08ENU
ARXENUL9:	ARX09ENU

LINKBOOK	PHASE
ARXENUL0:	ARX00ENU
ARXINLNK:	ARXINIT
ARXREXX1:	ARXREXX
ARXANCR1:	ARXANCHR
ARXCMPT1:	ARXCMPTM
ARXPARM1:	ARXPARMS
ARXIOLA1:	ARXIOLAR
ARXFLOC1:	ARXFLOC
ARXFUSE1:	ARXFUSER
ARXSTAM1:	ARXSTAM
ARXEMSG1:	ARXEMSG
ARXLINKL:	ARXLINK
ARXINTL:	ARXINT
ARXLOADL:	ARXLOAD
ARXLDL:	ARXLD
ARXSUBCL:	ARXSUBCM
ARXSUBL:	ARXSUB
ARXEXC1:	ARXCEXEC
ARXEXECL:	ARXEXEC
ARXEXL:	ARXEX
ARXINOUL:	ARXINOUT
ARXIOL:	ARXIO
ARXJCLL:	ARXJCL
ARXRLFL:	ARXRLT
ARXSTKL:	ARXSTK
ARXTRML1:	ARXTERM
ARXTRML2:	ARXTRM
ARXICLNK:	ARXIC
ARXUIDL:	ARXUID
ARXTERML:	ARXTERMA
ARXTMAL:	ARXTMA
ARXMSGIL:	ARXMSGID
ARXMIDL:	ARXMID
ARXEXCOL:	ARXEXCOM
ARXEXCL:	ARXEXC

LINKBOOK	PHASE
ARXSAYL:	ARXSAY
ARXERSL:	ARXERS
ARXHSTL:	ARXHST
ARXHLTL:	ARXHLT
ARXTXTL:	ARXTXT
ARXLINL:	ARXLIN
ARXRTEL:	ARXRTE
ARXRXVEL:	ARXRXVEC
ARXRX24L:	ARXRX24
ARXLNK04:	ARXVERFY
ARXLNK05:	ARXEFVSE
ARXSTOLK:	ARXSTO00
ARXSYSL1:	ARXSYSLN
ARXEOJ1:	ARXEOJTB
ARXJCL2:	ARXJCLAD
ARXIDCM1:	ARXIDCAM
ARXLIBRI:	ARXLIBR
ARXOCXI1:	ARXOCXIT
ARXOUTL:	ARXOUT
REXXLOA1:	REXXLOAD

Bibliography

This bibliography lists some publications that provide additional information about REXX or the VSE/ESA system.

- REXX/VSE User's Guide
- VSE/ESA REXX/VSE Diagnosis Reference
- z/VSE System Macros User's Guide
- z/VSE Guide to System Functions
- z/VSE System Control Statements
- VSE/POWER Administration and Operation
- VSE/POWER Application Programming
- *VSE/ESA Library Guide, GC33-6619*
- z/VSE Messages and Codes, Volume 1
- z/VSE Installation
- *SAA Common Programming Interface REXX Level2 Reference*, SC24-5549
- *IBM Compiler and Library for SAA REXX/370 Release 2: Introducing the Next Step in REXX Programming*, G511-1430
- IBM Compiler and Library for SAA REXX/370, User's Guide and Reference
- IBM Compiler and Library for SAA REXX/370 Release 2 Diagnosis Guide

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Product Number: 5686–066

SC33-6642-11