z/VM
Version 7 Release 1

REXX/VM Reference

IBM

Note:

Before you use this information and the product it supports, read the information in "Notices" on page 311.

Contents

List of Figures

List of Tables

About This Document

This is a reference document containing all of the REXX/VM instructions and functions. They are listed alphabetically in their own sections. Also included are details about general concepts you need to know in order to program in REXX, as well as details about parsing, math functions, condition trapping, system interfaces, and debug aids. You will need a terminal with access to IBM z/VM, and you should be reasonably familiar with z/VM, but you need not have had any previous programming experience.

The programming language described by this document is called the REstructured eXtended eXecutor language (abbreviated REXX). The document also describes how the z/VM REXX/VM language processor (shortened, hereafter, to the language processor) processes or *interprets* the REstructured eXtended eXecutor language.

Intended Audience

This document is intended for experienced programmers, particularly those who have used a block-structured, high-level language (for example, PL/I, Algol, or Pascal), who need to refer to REXX/VM instructions and functions and need to learn more details about items such as parsing.

Descriptions include the use and syntax of the language and explain how the language processor "interprets" the language as a program is running.

You should have read the *z/VM: REXX/VM User's Guide* to learn how to program in REXX. If you are new to REXX and to programming in general, read the *z/VM: REXX/VM User's Guide*.

Syntax, Message, and Response Conventions

The following topics provide information on the conventions used in syntax diagrams and in examples of messages and responses.

How to Read Syntax Diagrams

Special diagrams (often called *railroad tracks*) are used to show the syntax of external interfaces.

To read a syntax diagram, follow the path of the line. Read from left to right and top to bottom.

- The ►►── symbol indicates the beginning of the syntax diagram.
- The ──► symbol, at the end of a line, indicates that the syntax diagram is continued on the next line.
- The ►── symbol, at the beginning of a line, indicates that the syntax diagram is continued from the previous line.
- The ──►◄ symbol indicates the end of the syntax diagram.

Within the syntax diagram, items on the line are required, items below the line are optional, and items above the line are defaults. See the examples in Table 1 on page xvi.

Table 1: Examples of Syntax Diagram Conventions
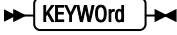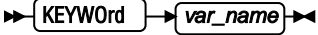
Syntax Diagram Convention	Example
Keywords and Constants A keyword or constant appears in uppercase letters. In this example, you must specify the item KEYWORD as shown. In most cases, you can specify a keyword or constant in uppercase letters, lowercase letters, or any combination. However, some applications may have additional conventions for using all-uppercase or all-lowercase.	▸▸─[KEYWORD]─◂◂
Abbreviations Uppercase letters denote the shortest acceptable abbreviation of an item, and lowercase letters denote the part that can be omitted. If an item appears entirely in uppercase letters, it cannot be abbreviated. In this example, you can specify KEYWO, KEYWOR, or KEYWORD.	▸▸─[KEYWOrd]─◂◂
Symbols You must specify these symbols exactly as they appear in the syntax diagram.	***** Asterisk **:** Colon **,** Comma **=** Equal Sign **-** Hyphen **( )** Parentheses **.** Period
Variables A variable appears in highlighted lowercase, usually italics. In this example, *var_name* represents a variable that you must specify following KEYWORD.	▸▸─[KEYWOrd]─▸[*var_name*]─◂◂

Table 1: Examples of Syntax Diagram Conventions (continued)
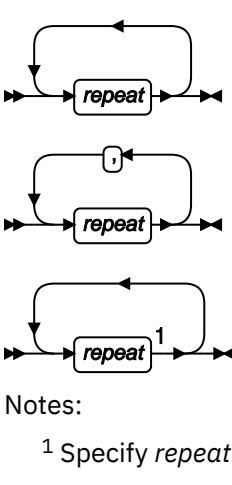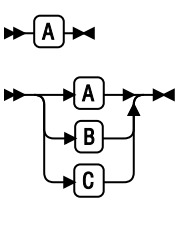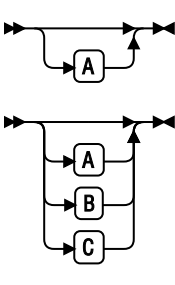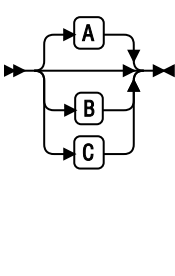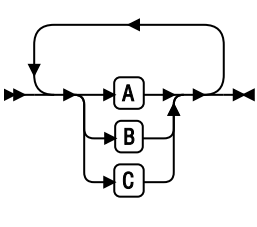
Syntax Diagram Convention	Example
Repetitions An arrow returning to the left means that the item can be repeated. A character within the arrow means that you must separate each repetition of the item with that character. A number (1) by the arrow references a syntax note at the bottom of the diagram. The syntax note tells you how many times the item can be repeated. Syntax notes may also be used to explain other special aspects of the syntax.	 Notes: [1] Specify *repeat* up to 5 times.
Required Item or Choice When an item is on the line, it is required. In this example, you must specify A. When two or more items are in a stack and one of them is on the line, you must specify one item. In this example, you must choose A, B, or C.	
Optional Item or Choice When an item is below the line, it is optional. In this example, you can choose A or nothing at all. When two or more items are in a stack below the line, all of them are optional. In this example, you can choose A, B, C, or nothing at all.	
Defaults When an item is above the line, it is the default. The system will use the default unless you override it. You can override the default by specifying an option from the stack below the line. In this example, A is the default. You can override A by choosing B or C.	
Repeatable Choice A stack of items followed by an arrow returning to the left means that you can select more than one item or, in some cases, repeat a single item. In this example, you can choose any combination of A, B, or C.	

Table 1: Examples of Syntax Diagram Conventions (continued)	
Syntax Diagram Convention	**Example**
Syntax Fragment Some diagrams, because of their length, must fragment the syntax. The fragment name appears between vertical bars in the diagram. The expanded fragment appears in the diagram after a heading with the same fragment name. In this example, the fragment is named "A Fragment."	 A Fragment

Examples of Messages and Responses

Although most examples of messages and responses are shown exactly as they would appear, some content might depend on the specific situation. The following notation is used to show variable, optional, or alternative content:

xxx
> Highlighted text (usually italics) indicates a variable that represents the data that will be displayed.

[ ]
> Brackets enclose optional text that might be displayed.

{ }
> Braces enclose alternative versions of text, one of which will be displayed.

|
> The vertical bar separates items within brackets or braces.

...
> The ellipsis indicates that the preceding item might be repeated. A vertical ellipsis indicates that the preceding line, or a variation of that line, might be repeated.

Where to Find More Information

You can find useful information in the *z/VM: REXX/VM User's Guide* and through the online z/VM HELP Facility available with z/VM. For any program written in the REstructured eXtended eXecutor (REXX) language, you can get information on how the language processor interprets the program or a particular instruction by using the REXX TRACE instruction.

For more information about VM and REXX, see the documents in the "Bibliography" on page 315.

Links to Other Documents and Websites

The PDF version of this document contains links to other documents and websites. A link from this document to another document works only when both documents are in the same directory or database, and a link to a website works only if you have access to the Internet. A document link is to a specific edition. If a new edition of a linked document has been published since the publication of this document, the linked document might not be the latest edition.

How to Send Your Comments to IBM

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

To send us your comments, go to z/VM Reader's Comment Form (www.ibm.com/systems/campaignmail/z/zvm/zvm-comments) and complete the form.

If You Have a Technical Problem

Do not use the feedback method. Instead, do one of the following:

- Contact your IBM service representative.
- Contact IBM technical support.
- See IBM: z/VM Support Resources (www.ibm.com/vm/service).
- Go to IBM Support Portal (www.ibm.com/support/entry/portal/Overview).

Summary of Changes for z/VM REXX/VM Reference

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

SC24-6314-00, z/VM Version 7 Release 1

This edition supports the general availability of z/VM V7.1.

SC24-6221-02, z/VM Version 6 Release 4

This edition includes changes to support the general availability of z/VM V6.4.

Chapter 1. REXX General Concepts

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

- Command procedures
- Application front ends
- User-defined macros (such as editor subcommands)
- User-defined XEDIT subcommands
- Prototyping
- Personal computing.

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.

There is no limit to the length of the values of variables, as long as all variables fit into the storage available.

Implementation Maximum: No single request for storage can exceed the fixed limit of 16MB. This limit applies to the size of a variable plus any control information. It also applies to buffers obtained to hold numeric results.

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

You can use compound symbols, such as

```
NAME.Y.Z
```

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

REXX programs can reside in CMS Shared File System (SFS) directories or on minidisks. REXX programs usually have a file type of EXEC. These files can contain CP and CMS commands. REXX programs with a file type of XEDIT can also contain XEDIT subcommands.

A language processor (interpreter) runs REXX programs. That is, the program is processed line-by-line and word-by-word, without first being translated to another form (compiled). The advantage of this to the user is that if the program fails with a syntax error of some kind, the point of error is clearly indicated; usually, it will not take long to understand the difficulty and make a correction.

Structure and General Syntax

Programs written in the REstructured eXtended eXecutor (REXX) language must start with a comment. (This distinguishes them from CMS EXEC and EXEC 2 language programs.)

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 3)
- 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 "Tokens" on page 3) 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 Appendix B, "Double-Byte Character Set (DBCS) Support," on page 279.

Comments

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

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

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

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

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

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

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

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

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

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

```
01    parse pull input
02 /* if substr(input,1,5) = '/' || '*123'
```

```
03      then call process
04 */ dept = substr(input,32,5)
```

For information about Double-Byte Character Set characters, see Appendix B, "Double-Byte Character Set (DBCS) Support," on page 279 and the OPTIONS instruction in "OPTIONS" on page 46.

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. See the note in Chapter 1, "REXX General Concepts," on page 1 for more information.

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.

Also note that in Assembler language hexadecimal numbers are represented with the X in front of the number. REXX only accepts hexadecimal numbers as was described in the previous note. In this book you will see hexadecimal numbers represented in both ways, but when you are coding a hexadecimal string in REXX, place the X after the number.

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                 /* == ''                  */
```

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

Symbols:
Symbols are groups of characters, selected from the:

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

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

These are valid symbols:

```
Fred
Albert.Hall
WHERE?
<.H.E.L.L.O>                /* This is DBCS */
```

For information about Double-Byte Character Set (DBCS) characters, see Appendix B, "Double-Byte Character Set (DBCS) Support," on page 279.

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.

---

[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 numbers in exponential notation:

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

Implementation maximum: A symbol can consist of up to 250 characters. But note that its value, if it is a variable, is limited only by the amount of storage available. See the note in Chapter 1, "REXX General Concepts," on page 1 for more information.

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 5, "Numbers and Arithmetic," on page 153 - "Errors" on page 160 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 '
'0E000'
```

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 7). 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 3, "Functions," on page 67), which are of the form:
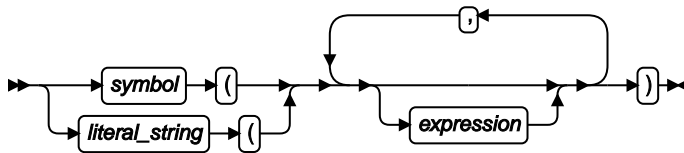
Evaluation of an expression is left to right, modified by parentheses and by operator precedence in the usual algebraic manner (see "Parentheses and Operator Precedence" on page 10). Expressions are wholly evaluated, unless an error occurs during evaluation.

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 a 16MB limitation on the amount of a single storage request available to the language processor. See the note in Chapter 1, "REXX General Concepts," on page 1 for more information.

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 3) 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 5, "Numbers and Arithmetic," on page 153 for details about precision, the format of valid numbers, and the operation rules for arithmetic. Note that if an arithmetic result is shown in exponential notation, it is likely that rounding has occurred.

Comparison

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

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

For all the other comparison operators, if *both* terms involved are numeric, a numeric comparison (in which leading zeros are ignored, and so forth—see "Numeric Comparisons" on page 157) 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.

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):

Operators	Description
+ - ¬ \	(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
- Labels
- Instructions
- Assignments
- Keyword instructions
- Commands.

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.

You can use DBCS characters. See Appendix B, "Double-Byte Character Set (DBCS) Support," on page 279 for more information.

Note: Labels within a WHEN/THEN/DO construct will not be traced.

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 13.

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, or the variable pool interface, 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. Currently, on VM 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=''`.

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 14), 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
<.D.A.T.E>
```

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.
<.F.R.E.D>.<.A.B>
```

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"   */

e = 'barney'        /*   sets E to lower case 'barney'   */
d.barney = 6        /*   sets D.BARNEY to 6              */
d.e = 7             /*   sets D.barney to 7             */
f = e               /*   sets F to 'barney'             */
say d.barney        /*   displays 6 , value of D.BARNEY  */
say d.f             /*   displays 7 , value of D.barney  */
```

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.
<.A.B>.
```

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 36), and PROCEDURE EXPOSE FRED. exposes *all possible* variables with that stem (see "PROCEDURE" on page 51).

1. When the ARG, PARSE, or PULL instruction or the VALUE built-in function or the variable pool interface 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 17), a possible ambiguity is resolved by the following rule: any clause that starts with a symbol and whose second token is (or starts with) an equal sign (=) is an **assignment**, rather than an expression (or a keyword instruction). This is not a restriction, because you can ensure the clause is processed as a command in several ways, such as by putting a null string before the first name, or by enclosing the first part of the expression in parentheses.

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

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

   is an assignment, not an ADDRESS instruction.

3. You can use the SYMBOL function (see "SYMBOL" on page 107) 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 Chapter 6, "Conditions and Condition Traps," on page 161).

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 environment for processing commands. An environment is selected by default on entry to a REXX program. You can change the environment by using the ADDRESS instruction. You can find out the name of the current environment by using the ADDRESS built-in function. The underlying operating system defines environments external to the REXX program.

The environment selected depends on the caller; for example if a program is called from CMS, the default environment is CMS. If called from an editor that accepts subcommands from the language processor, the default environment may be that editor.

You can also write a REXX program that issues editor subcommands and run your program during an editing session. Your program can inspect the file being edited, issue subcommands to make changes, test return codes to check that the subcommands have been processed as you expected, and display messages to the user when appropriate. The user can call your program by entering its name on the editor's command line. For a discussion of this mechanism see "Issuing Subcommands from Your Program" on page 20.

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 underlying system. 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 164.

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. (For example, a locate command to an editing system might report `requested string not found` as an error.) A **failure** is a condition raised by a command for which a program that uses that command would *not* usually be expected to recover (for example, a command that is not executable or cannot be found).

Errors and failures in commands can affect REXX processing if a condition trap for ERROR or FAILURE is ON (see Chapter 6, "Conditions and Condition Traps," on page 161). 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 61.

Here is an example of submitting a command. If the underlying system is CMS, the sequence:

```
filename = "JACK"; filetype = "RABBIT"
"STATE" filename filetype "A"
```

results in the string STATE JACK RABBIT A being submitted to CMS. The simpler expression:

```
"STATE JACK RABBIT A"
```

has the same effect.

On return, the return code placed in RC has the value 0 if the file JACK RABBIT A exists, or 28 if it does not.

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

Examples:

```
"erase * listing"       /* * does not mean "multiplied by"  */

"load" prog1 "(start"   /* ( is not unmatched parenthesis   */


a = any
"access 192 b/a"        /* / does not mean "divided by"     */
```

The CMS Environment

When the environment selected is CMS (which is the default for execs), the command is called exactly as if it had been entered from the command line (but cleanup after the command has completed is different). See . The language processor creates two parameter lists:

- The result of the expression, tokenized and translated to uppercase, is placed in a Tokenized Parameter List.
- The result of the expression, unchanged, is placed in an Extended Parameter List.

The language processor then asks CMS to process the command. This is the same search order that is used for a command entered from the CMS interactive command environment. The first token of the command is taken as the command name. As soon as the command name is found, the search stops and the command is processed.

The search order is:

1. Search for an exec with the specified command name:

   a. Search for an exec in storage. If an exec with this name is found, CMS determines whether the exec has a USER, SYSTEM, or SHARED attribute. If the exec has the USER or SYSTEM attribute, it is run.

      If the exec has the SHARED attribute, the INSTSEG setting of the SET command is checked. When INSTSEG is ON, all accessed directories and minidisks are searched for an exec with that name. (To find a file in a directory, read authority is required on both the file and the directory.) If an exec is found, the file mode of the EXEC is compared to the file mode of the CMS installation saved segment. If the file mode of the saved segment is equal to or higher (closer to A) than the file mode of the directory or minidisk, then the exec in the saved segment is run. Otherwise, the exec in the directory or on the minidisk is run. However, if the exec is in a directory and the file is locked, the processing fails with an error message.

   b. Search the table of active (open) files for a file with the specified command name and a file type of EXEC. If more than one open file is found, the one opened first is used.

   c. Search for a file with the specified command name and a file type of EXEC on any currently accessed disk or directory, using the standard CMS search order (A through Z).

      Note: To find a file in a directory, read authority is required on both the file and the directory. If the file is in a directory and the file is locked, the processing fails with an error message.

2. Search for a translation or synonym for the command name. If found, search for an exec with the valid translation or synonym by repeating Step 1. (For a description of the translation tables, see the SET

TRANSLATE command in the *z/VM: CMS Commands and Utilities Reference*. For a description of the synonym tables, see the SYNONYM command in the *z/VM: CMS Commands and Utilities Reference*.)

3. Using a CMSCALL, CMS now searches for:

   a. A command installed as a nucleus extension

   b. A transient module already loaded with the command name

   c. A nucleus resident command

   d. A MODULE.

   Note: For more information on using CMSCALL, see *z/VM: CMS Application Development Guide for Assembler*.

   For further explanation about the CMS search order, see *z/VM: CMS Commands and Utilities Reference*.

4. Search for a translation or synonym of the specified command name. If found, search for a module with the valid translation or synonym by repeating Step 3.

5. If CMS does not know the command name (that is, all the above fails), it is changed to uppercase and the language processor asks CMS to process the command as a CP command.

To call a CP command explicitly, use the CMS command prefix CP.

To illustrate these last two points, suppose your exec contains the clause:

```
cp spool printer class s
```

You may have a "cover" program, CP EXEC, which is intended to intercept all explicit CP commands. If such a program exists, it is called. If not, the CP command SPOOL is called. You prefix your command with the word cp if you want to avoid invoking SPOOL EXEC or SPOOL MODULE.

Notes:

1. If the command is passed to CP, it is processed as if it had been entered from the CMS command line. (Specifically, if the password suppression facility is in use, a CP command that provides a password is rejected. To enter such a command, use ADDRESS COMMAND CP cp_command.)

2. The searches for execs, translations, synonyms, and CP commands are all affected by the CMS SET command (IMPEX, ABBREV, IMPCP, and TRANSLATE options). The full search order just described assumes these are all ON.

3. Because execs are often used as "covers" or extensions to existing modules, there is one exception to this order. A command issued from within an exec does not implicitly call that same exec and, therefore, cause a possible recursive loop. To make your exec call itself recursively, use the CALL instruction or the EXEC command.

4. When the environment is CMS, the language processor provides both a Tokenized Parameter List and an Extended Parameter List. For example, the sequence:

```
filename="   Jack"; filetype="Assemblersource"
State filename filetype A
Myexec filename filetype A
```

   results in both a Tokenized Parameter List and an Extended Parameter List being built for each command and submitted to CMS. The STATE command uses the Tokenized Parameter List:

```
(STATE   )  (JACK    )  (ASSEMBLE)  (A      )
```

   while MYEXEC (if it is a REXX EXEC) uses the Extended Parameter List:

```
(MYEXEC    Jack Assemblersource A)
```

   For full details of this assembler language interface, see Chapter 8, "System Interfaces," on page 177.

The COMMAND Environment

To enter commands without searching for execs or CP commands and without any translation of the parameter lists (without any uppercasing of the Tokenized Parameter List) you can use the environment called COMMAND. Simply include the instruction ADDRESS COMMAND at the start of your exec (see "ADDRESS" on page 24). Commands are passed to CMS directly, using CMSCALL (see "Calls Originating from a Clause That Is an Expression" on page 178).

The COMMAND environment name is recommended for use in "system" execs that make heavy use of modules and nucleus functions. This makes these execs more predictable (user execs cannot usurp commands, and operations can be independent of the user's setting of IMPCP and IMPEX) and faster (the exec and first abbreviation searches are avoided).

Issuing Subcommands from Your Program

A command CMS is processing may accept **subcommands**. Usually, the command provides its own command line, from which it takes subcommands the user enters. But this can be extended so that the command accepts subcommands from a REXX program.

A typical example is an editor. You can write a REXX program that issues editor subcommands and run your program during an editing session. Your program can inspect the file being edited, issue subcommands to make changes, test return codes to check that the subcommands have been processed as you expected, and display messages to the user when appropriate. The user calls your program by entering its name on the editor's command line.

The editor (or any other program that is designed to accept subcommands from the language processor) must first create a **subcommand entry point**, naming the environment to which subcommands may be addressed, and then call your program. Programs that can issue subcommands are called **macros**. The REXX language processor has the convention that, unless instructed otherwise, it directs commands to a subcommand environment whose name is the file type of the macro. Usually, editors name their subcommand entry point with their own name and claim that name as the file type to be used for their macros.

For example, the XEDIT editor sets up a subcommand environment named XEDIT, and the file type for XEDIT macros is also XEDIT. Macros issue subcommands to the editor (for example, NEXT 4 or EXTRACT /ZONE/). The editor "replies" with a return code (which the language processor assigns to the special variable RC) and sometimes with further information, which may be assigned to other REXX variables. For example, a return code of 1 from NEXT 4 indicates that end-of-file has been reached; EXTRACT /ZONE/ assigns the current limits of the **zone** of XEDIT to the REXX variables ZONE.1 and ZONE.2. By testing RC and the other REXX variables, the macro has the ability to react appropriately, and the full flexibility of a programmable interface is available.

You can alter the default environment (between various subcommand environments or the host environment) by using the ADDRESS instruction.

Note: The CMS SUBCOM function creates, queries, or deletes subcommand entry points.

Only the query form of the SUBCOM function is a subcommand, in the sense that it can be entered from the terminal (or from a REXX program). The form of this subcommand is:

```
SUBCOM   name
```

This yields a return code of 0 if *name* is currently defined as a subcommand environment name, or 1 if it is not.

The create, delete, and query subfunctions of the SUBCOM function are described in the *z/VM: CMS Macros and Functions Reference*. Note that there is also a SUBCOM assembler language macro. The SUBCOM macro is described in the *z/VM: CMS Application Development Guide for Assembler* and *z/VM: CMS Macros and Functions Reference*.

Using the Online HELP Facility

You can receive online information about the commands described in this book using the z/VM HELP Facility. For example, to display a menu of REXX functions and instructions, enter:

```
help rexx menu
```

To display information about a specific REXX function or instruction, (SAY in this example), enter:

```
help rexx say
```

You can also display information about a message by entering one of the following commands:

```
help msgid or help msg msgid
```

For example, to display information about message DMSREX460E, you can enter one of the following commands:

```
help dmsrex460e or help dms460e or help msg dms460e
```

For more information about using the HELP Facility, see *z/VM: CMS User's Guide*. To display the main HELP Task Menu, enter:

```
help
```

For more information about the HELP command, see *z/VM: CMS Commands and Utilities Reference* or enter:

```
help cms help
```

Chapter 2. 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 13, 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 209.

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
```

23

ADDRESS

Note: This HELP file contains both the ADDRESS instruction and the ADDRESS function. Scroll down for the function.
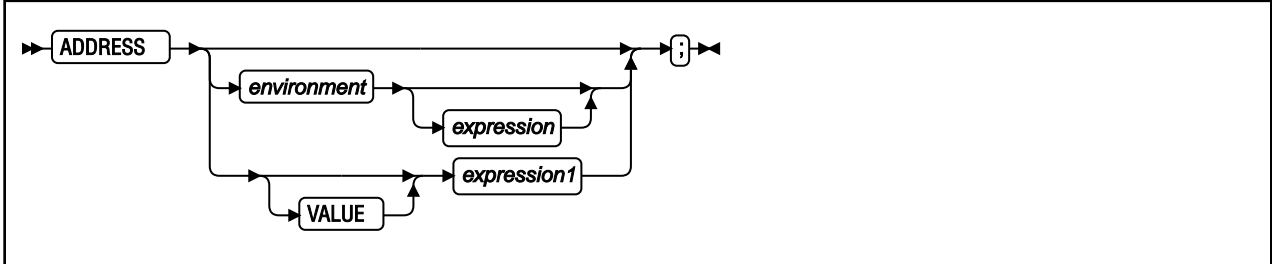
ADDRESS temporarily or permanently changes the destination of commands. Commands are strings sent to an external environment. You can send commands by specifying clauses consisting of only an expression or by using the ADDRESS instruction.

The concept of alternative subcommand environments is described in "The COMMAND Environment" on page 20.

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 environment name is limited to eight characters. 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 17.) Errors and failures in commands processed in this way are trapped or traced as usual.

Example:

```
ADDRESS CMS 'STATE PROFILE EXEC A' /*  VM      */
```

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

Example:

```
address cms
'STATE PROFILE EXEC A'
if rc=0 then 'COPY PROFILE EXEC A TEMP = ='
ADDRESS XEDIT
```

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 29) for more details.

The address setting is the currently selected environment name. You can retrieve the current address setting by using the ADDRESS built-in function (see "ADDRESS" on page 72).

Note: In CMS, there are environment names that have special meaning. Following are three commonly used environment names:

CMS

This environment name, which is the default for execs, implies full command resolution just as is provided in usual interactive command (terminal) mode. (See "The CMS Environment" on page 18 for details.)

CMSMIXED

This environment name implies full command resolution just as is provided in usual interactive command (terminal) mode. (See "The CMS Environment" on page 18 for details). The difference between this and ADDRESS CMS is that the input on ADDRESS CMSMIXED is not changed to uppercase—the case is kept just as you typed it.

Also, there is only a subset of commands that you can use in this new subcommand environment. These are: COPYFILE, DISCARD, ERASE, and RENAME. If you enter anything other than these, you will get a return code of -7.

Here are some examples and their results:

```
address cmsmixed 'SENDFILE my file A'
*-* ADDRESS CMSMIXED 'SENDFILE my file A'
+++ RC(-7) +++
Ready(-0007);
```

This example did not work because SENDFILE is not a command that is allowed in this environment.

```
address cmsmixed 'erase my file A'
*-* ADDRESS CMSMIXED 'erase my fil A'
+++ RC(-7) +++
Ready(-0007);
```

This example did not work because `erase` has no meaning in lowercase.

```
address cmsmixed 'ERASE my file a'
DMSSTT048E Invalid filemode a
Ready;
```

This example did not work because the file mode has to be in uppercase.

```
address cmsmixed 'ERASE my file A'
Ready;
```

COMMAND

This implies basic CMS CMSCALL command resolution. To call an exec, prefix the command with the word EXEC; to send a command to CP, use the prefix CP (see "The COMMAND Environment" on page 20).

"

(Null); same as COMMAND. Note that this is not the same as ADDRESS with no arguments, which switches to the previous environment.

Note: Be aware of CMS storage management techniques if you will be using host commands.

Syntax Notes:

When using CMS or COMMAND, you can put the word in quotation marks, but it must be in uppercase. For example, these will work:

```
address cms 'STATE PROFILE EXEC A'
ADDRESS CMS
ADDRESS 'CMS' your_host_command
```

But this will not work:

```
address 'cms'
```

For information about using ADDRESS to call CPI Communications routines, see Chapter 12, "Invoking Communications Routines," on page 213.

For information about using ADDRESS to call CPI Resource Recovery routines, see Chapter 13, "Invoking Resource Recovery Routines," on page 215.

For information about using ADDRESS to call OPENVM-type CSL routines (such as OpenExtensions™ for z/VM callable services), see Chapter 14, "Invoking OPENVM Routines," on page 217.

For more information about how CMS handles commands and how it manages storage for commands during and after processing, see the *z/VM: CMS Commands and Utilities Reference*.

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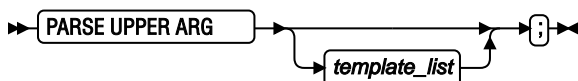
The *template_list* is often a single template but can be several templates separated by commas. If specified, each template is a list of symbols separated by blanks or patterns or both.

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 (Chapter 4, "Parsing," on page 137).

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 72.

2. The source of the data being processed is also made available on entry to the program. See the PARSE instruction (SOURCE option) on "PARSE" on page 48 for details.

3. A string passed from CMS command level is restricted to 255 characters (including the name of the exec being called).

Note for CMS EXEC and EXEC 2 Users: Unlike CMS EXEC and EXEC 2, the arguments passed to REXX programs can only be used after executing either the ARG or PARSE ARG instructions (or retrieving their value with the ARG built-in function). They are not immediately available in predefined variables as in the other languages.

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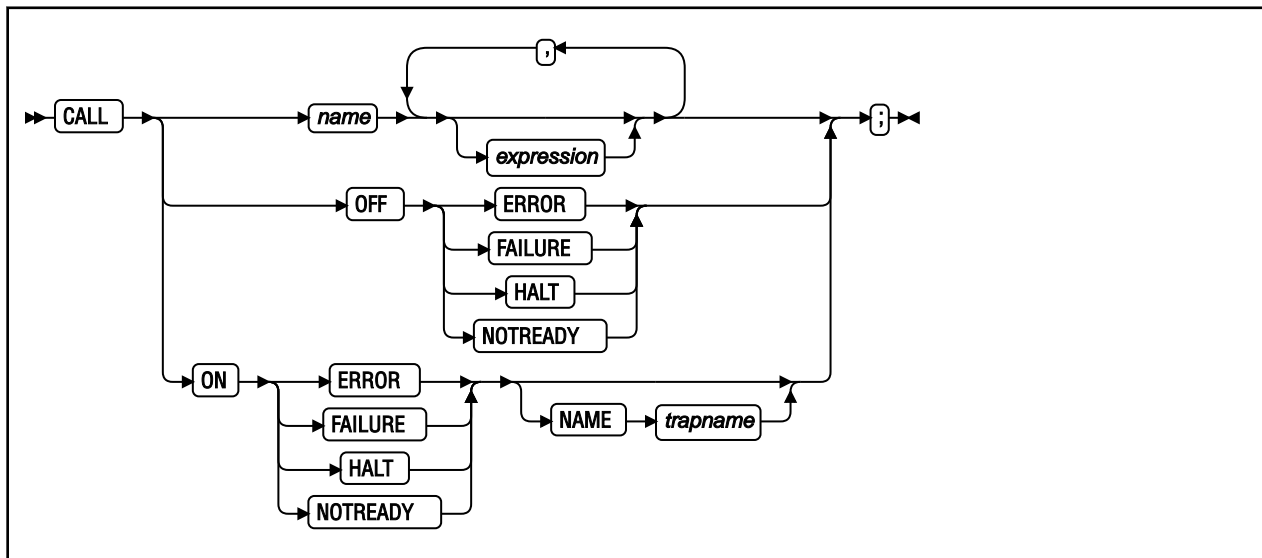
CALL calls a routine (if you specify *name*) or controls the trapping of certain conditions (if you specify ON or OFF).

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 6, "Conditions and Condition Traps," on page 161.

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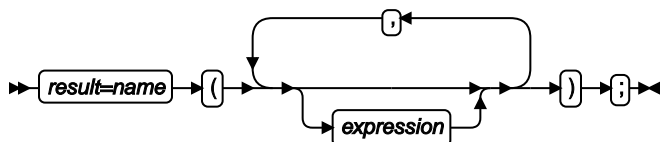
If the called routine does not return a result, then you will get an error if you call it as a function (as previously shown).

VM 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 3, "Functions," on page 67.) The search order is in the section on functions (see "Search Order" on page 68) 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 59). 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 71.)

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. See "Function Packages" on page 189 for details. 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 44) 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 24) 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 78.
- **Elapsed-time clocks**: A subroutine inherits the elapsed-time clock from its caller (see "TIME" on page 108), 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 46.

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

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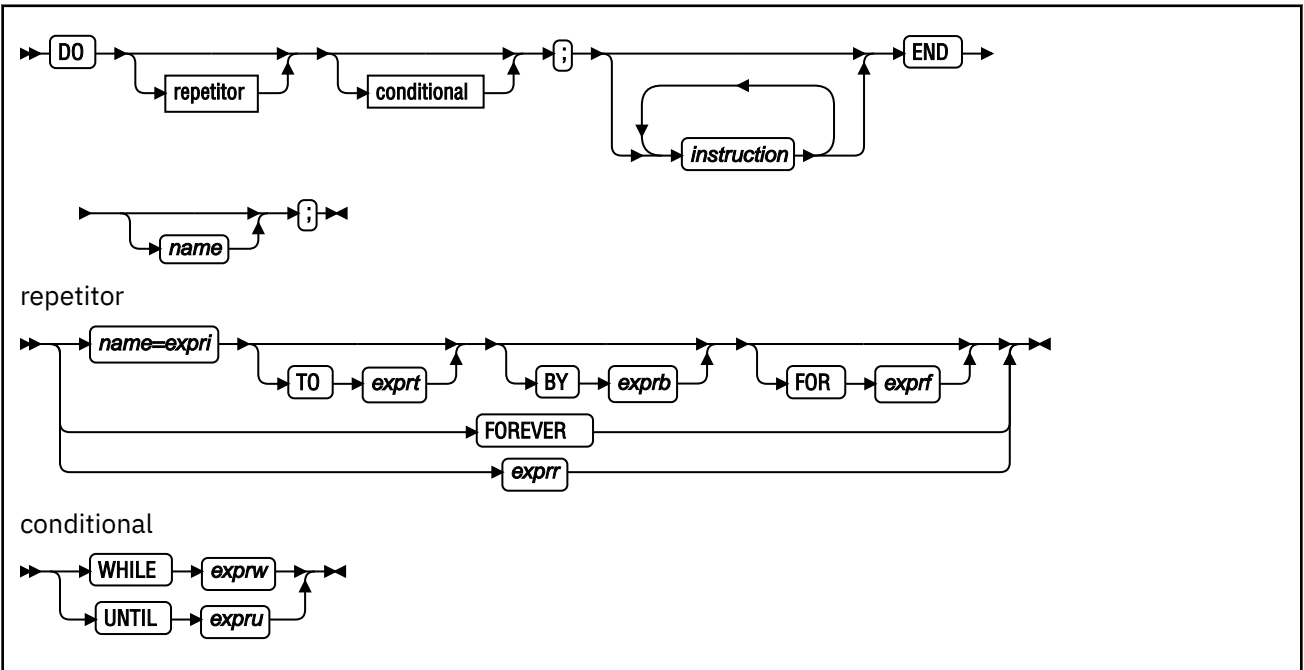
DO groups instructions together and optionally processes them repetitively. During repetitive execution, a control variable (*name*) can be stepped through some range of values.

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 34.

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 34.

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.
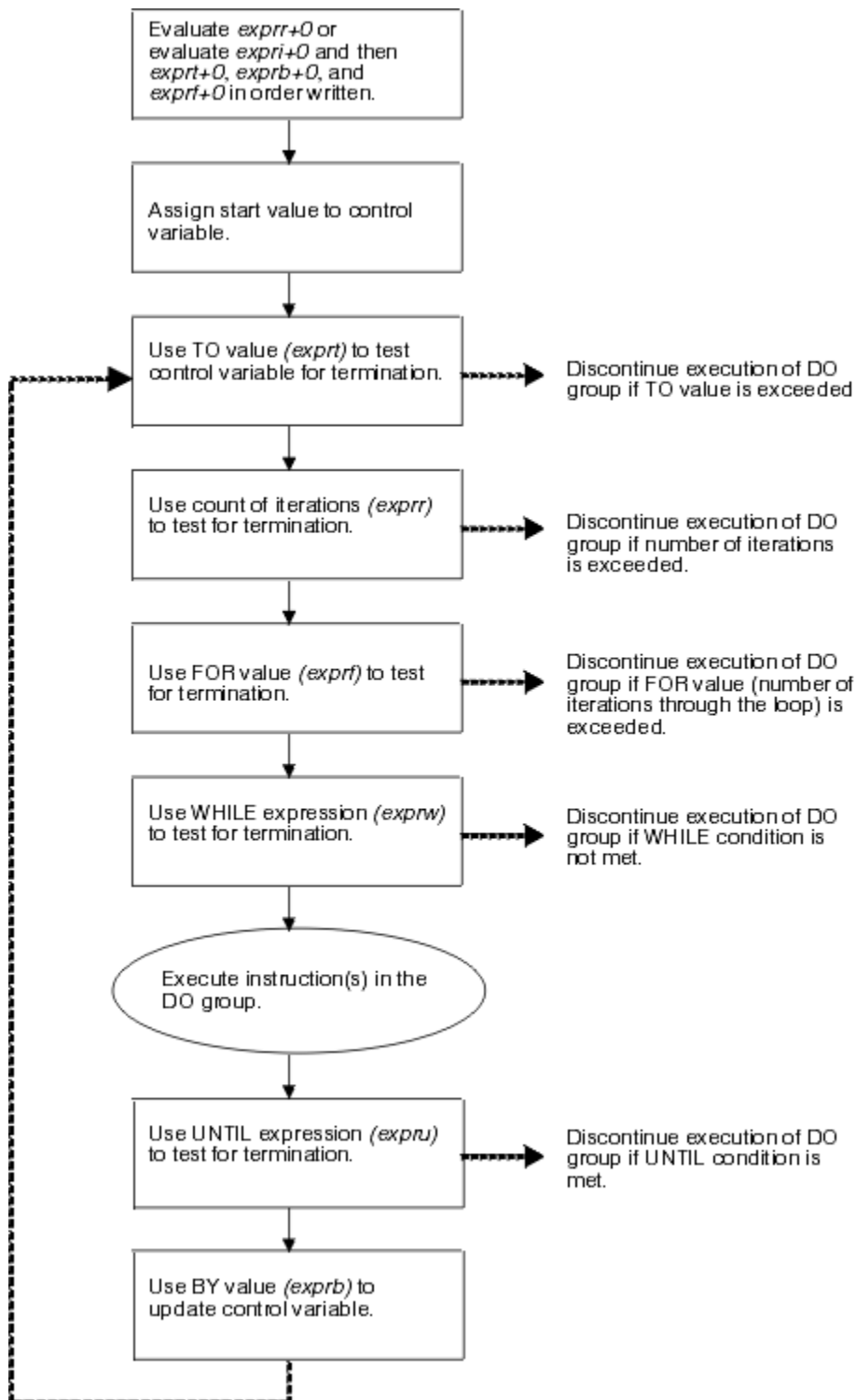
Figure 1: Concept of a DO Loop

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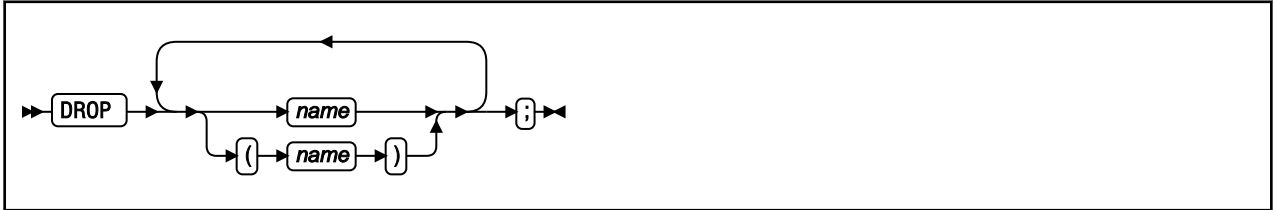
DROP "unassigns" variables, that is, restores them to their original uninitialized state. If *name* is not enclosed in parentheses, it identifies a variable you want to drop and must be a symbol that is a valid variable name, separated from any other *name* by one or more blanks or comments.

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 51), 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
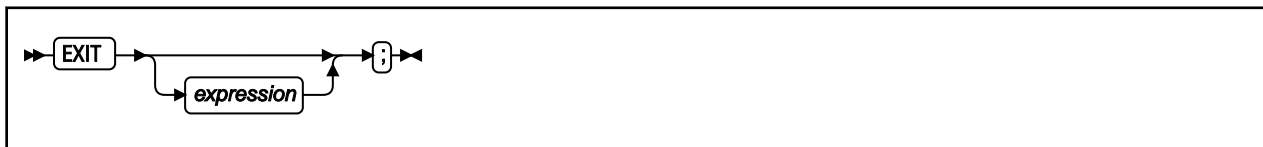
EXIT leaves a program unconditionally. Optionally EXIT returns a character string to the caller. The program is stopped immediately, even if an internal routine is currently being run. If no internal routine is active, RETURN (see "RETURN" on page 56) and EXIT are identical in their effect on the program 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 underlying operating system. If the conversion fails, it is deemed to be unsuccessful due to the underlying operating system 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). Note also that the standard CMS ready message displays only the last five digits of the return code (four digits for a negative return code).

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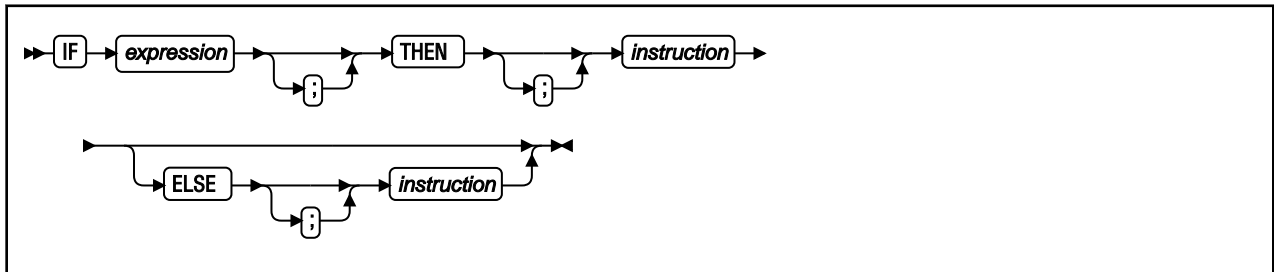
IF conditionally processes an instruction or group of instructions depending on the evaluation of the *expression*. The *expression* is evaluated and must result in 0 or 1.

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

Example:

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

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

Example:

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

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

Example:

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

Note:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, SELECT, or the IF instruction itself. A null clause is not an instruction, so putting an extra semicolon (or label) after the THEN or ELSE is not equivalent to putting a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.

2. The symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the IF clause to be ended by the THEN, without a ; being required. If this were not so, people who are accustomed to other computer languages would experience considerable difficulties.

INTERPRET

```
▶▶──[ INTERPRET ]──▶──( expression )──▶──[ ; ]──▶◀
```
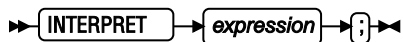
INTERPRET processes instructions that have been built dynamically by evaluating *expression*.

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

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

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

Example:

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

Example:

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

Note:

1. Label clauses are not permitted in an interpreted character string.

2. If you are new to the concept of the INTERPRET instruction and are getting results that you do not understand, you may find that executing it with TRACE R or TRACE I in effect is helpful.

   Example:

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

   When this is run, it gives the trace:

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

```
        >O>    "Hello Kitty!"
Hello Kitty!
```

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

3. For many purposes, you can use the VALUE function (see) 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├─┬──────────┬─►─┤;├─◄
             └─► name ─┘
```

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 just as though the END clause had been encountered. 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 ─┬──────────┬─ ; ─◄◄
           └─ name ─┘
```

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, just as though the END clause had been encountered and the termination condition had been met. However, on exit, 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 ─►─ ; ─►◄
```

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──┬──DIGITS──┬─────────────┬──────────────────┬──;──◀◀
             │          └─expression1─┘                  │
             │   ┌──SCIENTIFIC────────┐                  │
             ├─FORM──┤                ├──────────────────┤
             │       ├──ENGINEERING───┤                  │
             │       │    ┌─expression2─┐                │
             │       └─VALUE──┘        └────────────────┘│
             └─FUZZ──┬──────────────┬────────────────────┘
                     └─expression3──┘
```
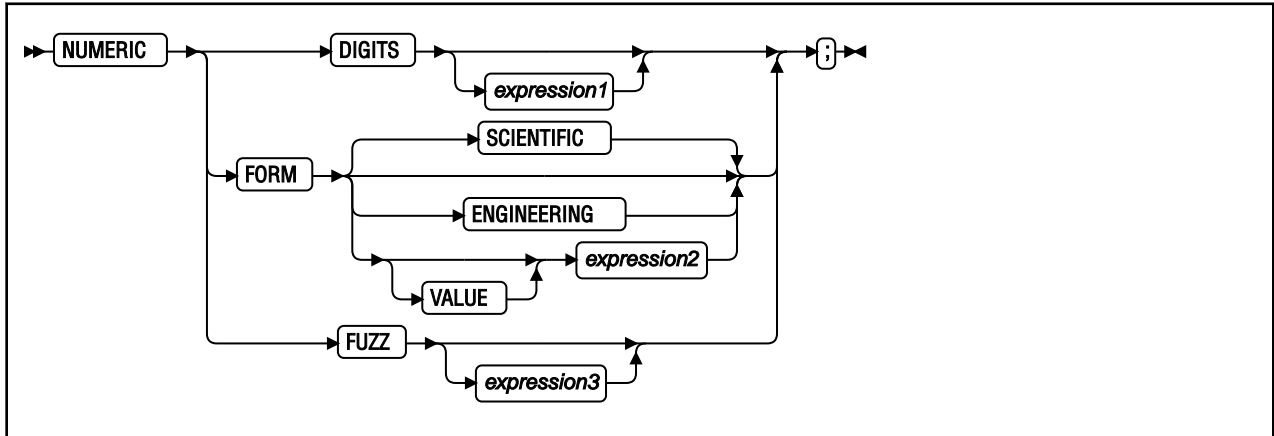
NUMERIC changes the way in which a program carries out arithmetic operations. The options of this instruction are described in detail in Chapter 5, "Numbers and Arithmetic," on page 153-"Errors" on page 160, but in summary:

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—see the note in Chapter 1, "REXX General Concepts," on page 1 for more information) 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 85.

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 86.

NUMERIC FUZZ

controls how many digits, at full precision, are ignored during a numeric comparison operation. (See "Numeric Comparisons" on page 157.) 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 88.

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

OPTIONS

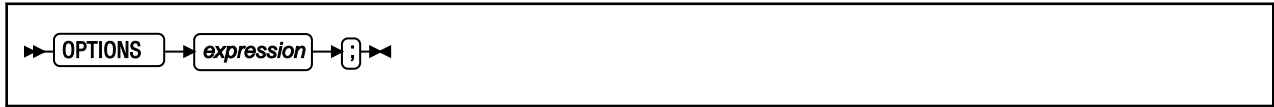OPTIONS ───► expression ───► ; ───►◄

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

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

The language processor recognizes the following words:

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

    If the *expression* is an external function call, for example OPTIONS 'GETETMOD'(), and the program contains DBCS literal strings, enclose the name of the function in quotation marks to ensure that the entire program is not scanned before the option takes effect. It is not 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 messageDMSREX 488E. 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──┬──────────┬──┬────────────────┬──────────────────┬──────────────────┬──▶
           └──UPPER───┘  │      ┌──ARG────┐│                  │  ┌─template_list─┐│
                         │      ├─EXTERNAL┤│                  │  └───────────────┘│
                         │      ├──LINEIN─┤│                  │                   │
                         │      ├─NUMERIC─┤│                  │                   │
                         │      ├──PULL───┤│                  │                   │
                         │      ├─SOURCE──┤│                  │                   │
                         │   ┌─VALUE─┬──────────────┬──WITH──┤                   │
                         │   │       └──expression──┘        │                   │
                         │   ├──VAR───name──────────────────┤                   │
                         │   └─VERSION───────────────────────┘                   │

           ▶──┤ ; ├──◀
```
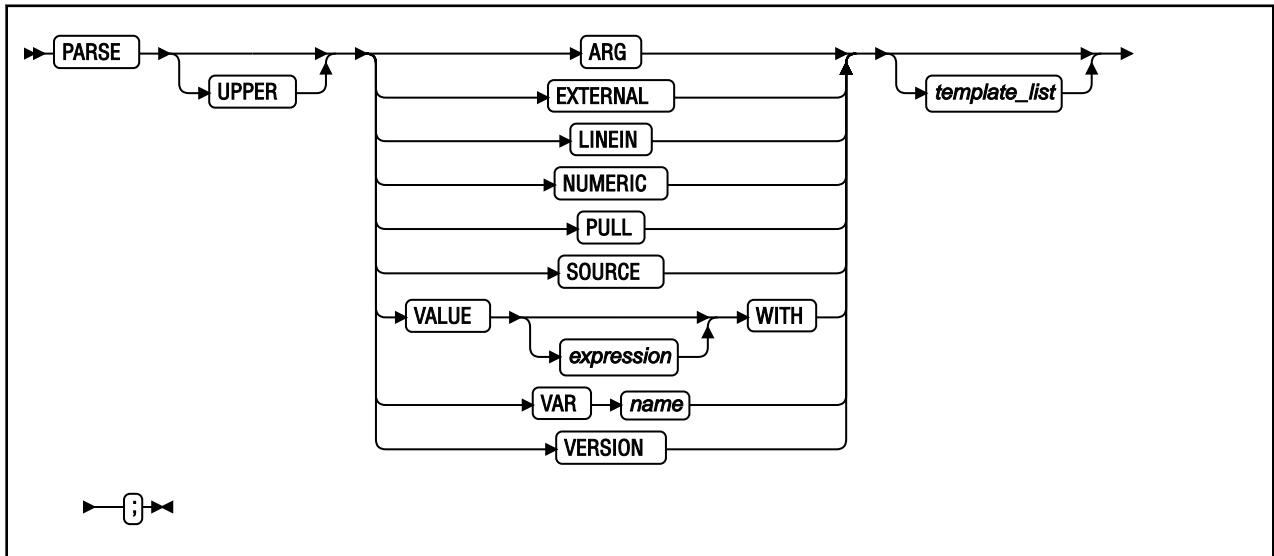
PARSE assigns data (from various sources) to one or more variables according to the rules of parsing. (See Chapter 4, "Parsing," on page 137.)

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

Each template is applied to a single source string. Specifying multiple templates is never a syntax error, but only the PARSE ARG variant can supply more than one non-null source string. See "Parsing Multiple Strings" on page 146 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 EXTERNAL and PARSE PULL, a data string is removed from the queue, for PARSE LINEIN (and PARSE PULL if the queue is empty), a line is taken from the default input stream, 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 in "ARG" on page 27 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 ("ARG (Argument)" on page 72).

PARSE EXTERNAL
This is a subkeyword provided in z/VM. The next string from the terminal input buffer is parsed. This queue may contain data that is the result of external asynchronous events—such as user console input, or messages. If that queue is empty, a console read results. Note that this mechanism should not be used for typical console input, for which PULL is more general, but rather it could be used for special applications (such as debugging) when the program stack cannot be disturbed.

You can find the number of lines currently in the queue with the EXTERNALS built-in function. (See "EXTERNALS" on page 116.)

PARSE LINEIN

parses the next line from the default input stream. (See Chapter 7, "Input and Output Streams," on page 167 for a discussion of REXX input and output.) PARSE LINEIN is a shorter form of the instruction

```
►►──┤PARSE VALUE LINEIN├──┤(├──┤)├──┤WITH├──┬──────────────────┬──┤;├──►◄
                                            └──►┤template_list├──┘
```
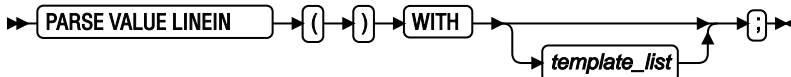
If no line is available, program execution will usually pause until a line is complete. Note that PARSE LINEIN should be used only when direct access to the character input stream is necessary. Usual line-by-line dialogue with the user should be carried out with the PULL or PARSE PULL instructions, to maintain generality.

To check if any lines are available in the default input stream, use the built-in function LINES. (See "LINES (Lines Remaining)" on page 91.) Also see "LINEIN (Line Input)" on page 89 for a description of the LINEIN function.

PARSE NUMERIC

This is a subkeyword provided in VM. The current numeric controls (as set by the NUMERIC instruction, see "NUMERIC" on page 44) 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 44 and the built-in functions "DIGITS" on page 85, "FORM" on page 86, and "FUZZ" on page 88.

PARSE PULL

parses the next string from the external data queue. If the external data queue is empty, PARSE PULL reads a line from the default input stream (the user's terminal), and the program pauses, if necessary, until a line is complete. You can add data to the head or tail of the queue by using the PUSH and QUEUE instructions, respectively. You can find the number of lines currently in the queue with the QUEUED built-in function. (See "QUEUED" on page 93.) 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 in "PULL" on page 53.

Note: PULL and PARSE PULL read from the program stack. If that is empty, they read from the terminal input buffer; and if that too is empty, a console read results. (See the PULL instruction, "PULL" on page 53, for further details.)

PARSE SOURCE

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

The source string contains the characters CMS, followed by either COMMAND, FUNCTION, or SUBROUTINE, depending on whether the program was called as some kind of host command (for example, exec or macro), or from a function call in an expression, or with the CALL instruction. These two tokens are followed by the program file name, file type, and file mode; each separated from the previous token by one or more blanks. (The file type and file mode may be unknown if the program is being run from storage, in which case the SOURCE string has one * for each unknown value.) Following the file mode is the name by which the program was called (because of synonyms, this may not be the same as the file name). It may be in mixed case and is truncated to 8 characters if necessary. (If it cannot be determined, ? is used as a placeholder.) The final word is the initial (default) address for commands.

If the language processor was called from a program that set up a subcommand environment, the file type is usually the name of the default address for commands—see "Issuing Subcommands from Your Program" on page 20 for details. Note that if a PSW is used for the default address, the PARSE SOURCE string uses ? in the initial address for commands position.

The string parsed might, therefore, look like this:

```
CMS COMMAND REXTRY EXEC * rext CMS
```

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
```

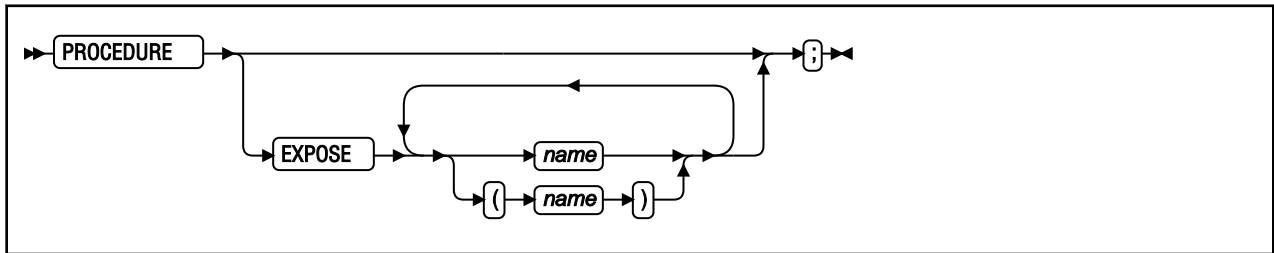in addition translates the data from *string* to uppercase before it is parsed.

PARSE VERSION

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

1. The string REXX370, signifying the 370 implementation
2. The language level description (for example, 4.00),
3. The language processor release date (for example, 31 May 1992).

PROCEDURE

```
►►─┤PROCEDURE├─┬──────────────────────────────────────┬──┤;├─►◄
               │                          ◄──────────┐ │
               └─┤EXPOSE├─┬──────────────┬─┤name├─┬──┴─┘
                         │              ◄─┐        │
                         └─┤(├─┤name├─┤)├─┘────────┘
```

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 15 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 "CALL" on page 29 and Chapter 3, "Functions," on page 67 for details and examples of how routines are called.

PULL

```
►►─ PULL ─┬──────────────────┬─ ; ─►◄
          └─ template_list ─┘
```
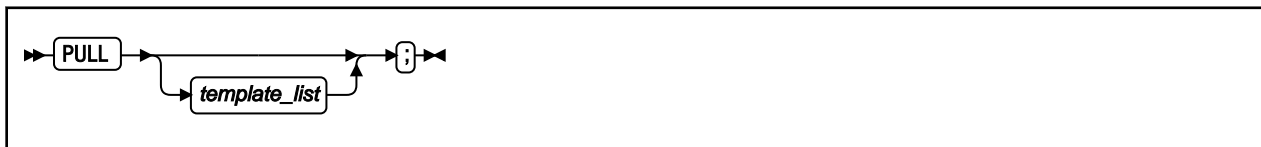
PULL reads a string from the head of the external data queue. (See Chapter 7, "Input and Output Streams," on page 167 for a discussion of REXX input and output.) It is just a short form of the instruction:

```
►►─ PARSE UPPER PULL ─┬──────────────────┬─ ; ─►◄
                      └─ template_list ─┘
```
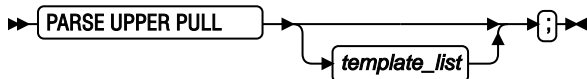
The current head-of-queue is read as one string. Without a *template_list* specified, no further action is taken (and the string is thus effectively discarded). If specified, a *template_list* is usually a single template, which is a list of symbols separated by blanks or patterns or both. (The *template_list* 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 (Chapter 4, "Parsing," on page 137). Use the PARSE PULL instruction if you do not desire uppercase translation.

Note: The VM implementation of the queue is the program stack. The language processor asks CMS to read a line from the most recently created program stack buffer. If this buffer is empty, CMS drops it (except for buffer 0), and reads a line from the next most recently created buffer. If the program stack is empty, the terminal input buffer is used. If that too is empty, a console read occurs. Conversely, if you type before an exec asks for your input, your input data is added to the end of the terminal input buffer and is read at the appropriate time. The length of an entry in the program stack is restricted to 255 characters and the length of data in the terminal input buffer is restricted to 255 characters. For more information on the program stack and the terminal input buffer, see *z/VM: CMS Application Development Guide*.

For the GCS implementation, see note "7" on page 303 in Appendix E, "z/VM REXX/VM Interpreter in the GCS Environment," on page 303.

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.

If the external data queue is empty, a line is read from the default input stream and the program pauses, if necessary, until a line is complete. (This is as though PARSE UPPER LINEIN had been processed. See "PARSE" on page 48.)

The QUEUED built-in function (see "QUEUED" on page 93) returns the number of lines currently in the external data queue.

PUSH

```
►►─┤PUSH├─┬─────────────────┬─►─┤;├─►◄
          └──►┤expression├──►┘
```
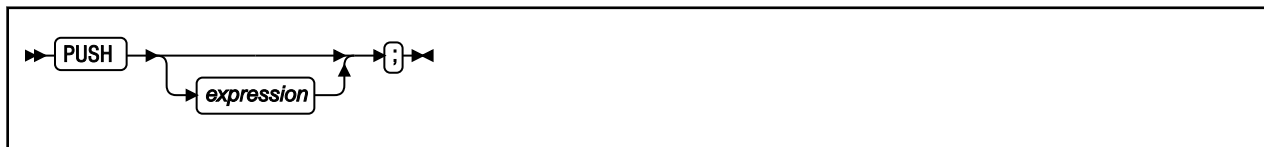
PUSH stacks the string resulting from the evaluation of *expression* LIFO (Last In, First Out) onto the external data queue. (See Chapter 7, "Input and Output Streams," on page 167 for a discussion of REXX input and output.)

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

Note: The VM implementation of the queue is the program stack. The length of an entry in the program stack is restricted to 255 characters. If longer, the data is truncated. The program stack contains one buffer initially, but additional buffers can be created using the CMS command 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 in "QUEUED" on page 93) returns the number of lines currently in the external data queue.

QUEUE

```
▶▶─ QUEUE ─┬──────────────┬─▶ ; ─◀
           └─ expression ─┘
```

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). (See Chapter 7, "Input and Output Streams," on page 167 for a discussion of REXX input and output.)

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

Note: The VM implementation of the queue is the program stack. The length of an element in the program stack is restricted to 255 characters. If longer, the data is truncated. The program stack contains one buffer initially, but additional buffers can be created using the CMS command MAKEBUF.

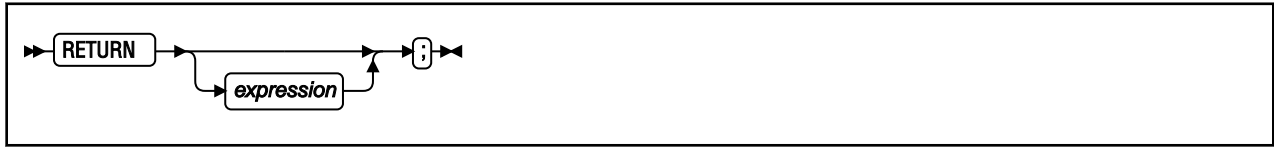**Example:**

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

The QUEUED built-in function (described in "QUEUED" on page 93) returns the number of lines currently in the external data queue.

RETURN

```
►►─ RETURN ─┬──────────────┬─► ─ ; ─►◄
            └─ expression ─┘
```

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 29.)

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 in Chapter 3, "Functions," on page 67 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 ─┬──────────────┬─► ─ ; ─►◄
         └─ expression ──┘
```

SAY writes a line to the default output stream (the terminal) so the user sees it displayed. See Chapter 7, "Input and Output Streams," on page 167 for a discussion of REXX input and output. The result of *expression* may be of any length. If you omit *expression*, the null string is written.

The SAY instruction is a shorter form of the instruction:

```
►►─ CALL LINEOUT ─►─ , ─┬──────────────┬─► ─ ; ─►◄
                        └─ expression ──┘
```
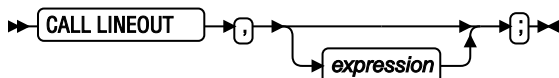
except that:

• SAY does not affect the special variable RESULT
• If you use SAY and omit *expression*, a null string is used
• CALL LINEOUT can raise NOTREADY; SAY cannot.

See "LINEOUT (Line Output)" on page 90 for details of the LINEOUT function.

Note: When in full-screen mode, the result from the SAY instruction is formatted to the width of the virtual screen. However, the window in which you are viewing the result may be smaller than your virtual screen. If so, you may not immediately see the characters in the columns defined by the virtual screen but not defined in the window. To view these characters you can scroll right. You can also reformat the data to fit within the bounds of the window being viewed.

For more information concerning windows and virtual screens, see *z/VM: CMS User's Guide*.

Also, when not in full-screen mode, the data may be reformatted to fit the terminal line size (which you can determine by using the LINESIZE built-in function), if necessary. The line size is restricted to a maximum of 130 characters. The language processor does this reformatting, allowing any length data to be displayed. Lines are typed on a typewriter terminal, or displayed on a display terminal. If you are disconnected (in which case there is no real console, but data can still be written to the console log), or CP TERMINAL LINESIZE OFF has been entered (in which case LINESIZE=0), SAY uses a default line size of 80.

Example:

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

SELECT
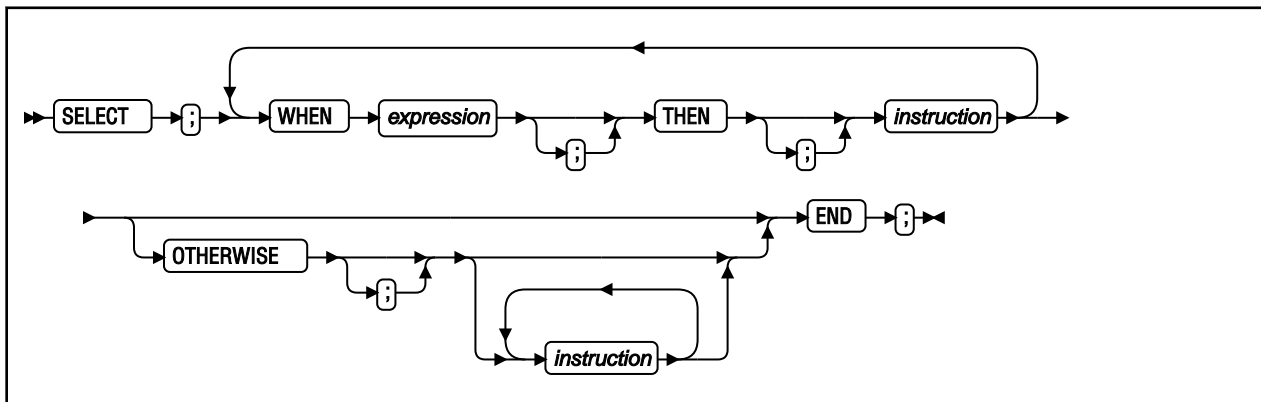
SELECT conditionally calls one of several alternative instructions.

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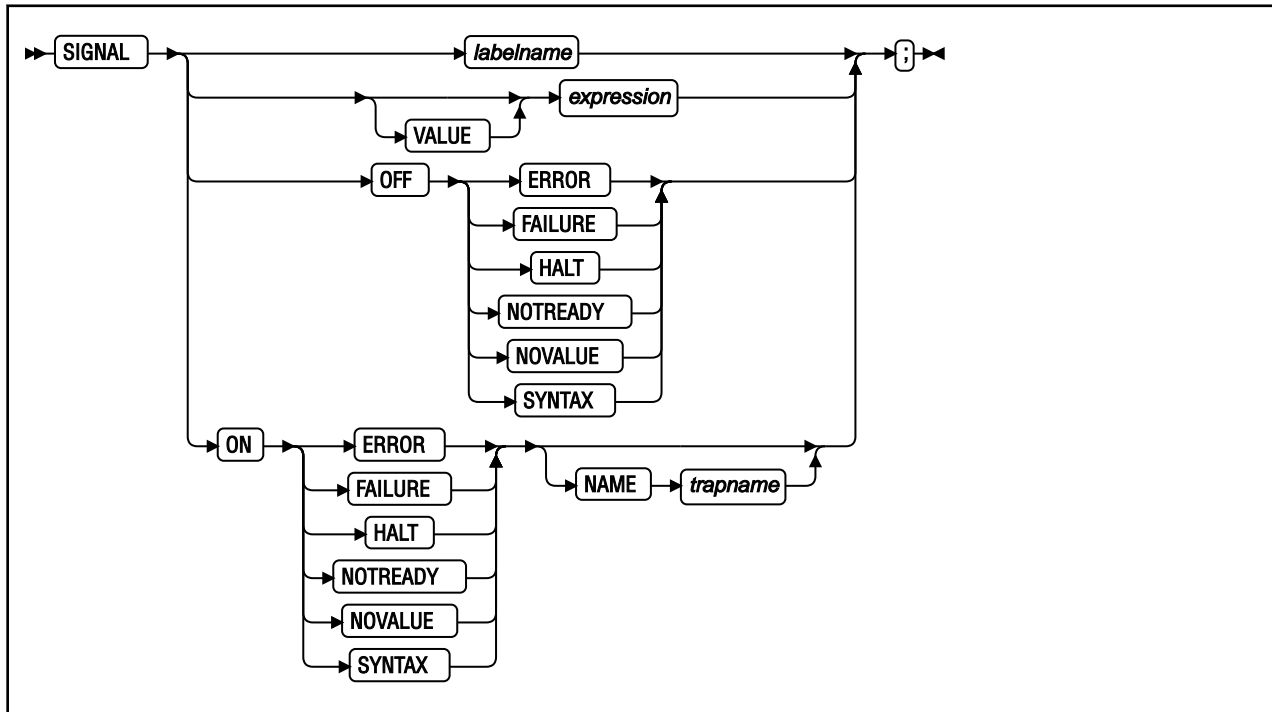
SIGNAL causes an *unusual* change in the flow of control (if you specify *labelname* or VALUE *expression*), or controls the trapping of certain conditions (if you specify ON or OFF).

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 6, "Conditions and Condition Traps," on page 161.

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

Example:

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

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

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

Using SIGNAL VALUE

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

Example:

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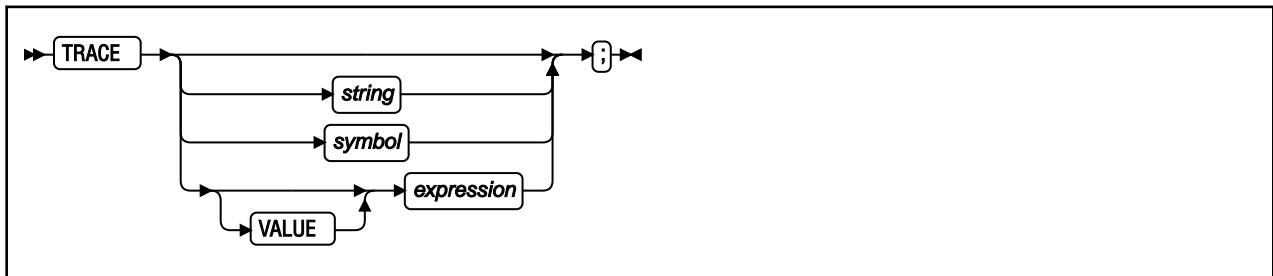
TRACE
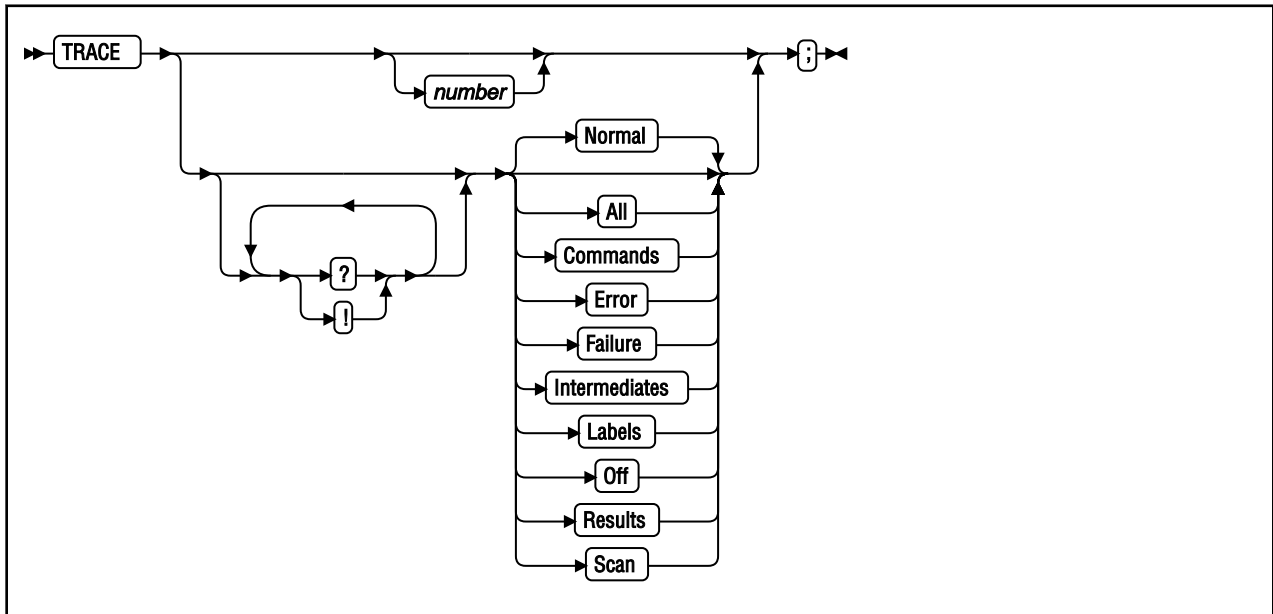
Or, alternatively:

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

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

The *string* or *expression* evaluates to:

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

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

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

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

Alphabetic Character (Word) Options

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

TRACE actions correspond to the alphabetic character options as follows:

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

Commands
Traces all commands before execution. If the command results in an error or failure (see "Commands" on page 17 for definitions of error and failure), then tracing also displays the return code from the command.

Error
Traces any command resulting in an error or failure (see "Commands" on page 17 for definitions of error and failure) after execution, together with the return code from the command.

Failure
Traces any command resulting in a failure (see "Commands" on page 17 for definitions of error and failure) 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:

?
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 205 for full

details of this facility.) While interactive debug is on, interpretation pauses after most clauses that are traced. For example, the instruction TRACE ?E makes the language processor pause for input after executing any command that returns an error (that is, a nonzero return code).

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

You can switch off interactive debug in several ways:

- Entering TRACE 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 also enter interactive debug by entering the CMS immediate command TS from the command line.

!

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 205, 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 the TRACE function).
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 29) 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.      */
```

Note: You can switch tracing on, without modifying a program, by using the CMS command SET EXECTRAC ON.

You can also turn tracing on or off asynchronously (that is, while an exec is running) by using the TS and TE immediate commands. See "Interrupting Execution and Controlling Tracing" on page 206 for the description of these facilities.

Format of TRACE Output

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

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

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

+++
 Identifies a trace message. This may be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error or thread switch 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 138).

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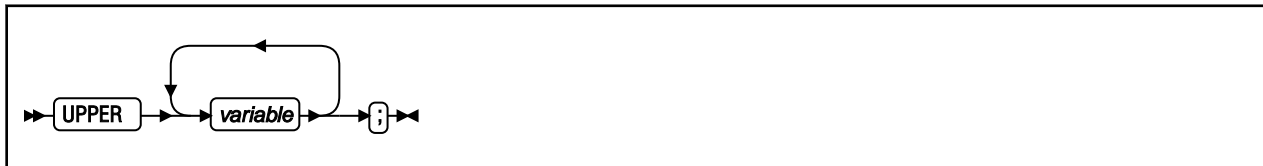
UPPER translates the contents of one or more variables to uppercase. The variables are translated in sequence from left to right.

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 14.)

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

Example:

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

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

Chapter 3. 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:
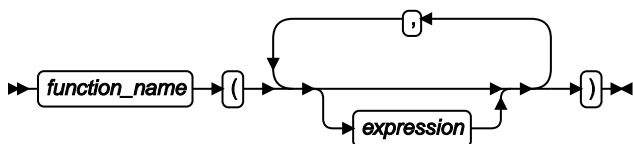
The *function_name* is a literal string or a single symbol, which is taken to be a constant.

There can be up to an implementation-defined maximum number of expressions, separated by commas, between the parentheses. In VM, 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 107) 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 29) 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 59).

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
```

While searching for an internal label, syntax checking is performed and the exec is tokenized. See Appendix C, "Performance Considerations," on page 295 for more details. 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 71.

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.

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.

External functions and **subroutines** have a system-defined search order.

External functions and **subroutines** have a specific search order.

1. The name has a prefix of RX, and the language processor attempts to run the program of that name, using CMSCALL.

2. If the function is not found, the function packages are interrogated and loaded if necessary (they return RC=0 if they contained the requested function, or RC=1 otherwise). The function packages are checked in the order RXUSERFN, RXLOCFN, and RXSYSFN. If the load is successful, step 2 is repeated and will succeed.

3. If still not found, the name is restored to its original form, and all directories and accessed minidisks are first checked for a program with the same file type as the currently executing program (if the file type is not EXEC, as with XEDIT macros for example), and then checked for a file with the file type of EXEC. If either is found, control is passed to it. (The IMPEX setting has no control over this.)

4. Finally the language processor attempts to run the function under its original name, using CMSCALL. (If still not found, an error results.)

The name prefix mechanism, RX, allows new REXX functions to be written with little chance of name conflict with existing MODULES.
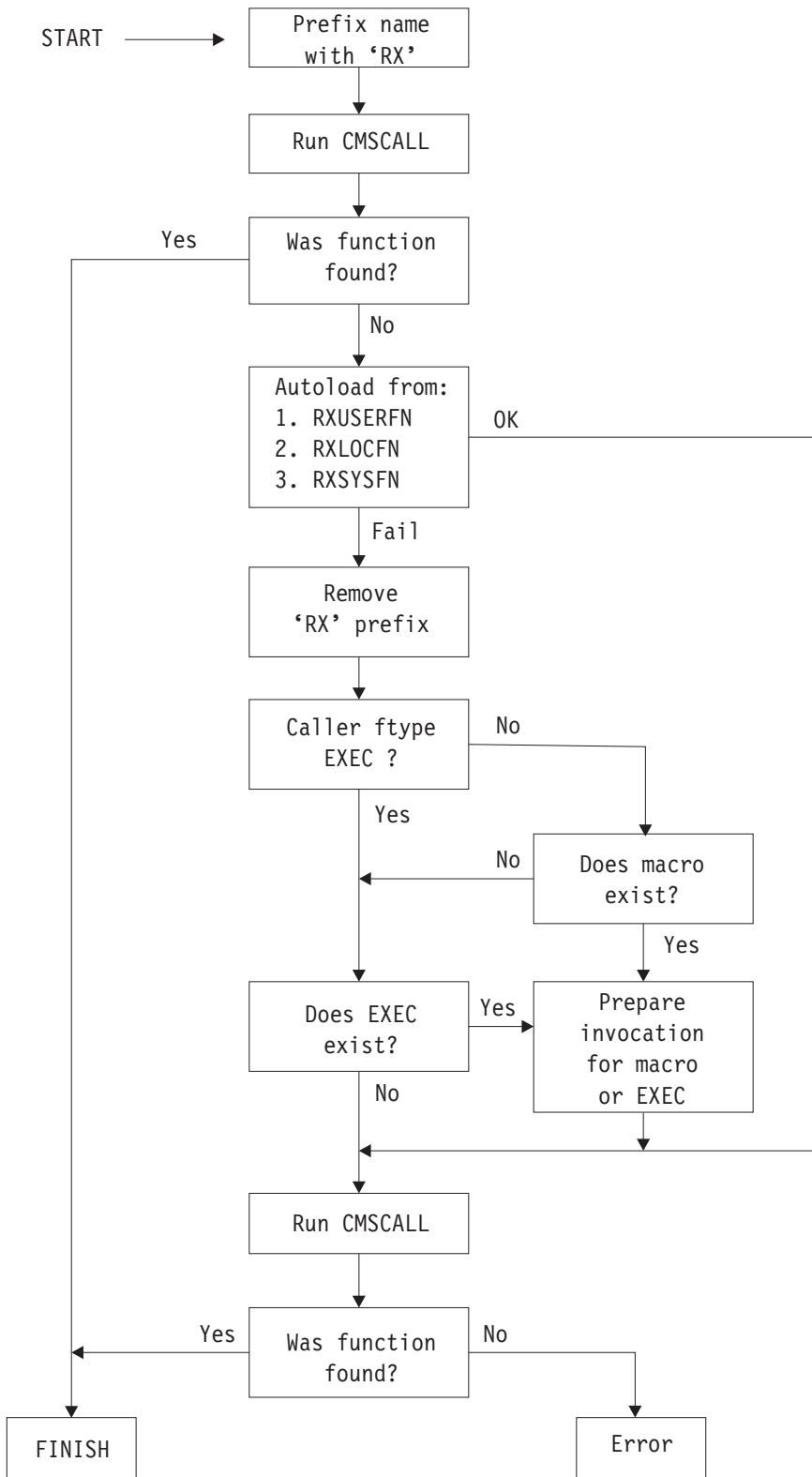
Figure 2: External Routine Resolution and Execution

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.

Other built-in and external functions are generally available—see "Additional Built-in Functions Provided in VM" on page 116, "Function Packages" on page 116, and "External Functions and Routines Provided in VM" on page 118.

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 *number* is rounded, if necessary, according to the current setting of NUMERIC DIGITS (just as though the number had been added to 0) and checked for validity before use. This occurs in the following functions: ABS, FORMAT, MAX, MIN, SIGN, and TRUNC, and for certain options of DATATYPE. This is not true for RANDOM.
- Any argument named as a *string* may be a null string.
- If an argument specifies a *length*, it must be a positive whole number or zero. If it specifies a start character or word in a string, it must be a positive whole number, unless otherwise stated.
- Where the last argument is optional, you can always include a comma to indicate you have omitted it; for example, DATATYPE(1,), like DATATYPE(1), would return NUM. You can include any number of trailing commas; they are ignored. (Where there are actual parameters, the default values apply.)
- 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 in "LEFT" on page 89.
- If a function has an *option* you can select by specifying the first character of a string, that character can be in upper- or lowercase.
- A number of the functions described in this chapter support DBCS. A complete list and descriptions of these functions are in Appendix B, "Double-Byte Character Set (DBCS) Support," on page 279.

ABBREV (Abbreviation)
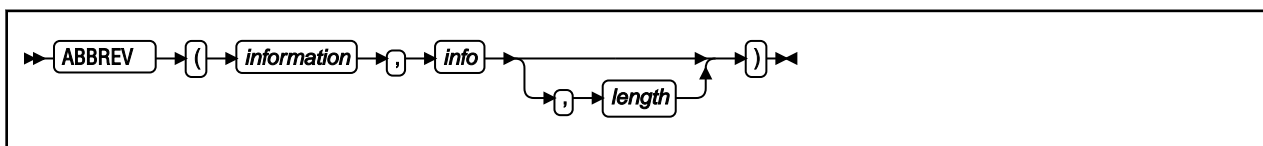
returns 1 if *info* is equal to the leading characters of *information* **and** the length of *info* is not less than *length*. Returns 0 if either of these conditions is not met.

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;
```
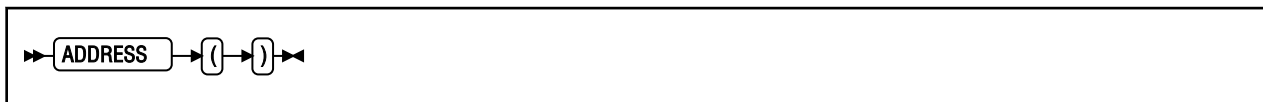
ABS (Absolute Value)

```
►►─ ABS ─►─ ( ─►─ number ─►─ ) ─►◄
```
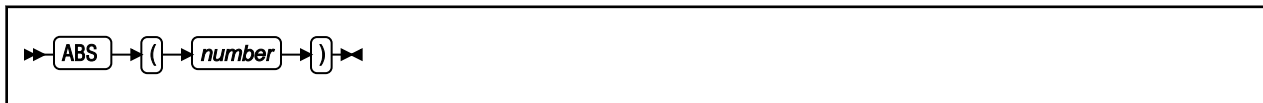
returns the absolute value of *number*. The result has no sign and is formatted according to the current NUMERIC settings.

Here are some examples:

```
ABS('12.3')      ->    12.3
ABS(' -0.307')   ->    0.307
```

ADDRESS

```
►►─ ADDRESS ─►─ ( ─►─ ) ─►◄
```

returns the name of the environment to which commands are currently being submitted. The environment may be a name of a subcommand environment or a PSW. See the ADDRESS instruction (“ADDRESS” on page 24) for more information. Trailing blanks are removed from the result.
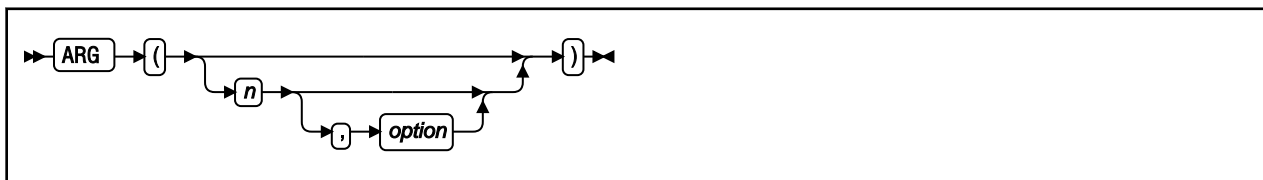
Here are some examples:

```
ADDRESS()   ->    'CMS'     /* default under VM    */
ADDRESS()   ->    'XEDIT'   /* default under XEDIT */
```

APILOAD

This is a CMS external function. See “APILOAD” on page 118.

ARG (Argument)

```
►►─ ARG ─►─ ( ─┬─────────────────────┬─►─ ) ─►◄
               ├─ n ─┤                │
               └─┬───►─ option ─┘     
                 , 
```

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 27, "PARSE" on page 48, and Chapter 4, "Parsing," on page 137.)

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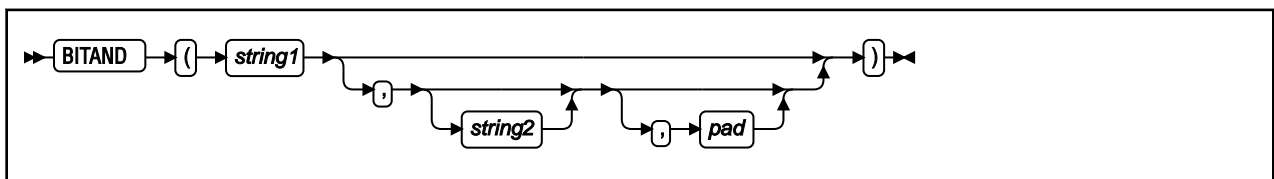
returns a string composed of the two input strings logically ANDed together, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the AND operation stops when the shorter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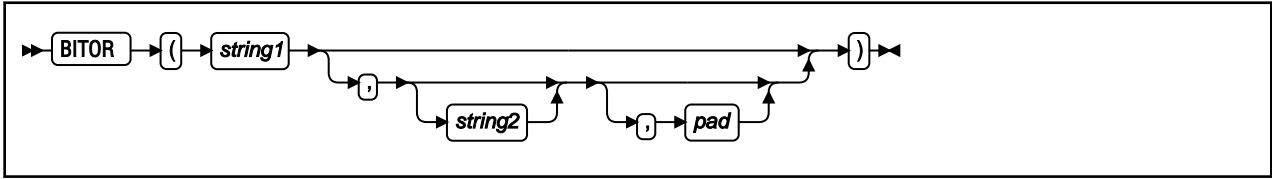
Here are some examples:

```
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)

```
►►─ BITOR ─(─ string1 ──────────────────────────────────────── ) ─►◄
                  ├─ , ─┬─ string2 ─┬──┬─ , ─ pad ─┬──┤
```
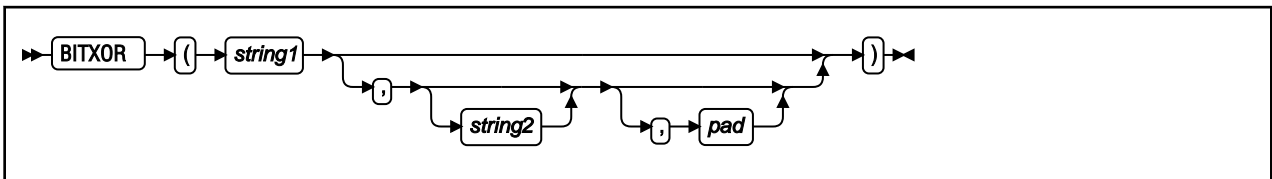
returns a string composed of the two input strings logically inclusive-ORed together, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the OR operation stops when the 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('pQrS',,'40'x)            ->      'PQRS' /* EBCDIC  */
```

BITXOR (Bit by Bit Exclusive OR)

```
►►─ BITXOR ─(─ string1 ──────────────────────────────────────── ) ─►◄
                   ├─ , ─┬─ string2 ─┬──┬─ , ─ pad ─┬──┤
```

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)

```
►►─ B2X ─(─ binary_string ─)─►◄
```
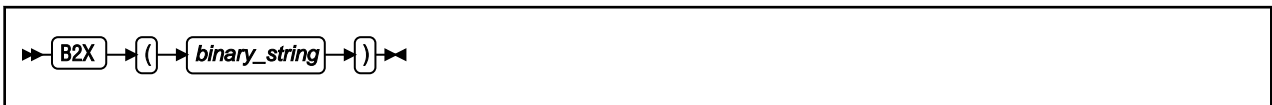
returns a string, in character format, that represents *binary_string* converted to hexadecimal.

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.
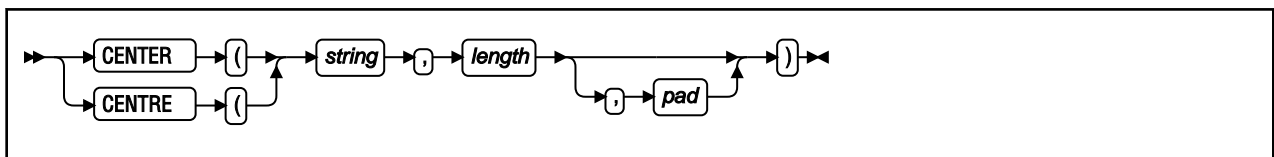
Here are some examples:

```
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.

CHARIN (Character Input)
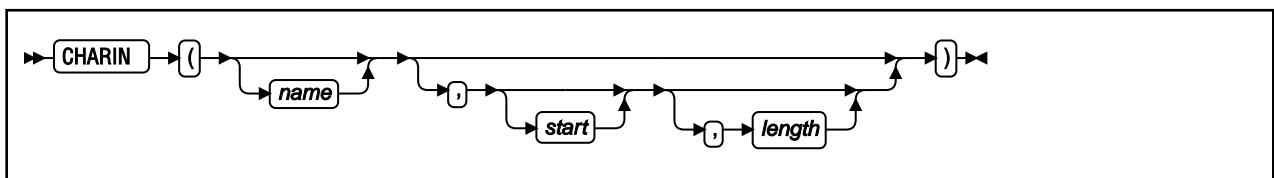
returns a string of up to *length* single-byte characters read from the character input stream *name*. (To understand the input and output functions, see Chapter 7, "Input and Output Streams," on page 167.) The form of *name* is also described in Chapter 7, "Input and Output Streams," on page 167. If you omit *name*, characters are read from default input stream. The default *length* is 1.

For persistent streams, a read position is maintained for each stream. Any read from the stream starts at the current read position by default. When the language processor completes reading, the read position is increased by the number of characters read. A *start* value of 1 (the only valid value) may be given to specify the start of the stream.

If you specify a *length* of 0, then the read position is set to the value of *start* but no characters are read and the null string is returned.

If the number of characters being returned causes a read of an entire line or multiple lines in the stream, a LINEEND character is appended at the end of each line. (See "STREAM" on page 96 for more information on the LINEEND character.) This is only for files that were opened with the TEXT option on the STREAM function. TEXT is the default if not specified on the STREAM function or if the file is implicitly opened by the first I/O call. This LINEEND character is counted in the number of characters returned.

In a transient stream, if there are fewer than *length* characters available, then execution of the program generally stops until sufficient characters do become available. If, however, it is impossible for those characters to become available because of an error or other problem, the NOTREADY condition is raised (see "Errors During Input and Output" on page 173) and CHARIN returns with fewer than the requested number of characters.

Here are some examples:

```
CHARIN(myfile,1,3)   ->    'MFC'     /* the first 3      */
                                     /* characters       */
CHARIN(myfile,1,0)   ->    ''        /* now at start     */
CHARIN(myfile)       ->    'M'       /* after last call  */
CHARIN(myfile,,2)    ->    'FC'      /* after last call  */

/* Reading from the default input (here, the keyboard) */
/* User types 'abcd efg'                               */
CHARIN()             ->    'a'       /* default is       */
                                     /* 1 character      */
CHARIN(,,5)          ->    'bcd e'
```

```
/* assume TEXT and LINEEND='15'x, only 1 character, A, left on */
/* the line and the next line starts with B                   */
CHARIN(myfile,,3)  -> 'A B'   /* in hex: 'C115C2'x            */
```

CHAROUT (Character Output)
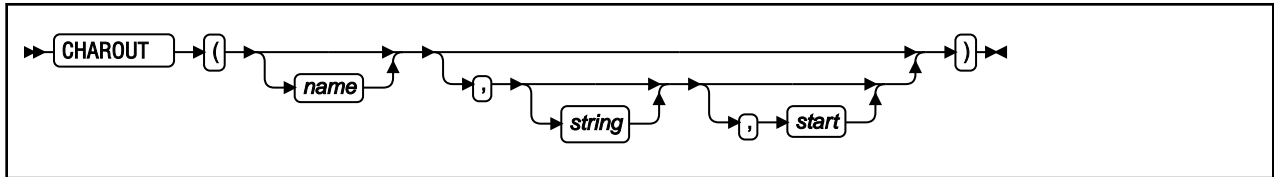
returns the count of single-byte characters remaining after attempting to write *string* to the character output stream *name*. (To understand the input and output functions, see Chapter 7, "Input and Output Streams," on page 167.) The form of *name* is also described in Chapter 7, "Input and Output Streams," on page 167. If you omit *name*, characters in *string* are written to the default output stream. The *string* can be the null string, in which case no characters are written to the stream, and 0 is always returned.

For variable-format streams with the TEXT option, the LINEEND character must be supplied to indicate the end of the record. For fixed-format streams with the TEXT option, the LINEEND does not have to be given, as the data will be split at the appropriate record length. If a LINEEND character is given causing a record shorter than the logical record length, the data will be padded with blanks before being written. The LINEEND character is never written to the stream in TEXT mode; it only serves as an indicator of the end of a line. For fixed- or variable-format streams with the BINARY option, a full buffer indicates the end of a record. (The size of the full buffer can be defined by the LRECL parameter on the STREAM function, or the default will be used.)

For persistent streams, a write position is maintained for each stream. Any write to the stream starts at the current write position by default. When the language processor completes writing, the write position is increased by the number of characters written. When the stream is first opened, the write position is at the end of the stream so that calls to CHAROUT will append characters to the end of the stream.

A *start* value of 1 (the only valid value) may be given to specify the start of the stream.

Note: You will get an error if you try to overwrite a record with another record that has a different length.

You can omit the *string* for persistent streams. In this case, the write position is set to the value of *start* that was given, no characters are written to the stream, and 0 is returned. If you do not specify *start* or *string*, nothing is written to the stream, and it is closed.

Execution of the program usually stops until the output operation is effectively complete.

For example, when data is sent to a printer, the system accepts the data and returns control to REXX, even though the output data may not have been printed on the printer. REXX considers this to be complete, even though the data has not been printed. If, however, it is impossible for all the characters to be written, the NOTREADY condition is raised (see "Errors During Input and Output" on page 173) and CHAROUT returns with the number of characters that could not be written (the residual count).

Here are some examples:

```
CHAROUT(myfile,'Hi')    ->   0   /* typically        */
CHAROUT(myfile)         ->   0   /* at end of stream */
CHAROUT(,'Hi')          ->   0   /* typically        */
CHAROUT(,'Hello')       ->   2   /* maybe            */
```

Note: This routine is often best called as a subroutine. The residual count is then available in the variable RESULT.

For example:

```
Call CHAROUT myfile,'Hello'
Call CHAROUT myfile
```

CHARS (Characters Remaining)
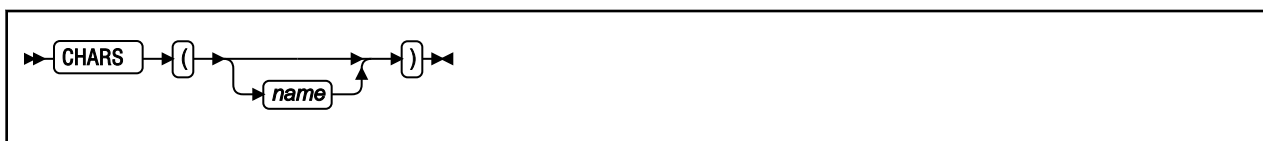
returns either 0 or 1 depending on whether there are characters available in the input stream. CHARS will return 1 if there is at least one character available in the stream and 0 otherwise. (To understand the input and output functions, see Chapter 7, "Input and Output Streams," on page 167.)

The form of *name* is described in Chapter 7, "Input and Output Streams," on page 167. If you omit *name* or it is null (both signifying the default input stream), 1 is returned.

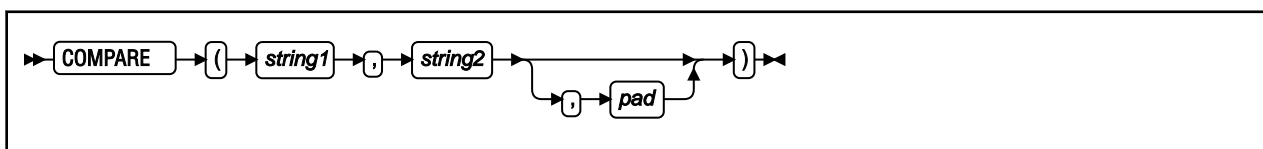Here are some examples:

```
CHARS(myfile)    ->   1   /* perhaps */
CHARS(nonfile)   ->   0   /* perhaps */
CHARS()          ->   1   /* perhaps */
```

Note: The LINES function may be used to return the number of partial and complete lines (rather than individual characters) remaining in the stream.

CMSFLAG

This is a CMS external function. See "CMSFLAG" on page 119.

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 6, "Conditions and Condition Traps," on page 161 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 Chapter 6, "Conditions and Condition Traps," on page 161 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

```
►►─ COPIES ─►─ ( ─►─ string ─►─ , ─►─ n ─►─ ) ─►◄
```

returns *n* concatenated copies of *string*. The *n* must be a positive whole number or zero.

Here are some examples:

```
COPIES('abc',3)    ->    'abcabcabc'
COPIES('abc',0)    ->    ''
```

CSL

This is a CMS external function. See .

C2D (Character to Decimal)

```
►►─ C2D ─►─ ( ─►─ string ─┬──────────────┬─►─ ) ─►◄
                          └─ , ─►─ n ─────┘
```

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)

```
►►─ C2X ─►─( ─►─ string ─►─) ─►◄
```

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

```
►►─ DATATYPE ─►─( ─►─ string ─┬──────────────┬─►─) ─►◄
                              └─, ─►─ type ─┘
```
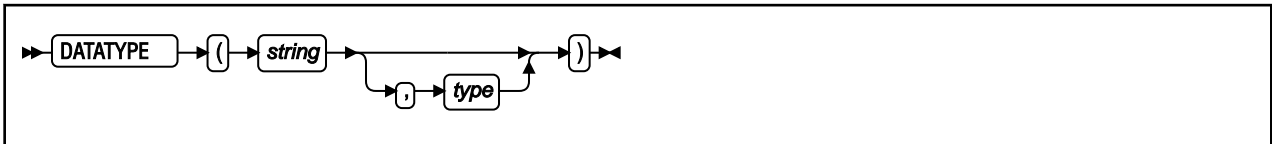
returns NUM if you specify only *string* and if *string* is a valid REXX number that can be added to 0 without error; returns CHAR if *string* is not a valid number.

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 3.) 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.
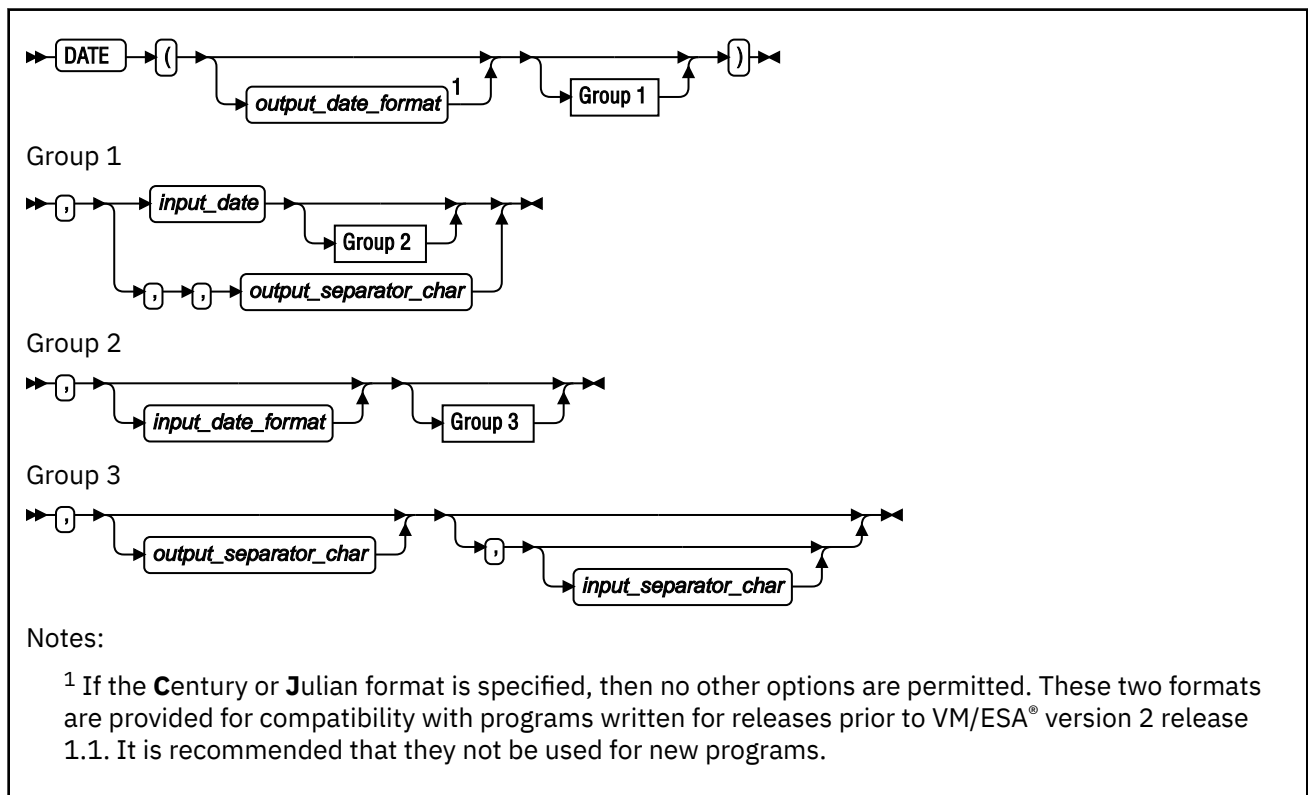
Here are some examples:

```
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 **C**entury or **J**ulian format is specified, then no other options are permitted. These two formats are provided for compatibility with programs written for releases prior to VM/ESA® version 2 release 1.1. 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 1996), 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 date and output date, respectively. Any single nonalphanumeric character is valid. See note for more information.

You can use the following options to obtain specific date formats. (Only the capitalized and highlighted letter 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 independent of the national language in which you are working.

Note: The base date of 1 January 0001 is determined by extending the current Gregorian calendar backward (365 days each year, with an extra day every year that is divisible by 4 except century years that are not divisible by 400). It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

Century
the number of days, including the current day, since and including January 1 of the last year that is a multiple of 100 in the form: *ddddd* (no leading zeros). Example: A call to DATE('C') on March 13 1992 returns 33675, the number of days from 1 January 1900 to 13 March 1992. Similarly, a call to DATE('C') on 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 the current year in the format: *ddd* (no leading zeros or blanks).

Julian
date in the format: *yyddd* (*yy* and *ddd* must have leading zeros).

European
date in the format: *dd/mm/yy* (*dd*, *mm*, and *yy* must have leading zeros).

Month
full name of the current month, in the active language. For example, August, if the active language is American English. 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). If the active language has an abbreviated form of the month name, then it is used (for example, Jan, Feb, and so on). If **N**ormal is specified for *input_date_format*, the *input_date* must have the month (*mon*) specified in American English (for example, Jan, Feb, Mar, and so on).

Ordered
date in the format: *yy/mm/dd* (suitable for sorting, and so forth; *yy*, *mm*, and *dd* must have leading zeros).

Standard
date in the format: *yyyymmdd* (suitable for sorting, and so forth; *yyyy*, *mm*, and *dd* must have leading zeros).

Usa
date in the format: *mm/dd/yy* (*mm*, *dd*, and *yy* must have leading zeros).

Weekday
the name for the day of the week, in the active language. For example, Tuesday, if the active language is American English. 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. For other related routines that perform conversion operations, see the DateTimeSubtract CSL routine in *z/VM: CMS Application Multitasking*, or see the CMS Pipelines DATECONVERT stage in *z/VM: CMS Pipelines User's Guide and Reference*.

4. The "active language" referred to in the Month, Normal, and Weekday options is the national language set by CMS during initial start up or by an explicit SET LANGUAGE command, for example, SET LANGUAGE UCENG.

```
DATE('M')                  ->      'MARCH'
DATE('N')                  ->      '13 MAR 1992'
DATE('N','35488','C')      ->      '28 FEB 1997'
DATE('W')                  ->      'FRIDAY'
```

5. *input_separator_char* and *output_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: Null is a valid value for *input_separator_char* and *output_separator_char*.

6. For consistency with CMS date formats ISODATE and FULLDATE, a REXX program can convert dates to these formats by using a combination of the TRANSLATE and DATE functions:

   • This example returns the current date in ISODATE format:

   ```
   TRANSLATE('year-mn-dt',DATE('S'),'yearmndt')
   ```

- This example returns the current date in FULLDATE format:

```
TRANSLATE('mn/dt/year',DATE('S'),'yearmndt')
```

DBCS (Double-Byte Character Set Functions)

The following are all part of DBCS processing functions. See Appendix B, "Double-Byte Character Set (DBCS) Support," on page 279.

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

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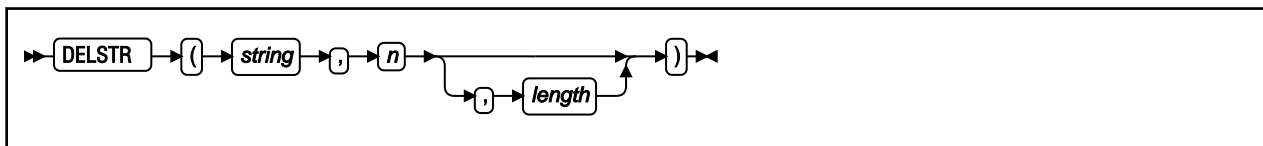
returns *string* after deleting the substring that begins at the *n*th character and is of *length* characters. If you omit *length*, or if *length* is greater than the number of characters from *n* to the end of *string*, the function deletes the rest of *string* (including the *n*th character). The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

Here are some examples:

```
DELSTR('abcd',3)        ->      'ab'
DELSTR('abcde',3,2)     ->      'abe'
DELSTR('abcde',6)       ->      'abcde'
```

DELWORD (Delete Word)
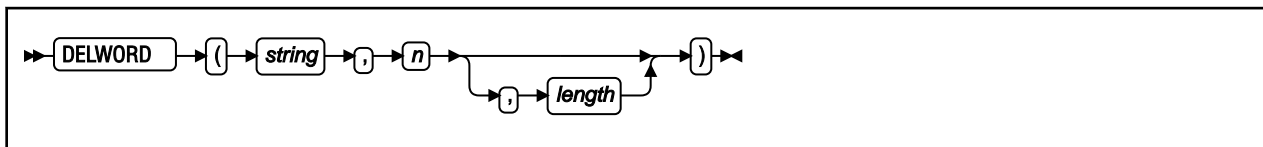
returns *string* after deleting the substring that starts at the *n*th word and is of *length* blank-delimited words. If you omit *length*, or if *length* is greater than the number of words from *n* to the end of *string*, the function deletes the remaining words in *string* (including the *n*th word). The *length* must be a positive whole number or zero. The *n* must be a positive whole number. If *n* is greater than the number of words in *string*, the function returns *string* unchanged. The string deleted includes any blanks following the final word involved but none of the blanks preceding the first word involved.

Here are some examples:

```
DELWORD('Now is the  time',2,2)  ->  'Now time'
DELWORD('Now is the time ',3)    ->  'Now is '
DELWORD('Now is the  time',5)    ->  'Now is the  time'
DELWORD('Now is   the time',3,1) ->  'Now is   time'
```

DIAG/DIAGRC

These are CMS external functions. See "DIAG" on page 124.

DIGITS

```
▶▶─ DIGITS ─( )─ ◀◀
```

returns the current setting of NUMERIC DIGITS. See the NUMERIC instruction in "NUMERIC" on page 44 for more information.

Here is an example:

```
DIGITS()   ->   9      /* by default */
```

D2C (Decimal to Character)
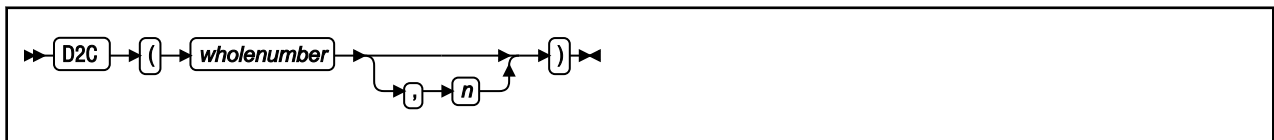
```
▶▶─ D2C ─(─ wholenumber ─┬──────────┬─)─ ◀◀
                         └─ , ─ n ─┘
```

returns a string, in character format, that represents *wholenumber*, a decimal number, converted to binary. If you specify *n*, it is the length of the final result in characters; after conversion, the input string is sign-extended to the required length. If the number is too big to fit into *n* characters, then the result is truncated on the left. The *n* must be a positive whole number or zero.

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

Here are some examples:

```
D2C(9)        ->   ' '    /* '09'x is unprintable in EBCDIC      */
D2C(129)      ->   'a'    /* '81'x is an EBCDIC 'a'              */
D2C(129,1)    ->   'a'    /* '81'x is an EBCDIC 'a'              */
D2C(129,2)    ->   ' a'   /* '0081'x is EBCDIC ' a'             */
D2C(257,1)    ->   ' '    /* '01'x is unprintable in EBCDIC      */
D2C(-127,1)   ->   'a'    /* '81'x is EBCDIC 'a'                */
D2C(-127,2)   ->   ' a'   /* 'FF'x is unprintable EBCDIC;        */
                          /* '81'x is EBCDIC 'a'                */
D2C(-1,4)     ->   '    ' /* 'FFFFFFFF'x is unprintable in EBCDIC */
D2C(12,0)     ->   ''     /* '' is a null string                */
```

Implementation maximum: The output string may not have more than 250 significant characters, though a longer result is possible if it has additional leading sign characters ('00'x and 'FF'x).

D2X (Decimal to Hexadecimal)
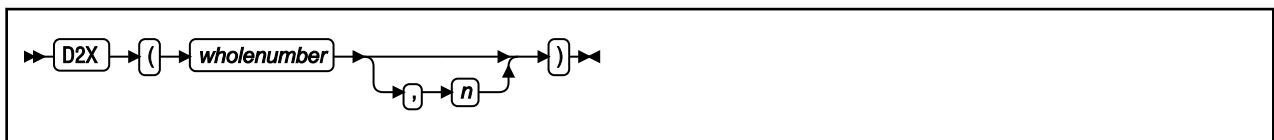
```
▶▶─ D2X ─(─ wholenumber ─┬──────────┬─)─ ◀◀
                         └─ , ─ n ─┘
```

returns a string, in character format, that represents *wholenumber*, a decimal number, converted to hexadecimal. The returned string uses uppercase alphabetics for the values A–F and does not include blanks.

If you specify *n,* it is the length of the final result in characters; after conversion the input string is sign-extended to the required length. If the number is too big to fit into *n* characters, it is truncated on the left. The *n* must be a positive whole number or zero.

If you omit *n, wholenumber* must be a positive whole number or zero, and the returned result has no leading zeros.

Here are some examples:

```
D2X(9)        ->    '9'
D2X(129)      ->    '81'
D2X(129,1)    ->    '1'
D2X(129,2)    ->    '81'
D2X(129,4)    ->    '0081'
D2X(257,2)    ->    '01'
D2X(-127,2)   ->    '81'
D2X(-127,4)   ->    'FF81'
D2X(12,0)     ->    ''
```

Implementation maximum: The output string may not have more than 500 significant hexadecimal characters, though a longer result is possible if it has additional leading sign characters (0 and F).

ERRORTEXT
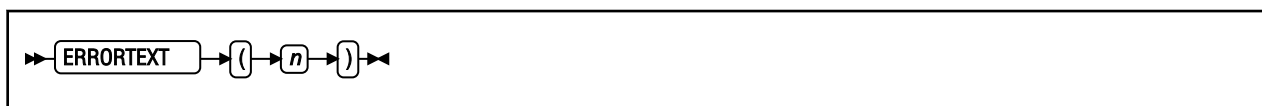
returns the REXX error message associated with error number *n.* The *n* must be in the range 0–99, and any other value is an error. Returns the null string if *n* is in the allowed range but is not a defined REXX error number. See Appendix A, "Error Numbers and Messages," on page 277 for a complete description of error numbers and messages.

Here are some examples:

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

EXTERNALS

This is a built-in function. See "EXTERNALS" on page 116 for a description.

FIND

FIND is a built-in function. See "FIND" on page 116 for a description. WORDPOS is the preferred built-in function for this type of word search; see "WORDPOS (Word Position)" on page 113 for a complete description.

FORM
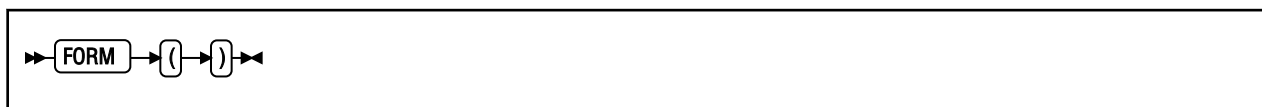
returns the current setting of NUMERIC FORM. See the NUMERIC instruction in "NUMERIC" on page 44 for more information.

Here is an example:

```
FORM()    ->    'SCIENTIFIC'  /* by default */
```

FORMAT
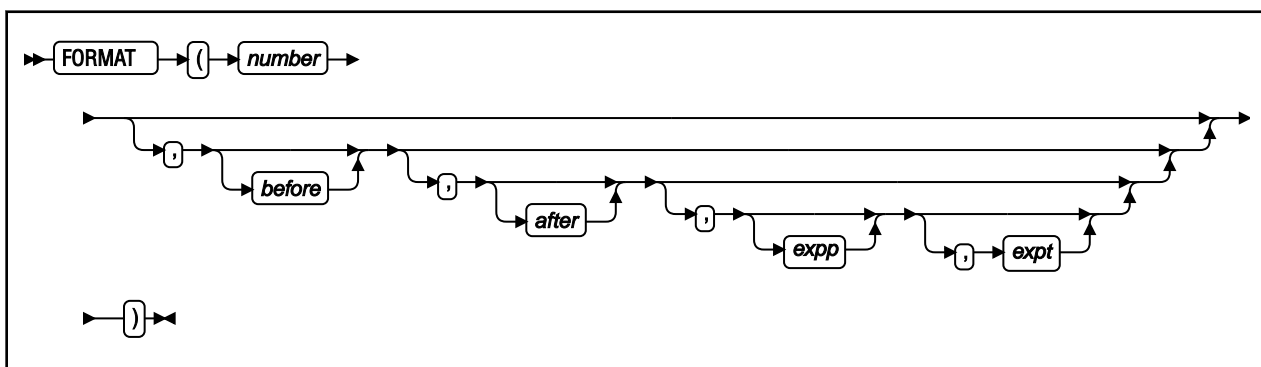
returns *number,* rounded and formatted.

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. The *expp* must be less than 10, but there is no limit on the other arguments.

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 in "NUMERIC" on page 44 for more information.

Here is an example:

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

INDEX

INDEX is a built-in function. See "INDEX" on page 117 for a description. POS is the preferred built-in function for obtaining the position of one string in another; see "POS (Position)" on page 93 for a complete description.

INSERT
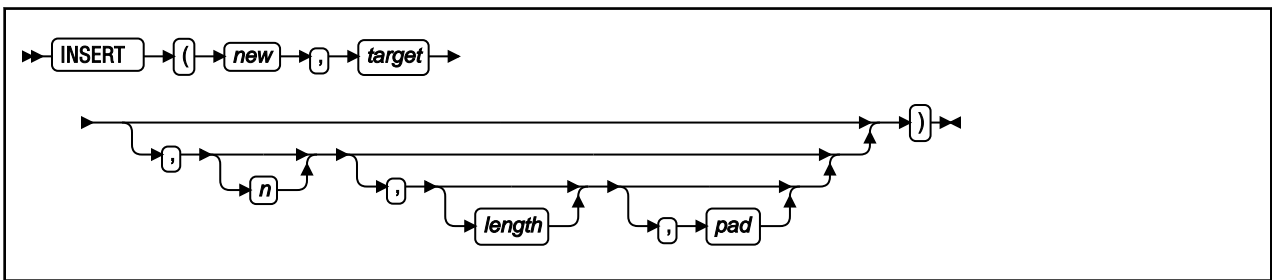
inserts the string *new*, padded or truncated to length *length*, into the string *target* after the *n*th character. The default value for *n* is 0, which means insert before the beginning of the string. If specified, *n* and *length* must be positive whole numbers or zero. If *n* is greater than the 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 built-in function. See "JUSTIFY" on page 117 for a description.

LASTPOS (Last Position)
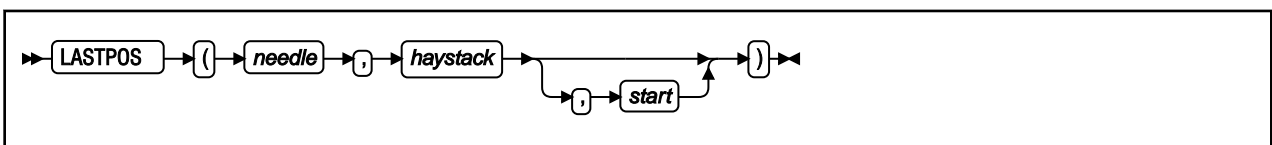
returns the position of the last occurrence of one string, *needle*, in another, *haystack*. (See also the POS function.) Returns 0 if *needle* is the null string or is not found. By default the search starts at the last

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
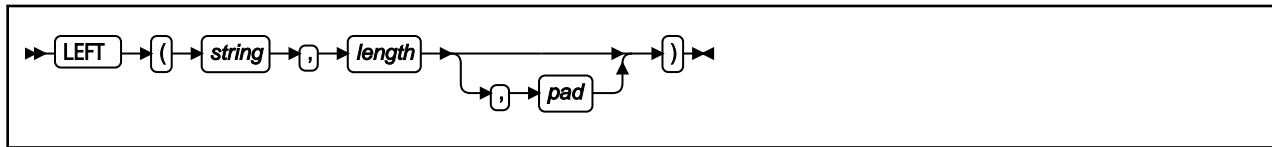
returns a string of length *length*, containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank. *length* must be a positive whole number or zero. The LEFT function is exactly equivalent to:
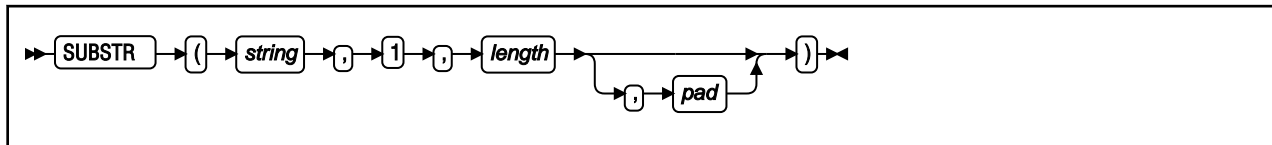
Here are some examples:

```
LEFT('abc d',8)         ->     'abc d   '
LEFT('abc d',8,'.')     ->     'abc d...'
LEFT('abc  def',7)      ->     'abc  de'
```

LENGTH
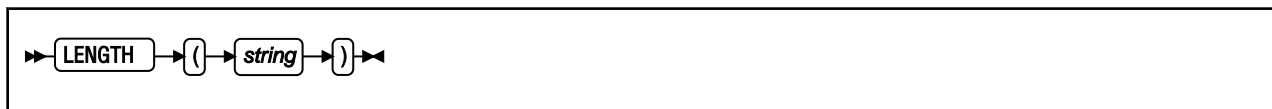
returns the length of *string*.

Here are some examples:

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

LINEIN (Line Input)
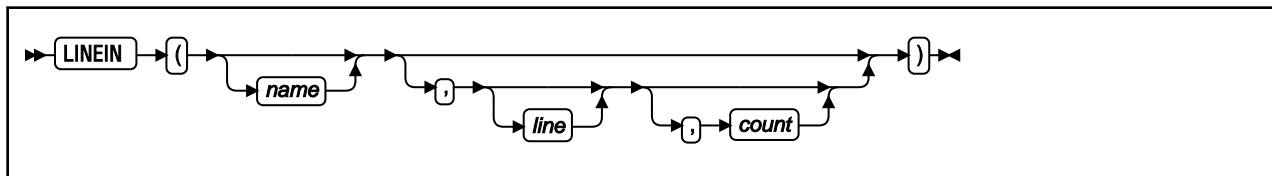
returns *count* lines read from the character input stream *name*. The *count* must be 1 or 0. (To understand the input and output functions, see Chapter 7, "Input and Output Streams," on page 167.) The form of *name* is also described in Chapter 7, "Input and Output Streams," on page 167. If you omit *name*, the line is read from the default input stream. The default *count* is 1.

For persistent streams, a read position is maintained for each stream. Any read from the stream starts at the current read position by default. (Under certain circumstances, a call to LINEIN returns a partial line. This can happen if the stream has already been read with the CHARIN function, and part but not all of a line (and its termination, if any) has already been read.) When the language processor completes reading, the read position is increased by the number of characters read. You can specify a *line* number to set the read position to the start of a particular line. This line number must be positive and within the bounds of the stream, and must not be specified for a transient stream. A value of 1 for *line* refers to the first line in the stream. (This is the initial read position when the stream is used for the first time after being opened).

If you give a *count* of 0, then the read position is set to the start of the specified *line*, but no characters are read and the null string is returned.

For transient streams, if a complete line is not available in the stream, then execution of the program usually stops until the line is complete. If, however, it is impossible for a line to be completed because of an error or other problem, the NOTREADY condition is raised (see "Errors During Input and Output" on page 173) and LINEIN returns whatever characters are available.

Here are some examples:

```
LINEIN(myfile)       ->   'MFC'    /* perhaps              */
LINEIN(myfile,5)     ->   'Line5'  /* perhaps              */
LINEIN(myfile,5,0)   ->   ''
LINEIN(myfile)       ->   'Line5'  /* after last call      */
LINEIN()             ->   'Hello'  /* would cause a        */
                                   /* VM READ if there is  */
                                   /* no string available  */
                                   /* in the default input */
                                   /* stream               */
```

Note: If the intention is to read complete lines from the default input stream, as in a simple dialogue with a user, use the PULL or PARSE PULL instruction instead for simplicity. The PARSE LINEIN instruction is also useful in certain cases. (See "PARSE" on page 48.)

LINEOUT (Line Output)
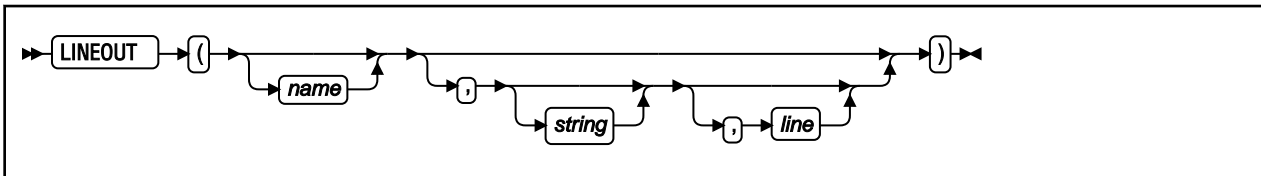
returns the count of lines remaining after attempting to write *string* to the character output stream *name*. (To understand the input and output functions, see Chapter 7, "Input and Output Streams," on page 167.) The count is either 0 (meaning the line was successfully written) or 1 (meaning that an error occurred while writing the line). The *string* can be the null string, in which case the only action is repositioning of the write position if repositioning is specified. Otherwise, the stream is closed.

The form of *name* is also described in Chapter 7, "Input and Output Streams," on page 167. If you omit *name*, the line is written to the default output stream.

For persistent streams, a write position is maintained for each stream. Any write to the stream starts at the current write position by default. (Under certain circumstances the characters written by a call to LINEOUT may be added to a partial line previously written to the stream with the CHAROUT routine. LINEOUT stops a line at the *end* of each call.) When the language processor completes writing, the write position is increased by the number of characters written. When the stream is first opened, the write position is at the end of the stream, so that calls to LINEOUT will append lines to the end of the stream.

You can specify a *line* number to set the write position to the start of a particular line in a persistent stream. This line number must be positive and within the bounds of the stream (though it can specify the line number immediately after the end of the stream). A value of 1 for *line* refers to the first line in the stream.

The output of the string may have associated side effects. That is, if *line* is specified and there are output characters previously written with CHAROUT to an incomplete line, this residual line is written out before the LINEOUT operation is performed. If *line* is not specified and there are output characters in the I/O buffer, which was previously written with CHAROUT, then *string* is appended to the existing data and the line written to the output stream. All lines written to an F-format output stream will be padded with blanks as necessary. If the string is too long, a NOTREADY condition will be raised. For V-format streams, no padding or truncation is done and if the output string is null, a null line is written to the stream.

Note: You will get an error if you try to overwrite a record with another record that has a different length.

You can omit the *string* for persistent streams. If you specify *line*, the write position is set to the start of the *line* that was given, nothing is written to the stream, and the function returns 0. If you specify neither *line* nor *string*, then any incomplete line previously written with CHAROUT will first be written to the stream with padding as appropriate, and the stream will be closed.

Execution of the program usually stops until the output operation is effectively complete. For example, when data is sent to a printer, the system accepts the data and returns control to REXX, even though the output data may not have been printed on the printer. REXX considers this to be complete, even though the data has not been printed. If, however, it is impossible for a line to be written, the NOTREADY condition is raised (see "Errors During Input and Output" on page 173), and LINEOUT returns a result of 1 (that is, the residual count of lines written).

Here are some examples:

```
LINEOUT(myfile,'Hi')     ->   0   /* usually        */
LINEOUT(myfile,'Hi',5)   ->   0   /* usually        */
LINEOUT(myfile,,6)       ->   0   /* now at line 6  */
LINEOUT(myfile)          ->   0   /* stream closed  */
LINEOUT(,'Hi')           ->   0   /* usually        */
LINEOUT(,'Hello')        ->   1   /* maybe          */
```

LINEOUT is often most useful when called as a subroutine. The residual line count is then available in the variable RESULT.

Here are some examples:

```
Call LINEOUT 'Output file','Hello'
Call LINEOUT ,'Hello'
```

Note: If the lines are to be written to the default output stream without the possibility of error, use the SAY instruction instead.

LINES (Lines Remaining)

returns the number of completed lines remaining in the character input stream *name*. If the stream has already been read with the CHARIN function, this can include an initial partial line. For persistent streams the count starts at the current read position. (To understand the input and output functions, see Chapter 7, "Input and Output Streams," on page 167.)

The form of *name* is also described in Chapter 7, "Input and Output Streams," on page 167. If you omit *name* or it is null, then the default input stream is tested.

Here are some examples:

```
LINES(myfile)    ->    7   /* 7 lines remain     */

LINES(myfile)    ->    0   /* at end of the file */
```

```
LINES()          ->   1    /* data remains in the  */
                           /* default input stream */
```

LINESIZE

This is a built-in function. See "LINESIZE" on page 118 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.

OVERLAY
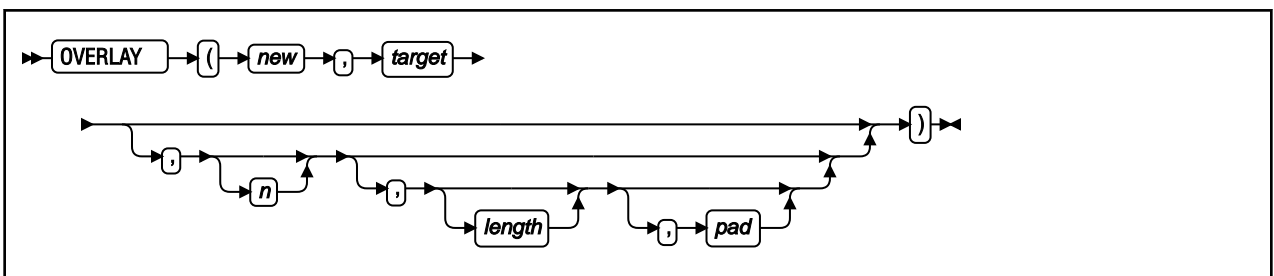
returns the string *target*, which, starting at the *n*th character, is overlaid with the string *new*, padded or truncated to length *length*. (The overlay may extend beyond the end of the original *target* string.) If you

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)
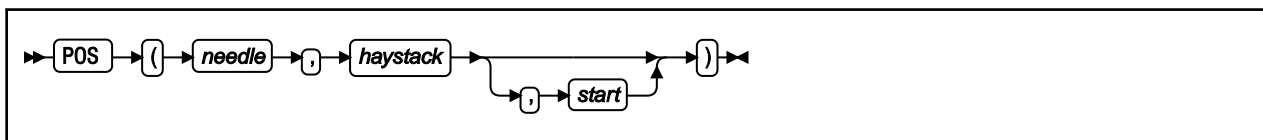
returns the position of one string, *needle*, in another, *haystack*. (See also the INDEX and LASTPOS functions.) Returns 0 if *needle* is the null string or is not found or if *start* is greater than the length of *haystack*. By default the search starts at the first character of *haystack* (that is, 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
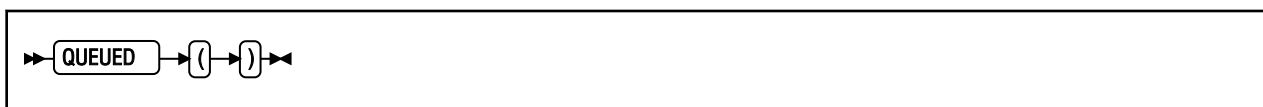
returns the number of lines remaining in the external data queue when the function is called. (See Chapter 7, "Input and Output Streams," on page 167 for a discussion of REXX input and output.) If no lines are remaining, a PULL or PARSE PULL reads from the terminal input buffer. If no terminal input is waiting, this causes a console read (VM READ).

Here is an example:

```
QUEUED()    ->    5    /* Perhaps */
```

RANDOM
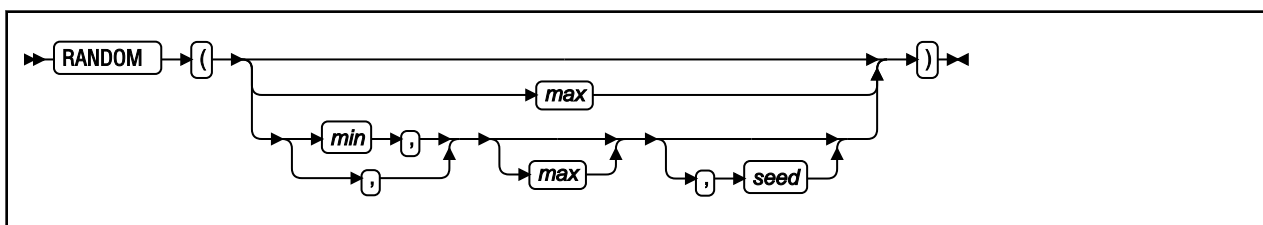
returns a quasi-random nonnegative whole number in the range *min* to *max* inclusive. If you specify *max* or *min* or both, *max* minus *min* cannot exceed 100000. The *min* and *max* default to 0 and 999, respectively. To start a repeatable sequence of results, use a specific *seed* as the third argument, as described in Note "1" on page 94. 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, the microsecond field of the time-of-day clock is used as the *seed*; and hence your program almost always 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
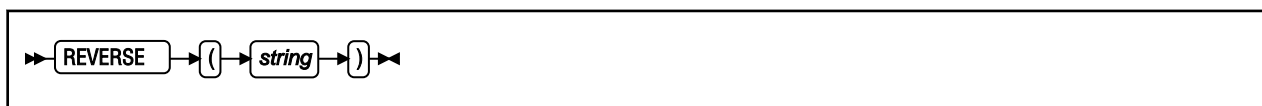
returns *string*, swapped end for end.

Here are some examples:

```
REVERSE('ABc.')    ->     '.cBA'
REVERSE('XYZ ')    ->     ' ZYX'
```

RIGHT
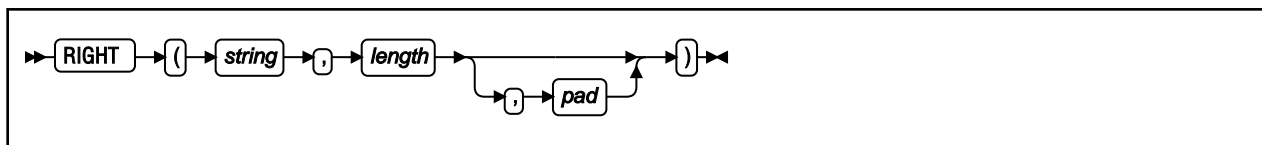
returns a string of length *length* containing the rightmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank. The *length* must be a positive whole number or zero.

Here are some examples:

```
RIGHT('abc  d',8)    ->     '  abc  d'
RIGHT('abc def',5)   ->     'c def'
RIGHT('12',5,'0')    ->     '00012'
```

SIGN

```
▶▶─ SIGN ─(─ number ─)─◀
```
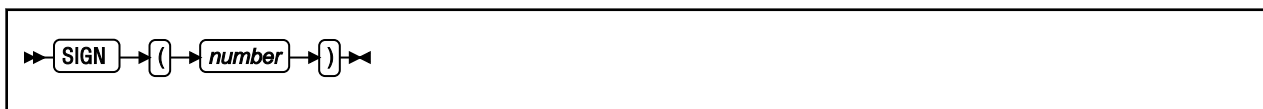
returns a number that indicates the sign of *number*. The *number* is first rounded according to standard REXX rules, just as though the operation number+0 had been carried out. Returns -1 if *number* is less than 0; returns 0 if it is 0; and returns 1 if it is greater than 0.

Here are some examples:

```
SIGN('12.3')      ->    1
SIGN(' -0.307')   ->   -1
SIGN(0.0)         ->    0
```

SOCKET

This is a CMS external function. See "SOCKET" on page 134 for a description.

SOURCELINE

```
▶▶─ SOURCELINE ─(─┬───────┬─)─◀
                  └─ n ─┘
```
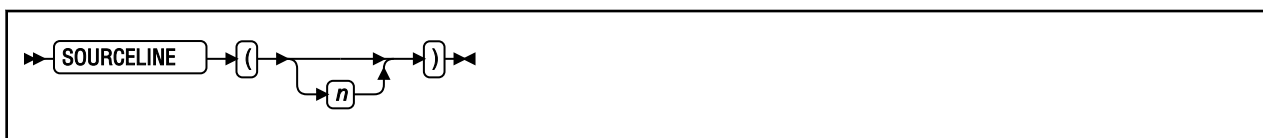
returns the line number of the final line in the program if you omit *n*, or returns the *n*th line in the program if you specify *n*. If specified, *n* must be a positive whole number and must not exceed the number of the final line in the program.

Here are some examples:

```
SOURCELINE()    ->   10
SOURCELINE(1)   ->   '/* This is a 10-line REXX program */'
```

SPACE

```
▶▶─ SPACE ─(─ string ─┬──────────────────────────┬─)─◀
                      └─,─┬─────┬─┬──────────────┬─┘
                         └─ n ─┘ └─,─ pad ─┘
```
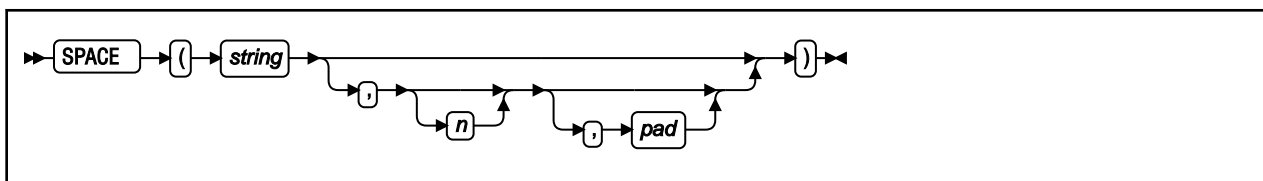
returns the blank-delimited words in *string* with *n pad* characters between each word. If you specify *n*, it must be a positive whole number or zero. If it is 0, all blanks are removed. Leading and trailing blanks are always removed. The default for *n* is 1, and the default *pad* character is a blank.

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

This is a CMS external function. See "STORAGE" on page 135.

STSI

This is a CMS external function. See "STSI" on page 136.

STREAM
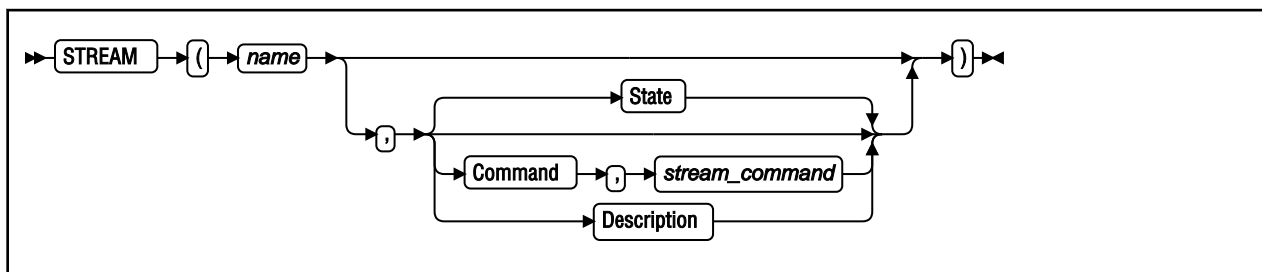
returns a string describing the state of, or the result of an operation upon, the character stream *name*. The result may depend on characteristics of the stream that you have specified in other uses of the STREAM function. (To understand the input and output functions, see Chapter 7, "Input and Output Streams," on page 167.) This function requests information on the state of an input or output stream or carries out some specific operation on the stream.

The first argument, *name*, specifies the stream to be accessed. The form of *name* is also described in Chapter 7, "Input and Output Streams," on page 167. The second argument can be one of the following strings that describe the action to be carried out. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Command
an operation (specified by the *stream_command* given as the third argument) is applied to the selected input or output stream. The *stream_command* string is a command in the list in "Stream Commands" on page 101. You can use the *stream_command* argument only with the Command option. The string that is returned depends on the command performed and may be the null string.

Description
returns a string that indicates the current state of the specified stream. The first part of the result is identical to the State option, but it returns more information: the returned string is followed by a colon, along with the return code and reason code returned by the last I/O on that stream, and all are separated by one blank. In certain cases, you will get additional information appended to the description string from the lower level routine in the form:

```
llname llretcode llreascode
```

(where *llname* is the lower level program name; *llretcode* is the lower level return code; and *llreascode* is the lower level reason code.)

You cannot specify *stream_command* with this option.

State
returns a string that indicates the current state of the specified stream. This is the default if no action is specified. You cannot specify *stream_command* with this option.

The strings returned are as follows:

ERROR
The stream has been subject to an erroneous operation (possibly during input, output, or through the STREAM function). See "Errors During Input and Output" on page 173. You may be able to obtain additional information about the error by invoking the STREAM function with a request for the description.

NOTREADY
The stream is known to be in a state such that usual input or output operations attempted upon it would raise the NOTREADY condition. (See "Errors During Input and Output" on page 173 For example, a simple input stream may have a defined length; an attempt to read that stream (with

the CHARIN or LINEIN built-in functions, perhaps) beyond that limit may make the stream unavailable until some operation resets the state of the stream.

READY

The stream is known to be in a state such that usual input or output operations may be attempted (this is the usual state for a stream, though it does not guarantee that any particular operation will succeed).

UNKNOWN

The state of the stream is unknown. That is, the stream is closed or has not yet been opened.

Here are some examples:

```
/* Possible results might be: */
STREAM(myfile)              ->    'READY'
STREAM(readitall)           ->    'NOTREADY'
STREAM(readitall,'D')       ->    'NOTREADY: 8 99509'
```

Note: The state (and operation) of an input or output stream is global to all REXX programs; it is not saved and restored across internal or external function and subroutine calls (including those calls that a CALL ON condition trap causes).

Stream Commands

The following stream commands:

- Open a stream for reading or writing
- Close a stream at the end of an operation
- Move the line read or line write position within a persistent stream (for example, a file)
- Get information about a stream (its existence, format, size, last edit or use date, and read/write position).

You must use the *stream_command* argument when—and only when—you select the Command operation. The syntax is:

►►─ STREAM ─→─ (─→─ name ─→─ , ─→─ 'C' ─→─ , ─→─ stream_command ─→─) ─►◄

In this form, the STREAM function itself returns a string corresponding to the given *stream_command* if the command is successful. If the command is unsuccessful, STREAM returns an error message string. For most error conditions, additional information is returned in the form of a return code and reason code.

See Appendix F, "Input and Output Return and Reason Codes," on page 307 for a full explanation of the return codes and reason codes generated by the CSL routines that perform the I/O. For the lower level routine codes not listed there, see the following books:

- *z/VM: CMS Callable Services Reference* and *z/VM: CP Messages and Codes*: For SFS files, minidisk files, or the program stack.
- *z/VM: CMS Macros and Functions Reference*: For spool files (reader, punch, and printer).
- *z/VM: CP Programming Services*: For CP diagnose codes. (The value in the reason code field is actually the condition code.)
- *z/VM: CP Commands and Utilities Reference*: For CP commands.

Command Strings

The argument *stream_command* can be any expression that REXX evaluates as one of the following command strings. (All capitalized letters are needed; case is insignificant.)

Notes:

¹ You can enter Options in any order.

OPEN *options*

opens the named stream. The stream may be opened for reading or writing by specifying an intent. Valid values for intent are:

READ

Open for reading only, object must exist.

WRITE

Open for read/write, object will be created if it does not exist.

NEW

Open for read/write, object must not exist and will be created.

REPLACE

Open for read/write, object will be created if it does not exist or will be replaced if it does exist. A replaced file is considered a new file, and, therefore, the record format and logical record length may change.

If an intent is not specified, then the open type will be determined by the capability of the named stream if known: READ for read-only streams and WRITE otherwise. If unknown, WRITE is used.

The rest of the options are described as follows:

LRECL *nnnnn*

used to specify the logical record length of the stream. If this parameter is not specified, the logical record length of the stream is used, if known. Otherwise, 1024 is used.

The value of LRECL is only used if a minidisk or SFS file is being created or replaced, otherwise it is ignored and the actual LRECL and RECFM of the stream is used.

Note: If the stream is an existing minidisk or SFS file and the open intent is not REPLACE, the LRECL and RECFM parameter is ignored and any subsequent output behavior (padding or truncation) will be consistent with the actual attributes of the file.

RECFM V or F
    specifies the record format of a stream, V (variable) or F (fixed). If RECFM F is specified and a minidisk or SFS file is being created, the LRECL default is 80. RECFM V will default to the length of the file if no LRECL is specified.

    The value of LRECL is only used if a minidisk or SFS file is being created or replaced, otherwise it is ignored and the actual LRECL and RECFM of the stream is used.

    Note: If the stream is an existing minidisk or SFS file and the open intent is not REPLACE, the LRECL and RECFM parameter is ignored and any subsequent output behavior (padding or truncation) will be consistent with the actual attributes of the file.

TEXT or BINARY
    specifies if LINEEND characters are significant in character I/O operations. BINARY means that all character codes may be present in the data stream and no indication of LINEEND characters will be provided or searched for. TEXT means that LINEEND characters are not included in the data stream and line ends should be noted. That is, LINEEND characters are appended to the end of each line when passing data to the user on character input operations; data to be written to a stream on character output operations is split at LINEEND characters. These LINEEND characters are never written to the data stream. Line operations are not affected by this parameter. TEXT is the default.

LINEEND xx
    specifies the line end character to be used when doing character-based operations on TEXT streams (ignored for line-based operations). The xx can be one or two hexadecimal digits that define the character to be used as the line end indicator. A leading 0 will be supplied if only one digit is given. These must be valid hexadecimal digits and contain no blanks. If this parameter is not specified, 15 is used. For portability, it is recommended that a value of 3F or less be used.

The STREAM function itself returns the string READY: followed by a unique stream identifier if the named stream is successfully opened. This unique identifier contains printable and nonprintable characters. If unsuccessful, the returned string consists of ERROR: followed by the return code and reason code from the open routine that failed, and then the words Open failed.

Examples:

```
stream(strinp,'c','open')                /* open stream    */
stream(strout,'c','open write')          /* open for write */
       /* open for read, LRECL is 80, character operations */
       /* should indicate line end as X'15'               */
stream(strinp,'c','open read lrecl 80 text lineend 15')
```

See "Stream Names Used by the Input and Output Functions" on page 168 for a code segment using this unique identifier.

CLOSE
    closes the named stream. The STREAM function itself returns READY: if the named stream is successfully closed. If the close is unsuccessful, the returned string is ERROR: followed by the return code and reason code from the failing routine and then Close failed. For an SFS file, this causes a close with commit. If an attempt is made to close an unopened stream, then STREAM returns a null string.

Example:

```
stream('MYDATA FILE','c','close')        /* close the file */
```

LINEPOS *offset type*
    sets the read or write line pointer to a specified *offset* within a persistent stream.

offset
>   a positive whole number that may have a modifier as follows:

>   **= or blank**
>>   specifies that *offset* is forward from the beginning of the stream. This is the default if no modifier is supplied.

>   **<**
>>   specifies that *offset* is backward from the end of the stream.

>   **+**
>>   specifies that *offset* is forward from the current read/write position.

>   **-**
>>   specifies that *offset* is backward from the current read/write position.

type
>   is either READ or WRITE, specifying that the read or write position is to be changed.

In order to position the line write pointer to the position just past the end of a stream, use an offset of <0. In order to position the line write pointer to the last record of a stream, use an offset of <1.

In order to use this command, the named stream must first be opened (with the OPEN stream command described above or implicitly with an I/O function call).

The STREAM function itself returns the new position in the stream if the read/write position is successfully located or an error message otherwise. This error message will include the string `ERROR:` followed by the return code and reason code from the failing routine, and then followed by the message, `Point failed`.

Examples:

```
                     /* move write pointer to line 2    */
stream(name,'Command','linepos 2 write')
                     /* move read pointer ahead 2 lines */
stream(name,'c','linepos +2' read)
                     /* move write pointer back 7 lines */
stream(name,'c','linepos -7 write')
fromend  = 125
                     /* move read pointer 125 lines     */
                     /* backward from the end           */
stream(name,'c','linepos <'fromend 'read')
```

In addition, the STREAM function returns specific information about a persistent stream when used with the following commands. Under certain error conditions, and when a stream does not exist, a null string is returned. (All capitalized letters are needed; case is insignificant.)

QUERY DATETIME
>   returns the date and time stamps of a stream (date and time when last modified). For example:

>   ```
>   stream('* RDRFILE CMSOBJECTS.','c','query datetime')
>   ```

>   The date is returned in the form:

>   ```
>   mm/dd/yyyy hh:mm:ss
>   ```

>   For a reader file, this is the date and time the file was created (punched). If the file is not open, the year is 0000.

>   For an SFS file or minidisk file, this is the date and time the file was last modified.

>   For all other objects, zeros are returned to indicate that the date and time are undetermined.

QUERY EXISTS
>   returns the full name of the object if it exists, or a null string otherwise. For example:

>   ```
>   stream('YOURDATA FILE *','c','query exists')
>   ```

For a reader file, this indicates whether the file is present in the virtual reader. If an asterisk (*) was specified for the spool ID, then this will return the full name (`spool_id RDRFILE CMSOBJECTS.`) of the first file in the reader if it exists.

For SFS and minidisk files, this indicates whether the file exists. For a minidisk file or an accessed SFS directory, if the file mode number was omitted or the file mode was specified as an asterisk (*), then the full file name will be returned. For SFS, the fully qualified name is returned.

QUERY FORMAT
returns the record format and logical record length of a stream. For example:

```
stream('MYFILE DATA A','c','query format')
```

This information is returned in the form:

```
recfm lrecl
```

For a virtual unit record device, the device's characteristics are evaluated according to device type such as reader, printer, and punch.

QUERY INFO
returns the record format, logical record length, number of records, date of last update, and time of last update. For example:

```
stream('MYDATA FILE A','c','query info')
```

The information is returned in the form:

```
recfm lrecl #records mm/dd/yyyy hh:mm:ss
```

Refer to the commands in this list for the various pieces (format, size, and datetime) for information on the content of the returned string.

QUERY LINEPOS READ
returns the value of the line read pointer for a persistent stream (or 0 if the stream cannot be read or is not open). For example:

```
stream('MYFILE DATA A','c','query linepos read')
```

QUERY LINEPOS WRITE
returns the value of the line write pointer for a persistent stream (or 0 if the stream cannot be written to or is not open). For example:

```
stream('MYFILE DATA A','c','query linepos write')
```

QUERY SIZE
returns the size in lines (or records) of a stream. For example:

```
stream(punch,'c','query size')
```

For minidisk and SFS files, this is the number of records in the file.

For a reader file, this is the number of card images in the reader file.

For a virtual punch or virtual printer, this is the number of records written to the device since it was opened if it is open and zero if it is closed.

Stream Commands

The following stream commands:

- Open a stream for reading or writing
- Close a stream at the end of an operation
- Move the line read or line write position within a persistent stream (for example, a file)

Functions

- Get information about a stream (its existence, format, size, last edit or use date, and read/write position).

You must use the *stream_command* argument when—and only when—you select the Command operation. The syntax is:

```
►►─ STREAM ─►─ ( ─►─ name ─►─ , ─►─ 'C' ─►─ , ─►─ stream_command ─►─ ) ─►◄
```

In this form, the STREAM function itself returns a string corresponding to the given *stream_command* if the command is successful. If the command is unsuccessful, STREAM returns an error message string. For most error conditions, additional information is returned in the form of a return code and reason code.

See Appendix F, "Input and Output Return and Reason Codes," on page 307 for a full explanation of the return codes and reason codes generated by the CSL routines that perform the I/O. For the lower level routine codes not listed there, see the following books:

- *z/VM: CMS Callable Services Reference* and *z/VM: CP Messages and Codes*: For SFS files, minidisk files, or the program stack.
- *z/VM: CMS Macros and Functions Reference*: For spool files (reader, punch, and printer).
- *z/VM: CP Programming Services*: For CP diagnose codes. (The value in the reason code field is actually the condition code.)
- *z/VM: CP Commands and Utilities Reference*: For CP commands.

Command strings

The argument *stream_command* can be any expression that REXX evaluates as one of the following command strings. (All capitalized letters are needed; case is insignificant.)

Options

> Notes:
>
> [1] You can enter Options in any order.

OPEN *options*
> opens the named stream. The stream may be opened for reading or writing by specifying an intent. Valid values for intent are:

READ
> Open for reading only, object must exist.

WRITE
> Open for read/write, object will be created if it does not exist.

NEW
> Open for read/write, object must not exist and will be created.

REPLACE
> Open for read/write, object will be created if it does not exist or will be replaced if it does exist. A replaced file is considered a new file, and, therefore, the record format and logical record length may change.

If an intent is not specified, then the open type will be determined by the capability of the named stream if known: READ for read-only streams and WRITE otherwise. If unknown, WRITE is used.

The rest of the options are described as follows:

LRECL *nnnnn*
> used to specify the logical record length of the stream. If this parameter is not specified, the logical record length of the stream is used, if known. Otherwise, 1024 is used.
>
> The value of LRECL is only used if a minidisk or SFS file is being created or replaced, otherwise it is ignored and the actual LRECL and RECFM of the stream is used.
>
> **Note:** If the stream is an existing minidisk or SFS file and the open intent is not REPLACE, the LRECL and RECFM parameter is ignored and any subsequent output behavior (padding or truncation) will be consistent with the actual attributes of the file.

RECFM V or F
> specifies the record format of a stream, V (variable) or F (fixed). If RECFM F is specified and a minidisk or SFS file is being created, the LRECL default is 80. RECFM V will default to the length of the file if no LRECL is specified.
>
> The value of LRECL is only used if a minidisk or SFS file is being created or replaced, otherwise it is ignored and the actual LRECL and RECFM of the stream is used.
>
> **Note:** If the stream is an existing minidisk or SFS file and the open intent is not REPLACE, the LRECL and RECFM parameter is ignored and any subsequent output behavior (padding or truncation) will be consistent with the actual attributes of the file.

TEXT or BINARY
> specifies if LINEEND characters are significant in character I/O operations. BINARY means that all character codes may be present in the data stream and no indication of LINEEND characters will be provided or searched for. TEXT means that LINEEND characters are not included in the data stream and line ends should be noted. That is, LINEEND characters are appended to the end of each line when passing data to the user on character input operations; data to be written to a stream on character output operations is split at LINEEND characters. These LINEEND characters are never written to the data stream. Line operations are not affected by this parameter. TEXT is the default.

LINEEND *xx*
> specifies the line end character to be used when doing character-based operations on TEXT streams (ignored for line-based operations). The *xx* can be one or two hexadecimal digits that define the character to be used as the line end indicator. A leading 0 will be supplied if only one

digit is given. These must be valid hexadecimal digits and contain no blanks. If this parameter is not specified, 15 is used. For portability, it is recommended that a value of 3F or less be used.

The STREAM function itself returns the string `READY:` followed by a unique stream identifier if the named stream is successfully opened. This unique identifier contains printable and nonprintable characters. If unsuccessful, the returned string consists of `ERROR:` followed by the return code and reason code from the open routine that failed, and then the words `Open failed`.

Examples:

```
stream(strinp,'c','open')              /* open stream     */
stream(strout,'c','open write')        /* open for write */
      /* open for read, LRECL is 80, character operations */
      /* should indicate line end as X'15'            */
stream(strinp,'c','open read lrecl 80 text lineend 15')
```

See "Stream Names Used by the Input and Output Functions" on page 168 for a code segment using this unique identifier.

CLOSE
closes the named stream. The STREAM function itself returns `READY:` if the named stream is successfully closed. If the close is unsuccessful, the returned string is `ERROR:` followed by the return code and reason code from the failing routine and then `Close failed`. For an SFS file, this causes a close with commit. If an attempt is made to close an unopened stream, then STREAM returns a null string.

Example:

```
stream('MYDATA FILE','c','close')      /* close the file */
```

LINEPOS *offset type*
sets the read or write line pointer to a specified *offset* within a persistent stream.

offset
a positive whole number that may have a modifier as follows:

= or blank
specifies that *offset* is forward from the beginning of the stream. This is the default if no modifier is supplied.

<
specifies that *offset* is backward from the end of the stream.

+
specifies that *offset* is forward from the current read/write position.

-
specifies that *offset* is backward from the current read/write position.

type
is either READ or WRITE, specifying that the read or write position is to be changed.

In order to position the line write pointer to the position just past the end of a stream, use an offset of <0. In order to position the line write pointer to the last record of a stream, use an offset of <1.

In order to use this command, the named stream must first be opened (with the OPEN stream command described above or implicitly with an I/O function call).

The STREAM function itself returns the new position in the stream if the read/write position is successfully located or an error message otherwise. This error message will include the string `ERROR:` followed by the return code and reason code from the failing routine, and then followed by the message, `Point failed`.

Examples:

```
                    /* move write pointer to line 2    */
stream(name,'Command','linepos 2 write')
```

```
                        /* move read pointer ahead 2 lines */
stream(name,'c','linepos +2' read)
                        /* move write pointer back 7 lines */
stream(name,'c','linepos -7 write')
fromend  = 125
                        /* move read pointer 125 lines    */
                        /* backward from the end          */
stream(name,'c','linepos <'fromend 'read')
```

In addition, the STREAM function returns specific information about a persistent stream when used with the following commands. Under certain error conditions, and when a stream does not exist, a null string is returned. (All capitalized letters are needed; case is insignificant.)

QUERY DATETIME

returns the date and time stamps of a stream (date and time when last modified). For example:

```
stream('* RDRFILE CMSOBJECTS.','c','query datetime')
```

The date is returned in the form:

```
mm/dd/yyyy hh:mm:ss
```

For a reader file, this is the date and time the file was created (punched). If the file is not open, the year is 0000.

For an SFS file or minidisk file, this is the date and time the file was last modified.

For all other objects, zeros are returned to indicate that the date and time are undetermined.

QUERY EXISTS

returns the full name of the object if it exists, or a null string otherwise. For example:

```
stream('YOURDATA FILE *','c','query exists')
```

For a reader file, this indicates whether the file is present in the virtual reader. If an asterisk (*) was specified for the spool ID, then this will return the full name (`spool_id RDRFILE CMSOBJECTS.`) of the first file in the reader if it exists.

For SFS and minidisk files, this indicates whether the file exists. For a minidisk file or an accessed SFS directory, if the file mode number was omitted or the file mode was specified as an asterisk (*), then the full file name will be returned. For SFS, the fully qualified name is returned.

QUERY FORMAT

returns the record format and logical record length of a stream. For example:

```
stream('MYFILE DATA A','c','query format')
```

This information is returned in the form:

```
recfm lrecl
```

For a virtual unit record device, the device's characteristics are evaluated according to device type such as reader, printer, and punch.

QUERY INFO

returns the record format, logical record length, number of records, date of last update, and time of last update. For example:

```
stream('MYDATA FILE A','c','query info')
```

The information is returned in the form:

```
recfm lrecl #records mm/dd/yyyy hh:mm:ss
```

Refer to the commands in this list for the various pieces (format, size, and datetime) for information on the content of the returned string.

QUERY LINEPOS READ
returns the value of the line read pointer for a persistent stream (or 0 if the stream cannot be read or is not open). For example:

```
stream('MYFILE DATA A','c','query linepos read')
```

QUERY LINEPOS WRITE
returns the value of the line write pointer for a persistent stream (or 0 if the stream cannot be written to or is not open). For example:

```
stream('MYFILE DATA A','c','query linepos write')
```

QUERY SIZE
returns the size in lines (or records) of a stream. For example:

```
stream(punch,'c','query size')
```

For minidisk and SFS files, this is the number of records in the file.

For a reader file, this is the number of card images in the reader file.

For a virtual punch or virtual printer, this is the number of records written to the device since it was opened if it is open and zero if it is closed.

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)
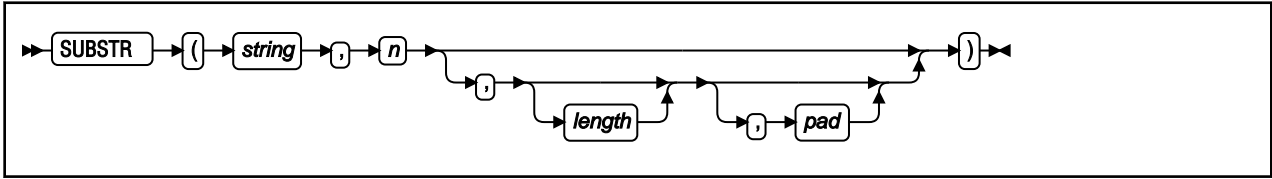
returns the substring of *string* that begins at the *n*th character and is of length *length*, padded with *pad* if necessary. The *n* must be a positive whole number. If *n* is greater than LENGTH(`string`), then only pad characters are returned.

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
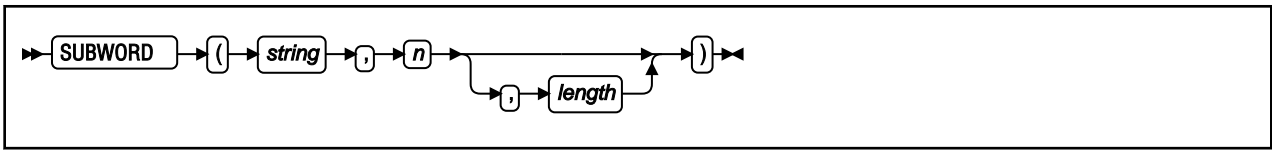
returns the substring of *string* that starts at the *n*th word, and is up to *length* blank-delimited words. The *n* must be a positive whole number. If you omit *length*, it defaults to the number of remaining words in *string*. The returned string never has leading or trailing blanks, but includes 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
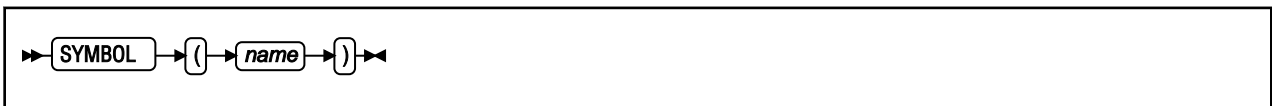
returns the state of the symbol named by *name*. Returns BAD if *name* is not a valid REXX symbol. Returns VAR if it is the name of a variable (that is, a symbol that has been assigned a value). Otherwise returns LIT, indicating that it is either a constant symbol or a symbol that has not yet been assigned 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 */
```

TIME

```
>>--[TIME]--(---+---------+---)--><
                |         |
                +-[option]+
```

returns the local time in the 24-hour clock format: hh:mm:ss (hours, minutes, and seconds) by default, for example, 04:41:37.

You can use the following *option*s to obtain alternative formats, or to gain access to the elapsed-time clock. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Civil
returns the time in Civil format: hh:mmxx. The hours may take the values 1 through 12, and the minutes the values 00 through 59. The minutes are followed immediately by the letters am or pm. This distinguishes times in the morning (12 midnight through 11:59 a.m.—appearing as 12:00am through 11:59am) from noon and afternoon (12 noon through 11:59 p.m.—appearing as 12:00pm through 11:59pm). The hour has no leading zero. The minute field shows the current minute (rather than the nearest minute) for consistency with other TIME results.

Elapsed
returns sssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock (described later) was started or reset. The number has no leading zeros or blanks, and the setting of NUMERIC DIGITS does not affect the number. The fractional part always has six digits.

Hours
returns up to two characters giving the number of hours since midnight in the format: hh (no leading zeros or blanks, except for a result of 0).

Long
returns time in the format: hh:mm:ss.uuuuuu (uuuuuu is the fraction of seconds, in microseconds). The first eight characters of the result follow the same rules as for the Normal form, and the fractional part is always six digits.

Minutes
returns up to four characters giving the number of minutes since midnight in the format: mmmm (no leading zeros or blanks, except for a result of 0).

Normal
returns the time in the default format hh:mm:ss, as described previously. The hours can have the values 00 through 23, and minutes and seconds, 00 through 59. All these are always two digits. Any fractions of seconds are ignored (times are never rounded up). **This is the default**.

Reset
returns sssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock (described later) was started or reset and also resets the elapsed-time clock to zero. The number has no leading zeros or blanks, and the setting of NUMERIC DIGITS does not affect the number. The fractional part always has six digits.

Seconds
returns up to five characters giving the number of seconds since midnight in the format: sssss (no leading zeros or blanks, except for a result of 0).

Here are some examples, assuming that the time is 4:54 p.m.:

```
TIME()      ->   '16:54:22'
TIME('C')   ->   '4:54pm'
TIME('H')   ->   '16'
TIME('L')   ->   '16:54:22.123456'   /* Perhaps */
TIME('M')   ->   '1014'              /* 54 + 60*16 */
TIME('N')   ->   '16:54:22'
TIME('S')   ->   '60862'  /* 22 + 60*(54+60*16) */
```

The elapsed-time clock:

You can use the TIME function to measure real (elapsed) time intervals. On the first call in a program to TIME('E') or TIME('R'), the elapsed-time clock is started, and either call returns 0. From then on, calls to TIME('E') and to TIME('R') return the elapsed time since that first call or since the last call to TIME('R').

The clock is saved across internal routine calls, which is to say that an internal routine inherits the time clock its caller started. Any timing the caller is doing is not affected, even if an internal routine resets the clock. An example of the elapsed-time clock:

```
time('E')   ->    0          /* The first call */
/* pause of one second here */
time('E')   ->    1.002345   /* or thereabouts */
/* pause of one second here */
time('R')   ->    2.004690   /* or thereabouts */
/* pause of one second here */
time('R')   ->    1.002345   /* or thereabouts */
```

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
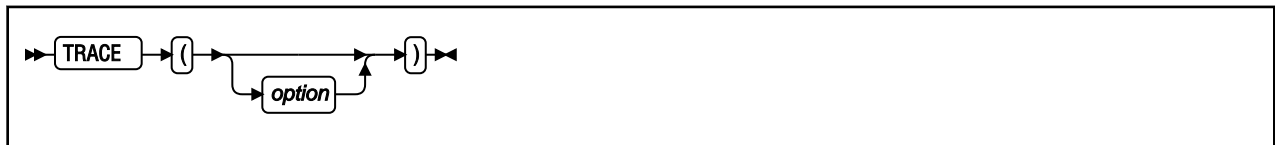
returns trace actions currently in effect and, optionally, alters the setting.

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 in "Alphabetic Character (Word) Options" on page 62.) 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

```
          ┌──────────────────────────────────────────────────────────────────┐
►►─ TRANSLATE( ─── string ─┤                                                    ├─ ) ─►◄
                          │  ┌─ , ─┐        ┌─ , ─┐                    │
                          └────┴─ tableo ─┴──┴─────┴── tablei ──┬─ , ─ pad ─┘
```
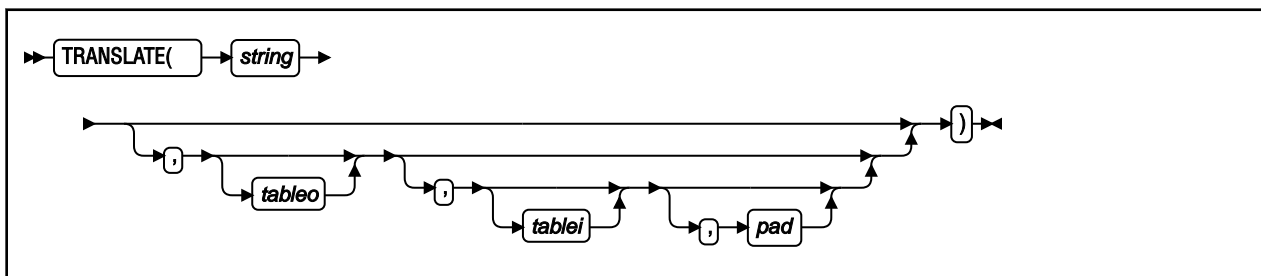
returns *string* with each character translated to another character or unchanged. You can also use this function to reorder the characters in *string*.

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)

```
►►─ TRUNC ─ ( ─ number ─┬───────────┬─ ) ─►◄
                        └─ , ─ n ─┘
```
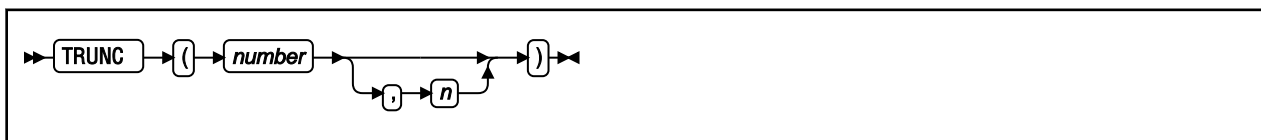
returns the integer part of *number* and *n* decimal places. The default *n* is 0 and returns an integer with no decimal point. If you specify *n*, it must be a positive whole number or zero. The *number* is first rounded according to standardREXX rules, just as though the operation number+0 had been 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 built-in function. See "USERID" on page 118 for a description.

VALUE

```
►►─ VALUE ─( ─ name ─┬──────────────────────────────────────┬─ ) ─►◄
                     └─ , ─┬──────────┬─┬──────────────────┬─┘
                          └ newvalue ┘ └─ , ─ selector ────┘
```
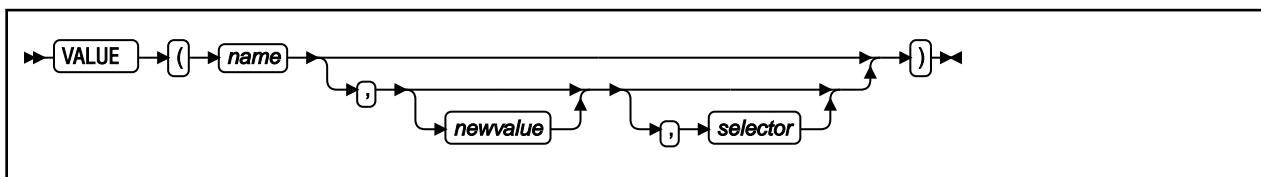
returns the value of the symbol that *name* (often constructed dynamically) represents and optionally assigns it a new value. By default, VALUE refers to the current REXX-variables environment, but CMS global variables may be selected. If you use the function to refer to REXX variables, then *name* must be a valid REXX symbol. (You can confirm this by using the SYMBOL function.) Lowercase characters in *name* are translated to uppercase. Substitution in a compound name (see "Compound Symbols" on page 14) 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 */
```

The third argument, *selector*, is of the form:

```
►►─┬─ GLOBAL ──┬─┬─────────────────┬─►◄
   ├─ LASTING ─┤ └─ group_name ─────┘
   └─ SESSION ─┘
```
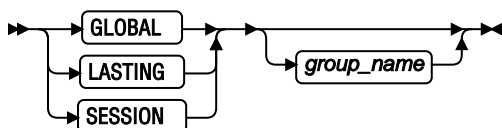
When this third argument is used, the VALUE function manipulates CMS global variables. The value of a variable is always retrieved from the in-storage table, no matter which of the three forms of *selector* is used. When you set the value of a variable, *selector* specifies which CMS global pool to use. The first form, GLOBAL, sets the variable in the in-storage table. The second form, LASTING, sets the variable in the in-storage table and also in the LASTING GLOBALV file. The third form, SESSION, sets the variable in the in-storage table and also in the SESSION GLOBALV file. These are the only accepted forms for *selector*. In these cases, the variable *name* need not be a valid REXX symbol. The argument *group_name* is the GLOBALV group name containing the variable. If this is not specified, UNNAMED is assumed. When VALUE sets or changes the value of a global variable, the new value is retained after the REXX procedure ends.

Here are some examples:

```
/* Given that a global variable Fred in the group MyGroup   */
/* has a value of 4                                         */
global = 'GLOBAL MyGroup'
say VALUE('Fred',7,global)     /* says '4' and assigns     */
                               /* Fred a new value of 7     */

say VALUE('Fred',,global)      /* says '7'                  */

/* After this procedure ends, Fred still has a value of 7    */
```

Note:

1. 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. A reference to CMS global variables never raises NOVALUE.

2. If you specify the *name* as a single literal string and omit *newvalue* and *selector*, 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 6, "Conditions and Condition Traps," on page 161.)

VERIFY
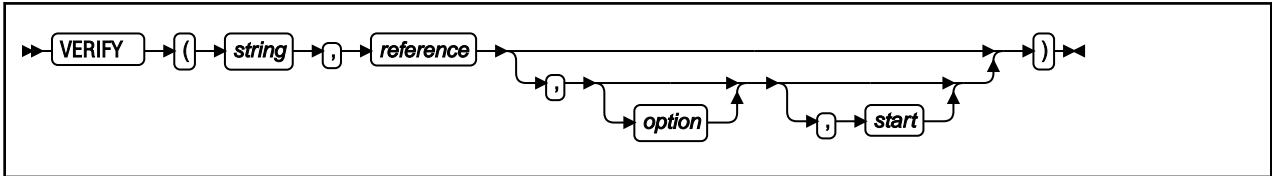
returns a number that, by default, indicates whether *string* is composed only of characters from *reference*; returns 0 if all characters in *string* are in *reference*, or returns the position of the first character in *string* **not** in *reference*.

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
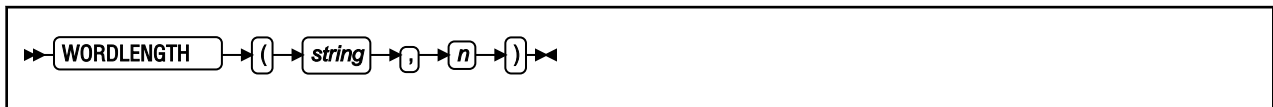
returns the length of the *n*th blank-delimited word in *string* or returns 0 if fewer than *n* words are in *string*. The *n* must be a positive whole number.

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)
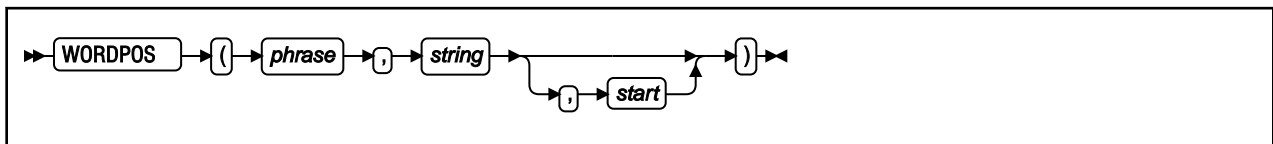
returns the word number of the first word of *phrase* found in *string* or returns 0 if *phrase* contains no words or if *phrase* is not found. Multiple blanks between words in either *phrase* or *string* are treated as a single blank for the comparison, but otherwise the words must match exactly.

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)
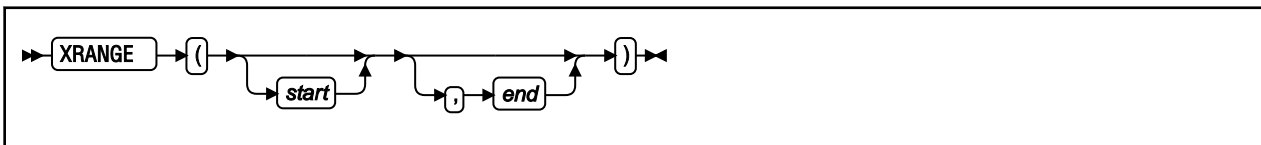
returns a string of all valid 1-byte encodings (in ascending order) between and including the values *start* and *end*. The default value for *start* is '00'x, and the default value for *end* is 'FF'x. If *start* is greater than *end*, the values wrap from 'FF'x to '00'x. If specified, *start* and *end* 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)
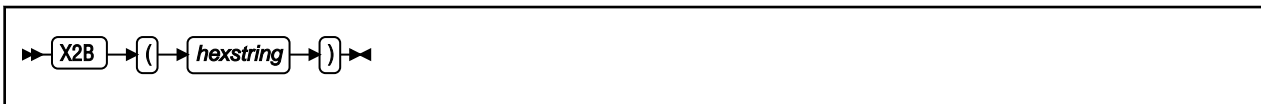
returns a string, in character format, that represents *hexstring* converted to binary. The *hexstring* is a string of hexadecimal characters. It can be of any length. Each hexadecimal character is converted to a string of four binary digits. You can optionally include blanks in *hexstring* (at byte 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 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.

Function Packages

If an external function or subroutine that is in a **function package** known to the language processor is called, the language processor automatically loads the **function package** before calling the function. To the general user with adequate virtual storage, the functions that have been provided in packages seem like ordinary built-in functions.

The language processor searches each of the function packages named below, if it is installed.

RXUSERFN
> This is the name of a package that the general user may write. The package would be written in assembler language and would contain a number of functions and their common subroutines. For a description of assembler language interfaces to the language processor, see Chapter 8, "System Interfaces," on page 177. For a description of function packages, see "Function Packages" on page 189.

RXLOCFN
> Similarly, this is the name of a package that system support people at your installation may write.

RXSYSFN
> This is the name of the additional function package that both system support personnel and general users can create and use.

The language processor searches for a function in the packages in the order given previously. See "Search Order" on page 68 for the complete search order.

Additional Built-in Functions Provided in VM

In addition to the built-in functions REXX provides, the following functions are provided in the VM environment.

EXTERNALS
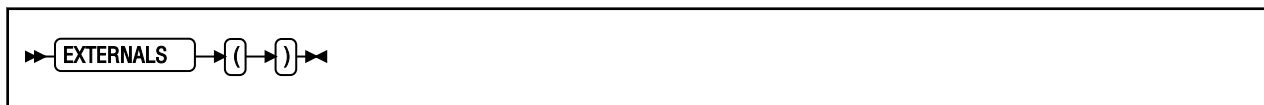
returns the number of elements in the terminal input buffer (system external event queue), that is, the number of logical typed-ahead lines, if any. See PARSE EXTERNAL in "PARSE" on page 48 for a description of this queue.

Here is an example:

```
EXTERNALS()    ->    0    /* Usually */
```

FIND

WORDPOS is the preferred built-in function for this type of word search. See "WORDPOS (Word Position)" on page 113 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
```

INDEX

POS is the preferred built-in function for obtaining the position of one string in another. See "POS (Position)" on page 93 for a complete description.
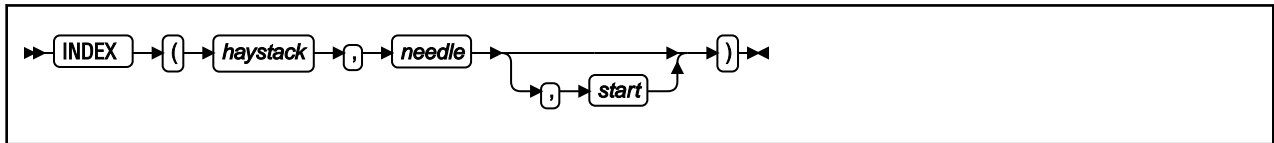
returns the character position of one string, *needle,* in another, *haystack,* or returns 0 if the string *needle* is not found or is a null string. By default the search starts at the first character of *haystack* (*start* has the value 1). You can override this by specifying a different *start* point, 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
```

JUSTIFY
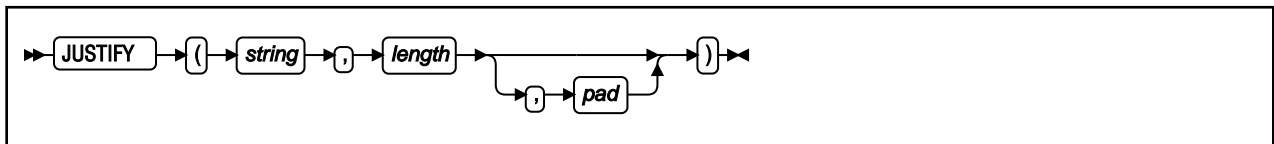
returns *string* formatted by adding *pad* characters between blank-delimited words to justify to both margins. This is done to width *length* (*length* must be a positive whole number or zero). The default *pad* character is a blank.

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'
```

LINESIZE

```
►►─ LINESIZE ─►─(─►─)─►◄
```

returns the current terminal line width (the point at which the language processor breaks lines displayed using the SAY instruction). Returns a value of 0 if any of the following is true:

- The language processor cannot determine the terminal line size
- The virtual machine is disconnected
- The command CP TERMINAL LINESIZE OFF is in effect.

Note: You can set the terminal width with the CP TERM LINESIZE command. When not in full-screen CMS, the terminal line width is limited to the CMS maximum of 130. When in full-screen CMS, LINESIZE always returns a value of 999999999.

Testing for disconnect status using LINESIZE is not recommended because a connected user can enter CP TERM LINESIZE OFF, in which case LINESIZE returns 0. A proper way of checking for disconnect status is:

```
/*  Returns 1 if disconnected, else returns 0  */
disc: return (substr(diag(24,-1),13,1)¬=0)
```

USERID

```
►►─ USERID ─►─(─►─)─►◄
```
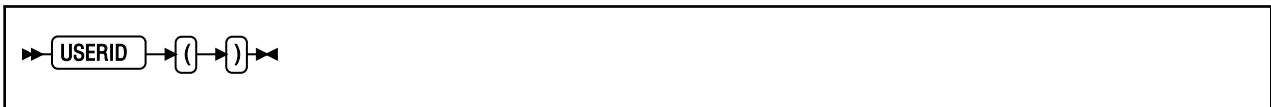
returns the system-defined User Identifier.

Here is an example:

```
USERID()     ->     'ARTHUR' /* Maybe */
```

External Functions and Routines Provided in VM

The following are additional external functions and routines provided in the VM environment: APILOAD includes a binding file in a REXX program, CMSFLAG returns the setting of certain indicators, CSL calls CSL (callable services library) routines, DIAG and DIAGRC issue special commands to CP, SOCKET provides access to the TCP/IP socket interface, and STORAGE inspects or alters the main storage of your virtual machine.

APILOAD

```
►►─ APILOAD ─►─(─►─ binding_file_name ─►─)─►◄
```

includes a binding file in a REXX program. A binding file declares the external functions, constants, return codes, and reason codes used by a set of CSL routines. The APILOAD function:

1. Takes as input the file name of a binding file (the file type of a binding file is COPY)
2. Reads the contents of the specified binding file
3. Creates and loads the variables in the invoking program.

The previous syntax shows how to call APILOAD as a function, but the preferred way is as a subroutine. The format of that is:

►►─ CALL APILOAD ─►─ ' ─►─ binding_file_name ─►─ ' ─►◄

A binding file may contain one or both of the following:

- The names of other REXX binding files
- The names of individual REXX variables.

For example, the VMREXMT binding file contains a list of all the individual CMS application multitasking binding files.

A REXX binding file consists of comments and simple assignment statements. Simple assignment statements are of the form $variable$ = token. Valid tokens are:

- Literal Strings
- Hexadecimal Strings
- Numbers
- Symbols
- Operators
- Special Characters.

Note:

1. Comments in binding files are processed in the same manner as comments in a REXX program.
2. Interpretation is performed on the right-hand side of an assignment statement in a binding file.
3. Logic statements (for example, IF/THEN/ELSE, SELECT) can have unpredictable results. These should be used with extreme care.

CMSFLAG

►►─ CMSFLAG ─►─ (─►─ flag ─►─) ─►◄

returns the value 1 or 0 depending on the setting of *flag*. Specify any one of the following *flag* names. (No abbreviations are allowed.) For more information on the following flags, see *z/VM: CMS Commands and Utilities Reference*.

ABBREV
    returns 1 if, when searching the synonym tables, truncations are accepted; otherwise, returns 0. Set by SET ABBREV ON; reset by SET ABBREV OFF.

AUTOREAD
    returns 1 if a console read is to be issued immediately after command execution; otherwise, returns 0. Set by SET AUTOREAD ON; reset by SET AUTOREAD OFF.

CMSTYPE
    returns 1 if console output is to be displayed (or typed) within an exec; returns 0 if console output is to be suppressed. Set by SET CMSTYPE RT or the immediate command RT (Resume Typing). Reset by SET CMSTYPE HT or the immediate command HT (Halt Typing).

DOS

returns 1 if your virtual machine is in the DOS environment; otherwise, returns 0. Set by SET DOS ON; reset by SET DOS OFF.

EXECTRAC

returns 1 if EXEC Tracing is turned on (equivalent to the TRACE prefix option ?); otherwise, returns 0. Set by SET EXECTRAC ON or the immediate command TS. Reset by SET EXECTRAC OFF or the immediate command TE. (See "Interrupting Execution and Controlling Tracing" on page 206.)

IMPCP

returns 1 if commands that CMS does not recognize are to be passed to CP; otherwise, returns 0. Set by SET IMPCP ON; reset by SET IMPCP OFF. Applies to commands issued from the CMS command line and also to REXX clauses that are commands to the CMS environment.

IMPEX

returns 1 you can call execs by file name; otherwise, returns 0. Set by SET IMPEX ON; reset by SET IMPEX OFF. Applies to commands issued from the CMS command line and also to REXX clauses that are commands to the CMS environment.

PROTECT

returns 1 if the CMS nucleus is storage-protected; otherwise, returns 0. Set by SET PROTECT ON; reset by SET PROTECT OFF.

RELPAGE

returns 1 if pages are to be released after certain commands have completed execution; otherwise, returns 0. Set by SET RELPAGE ON; reset by SET RELPAGE OFF.

SUBSET

returns 1 if you are in the CMS subset; otherwise, returns 0. Set by SUBSET (some editors issue this command); reset by RETURN. (For details, see "CMS subset" in the reference manual of the editor you are using).

XA

returns 1 if the program is executing in an XA, XC, or Z virtual machine; returns 0 if the program is executing in a 370 virtual machine.

XC

returns 1 if the program is executing in an XC virtual machine; returns 0 if the program is executing in an XA or 370 virtual machine.

Note: The XA flag also returns a value of 1 if the program is running in an XC virtual machine.

YEAR2000

returns 1 if both CP and CMS have the support for the year 2000 and beyond; returns 0 (zero) if either CP or CMS does not have the support for the year 2000 and beyond.

Z

returns 1 if the program is executing in an XA or Z virtual machine and running ZCMS.

370

returns 1 if the program is executing in a 370 virtual machine; returns 0 if the program is executing in an XA or XC virtual machine.

Note: Only CMS levels prior to CMS Level 12 will execute in a 370 virtual machine.

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

CSL
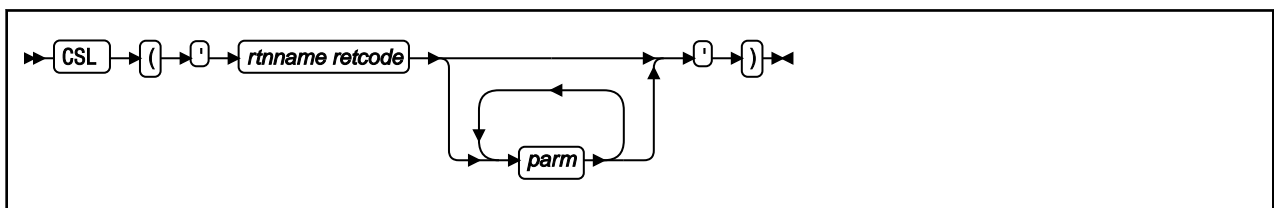
calls a routine that resides in a callable services library (CSL). Unlike other REXX functions (which use commas to separate expressions), the CSL function uses blanks to separate the parameters.

The previous syntax shows how to call CSL as a function, but the preferred way is as a subroutine. The format of that is:

```
>>──CALL CSL──'──rtnname retcode──┬─────────────────┬──'──><
                                  │  ┌───────────┐  │
                                  └──┴── parm ───┴──┘
```
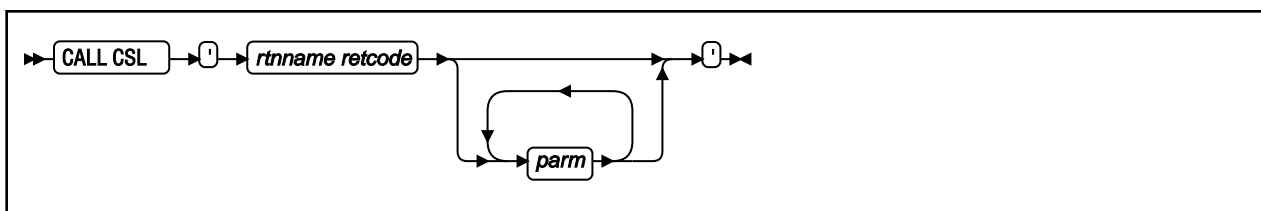
rtnname
  is the name of the CSL routine to be called.

retcode
  is the name of a variable to receive the return code from the CSL routine. This return code value is also returned as the value of the function call.

parm
  is the name of one or more parameters to be passed to the communications routine. The number and type of these parameters are routine-dependent. A parameter being passed must be the name of a variable.

Usage Notes

1. Use CSL in a REXX program to call:

   - Routines in the z/VM-supplied VMLIB callable services library that do tasks such as:

     – Perform CMS file system management or file pool administration functions
     – Call the CMS extract/replace facility
     – Use the CMS program stack

     See *z/VM: CMS Callable Services Reference* for more information about these routines.

   - Routines in the z/VM-supplied VMMTLIB callable services library that do tasks such as:

     – Perform CMS application multitasking functions. See *z/VM: CMS Application Multitasking* for more information.
     – Get the value set for the workstation display address. See *z/VM: CMS Callable Services Reference* for more information.

   However, *do not* use CSL to call:

   - z/VM-supplied routines that perform program-to-program communications. These routines are part of CPI Communications and must be called in a REXX program by using the ADDRESS CPICOMM instruction. For more information, see Chapter 12, "Invoking Communications Routines," on page 213.

   - z/VM-supplied routines that perform resource recovery. These routines are part of the CPI resource recovery interface and must be called in a REXX program by using the ADDRESS CPIRR instructions. For more information, see Chapter 13, "Invoking Resource Recovery Routines," on page 215.

   - OPENVM-type CSL routines, such as OpenExtensions callable services. These routines may not follow the usual syntax conventions for CSL routines and must be called in a REXX program by using the ADDRESS OPENVM instruction. For more information, see Chapter 14, "Invoking OPENVM Routines," on page 217.

2. Before using CSL to call CMS application multitasking routines, the WorkstationGetAddress routine, or routines from VMLIB that have names longer than eight characters (such as StackBufferCreate), you

should use the APILOAD function to include the appropriate programming language binding files into your program. Binding files declare the external functions, constants, return codes, reason codes, and other values used by the routines. For more information about APILOAD, see "APILOAD" on page 118.

For information about the binding files for CMS application multitasking routines, see *z/VM: CMS Application Multitasking*. For information about the binding files for the WorkstationGetAddress routine and VMLIB routines, see *z/VM: CMS Callable Services Reference*.

3. Only character string and signed binary data can be passed to a CSL routine. If a CSL parameter is defined as a signed binary number, the REXX CSL interface makes the necessary translations to and from the CSL routine. However, the CSL interface cannot translate a number in exponential notation to signed binary. Use the NUMERIC DIGITS instruction to ensure that numbers used in your program will not be represented in exponential form.

4. You cannot call CSL routines asynchronously from a REXX program.

5. Some CSL routine descriptions state that the previous values of the output parameters are not changed for certain input combinations. This is true when the calls are made from languages other than REXX. When a call to a CSL routine is made, REXX creates a buffer for each output parameter and initializes the buffer to blanks. After completing the CSL routine call, REXX saves each buffer value into the corresponding REXX variable. Therefore, if the CSL routine does not change one or more output parameter values, the corresponding REXX variable values will be set to blanks when the call is completed.

Returned Values

The list below shows the possible returned values from the CSL interface of REXX. These are in addition to any values returned from the specific CSL routine being called. If CSL was called as a function, then the returned values replace the function call in the expression evaluation, the same as any function call. If CSL was called as a subroutine, then the variable RESULT has the returned value.

0
    Routine was run and control returned to the REXX exec

-7
    Routine was not loaded from a callable services library

-8
    Routine was dropped from a callable services library

-9
    Insufficient storage was available (See Chapter 1, "REXX General Concepts," on page 1.)

-10
    More parameters than allowed were specified

-11
    Fewer parameters than required were specified

-20
    Invalid call

-22
    Invalid REXX argument

-23
    Subpool create failure

-24
    REXX fetch failure

-25
    REXX set failure

-26*nnn*
    Incorrect data length for parameter number *nnn*. Possible reasons are:

    • The passed length parameter was greater than the maximum allowed for the parameter

- A length value greater than 65535 was supplied for a variable-length character or bit string
- The length specified for an output variable-length character or bit string parameter is greater than the size the variable was initialized to before the call
- A binary or length input variable is too big.

-27*nnn*

Incorrect data or data type for parameter number *nnn*. Possible reasons are:

- A binary value was passed that contained a non-numeric character or had an initial character that was not numeric, +, -, or ' '
- A parameter defined as unsigned binary (with a length of 4, 3, 2, or 1) was supplied a negative value
- A value was supplied for a bit string parameter that contained characters other than 0 or 1
- At least one of the stemmed variables containing values for an input column was not defined.

-28*nnn*

Incorrect variable name for parameter number *nnn*. Possible reasons are:

- An incorrect variable name was specified for an output variable
- A quoted literal value was supplied as the name for an output variable
- A quoted literal value was supplied as the name for a table column stemmed variable name
- No second quote character was passed for a quoted literal
- The second quote character for a quoted literal was followed by a character that was not a blank.

-29*nnn*

Incorrect length value (for example, a negative value) was specified for length parameter, parameter number *nnn*.

Note: For the last four return codes, parameters are numbered serially, corresponding to the order in which they are coded. The *rtnname* is always parameter number 001, *retcode* is always parameter number 002, the next parameter is 003, and so forth.

Examples

The following example program section shows the CSL function of REXX calling a routine DMSEXIFI to check if a given shared file exists.

```
/*    Portion of Example REXX Program that Uses CSL function   */

fileid = 'SAMPLE FILE .subdir1.subdir2'
f_len = length(fileid)
commit = 'COMMIT'
c_len = length(commit)
CALL csl 'DMSEXIFI rtnc rsnc fileid f_len commit c_len'

if rtnc = 0              /* RESULT and rtnc are always the same */
  then say 'File Exists'
  else Do
        Say 'File does not exist as specified.'
        Say 'Return code is ' rtnc
        Say 'Reason code is ' rsnc
      End

Exit rtnc

/* --- End of Example --- */
```

In the preceding example

```
fileid = 'SAMPLE FILE .subdir1.subdir2'
```

identifies the directory where SAMPLE FILE is located. Using a period as the first character indicates that the user's top directory in the default file pool is searched.

To search for SAMPLE FILE located on another user's directory, specify the user ID before the first period. For example, to see if SAMPLE FILE exists in a directory SMITH owns, use the following statement:

```
fileid = 'SAMPLE FILE SMITH.subdir1.subdir2'
```

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

DIAG
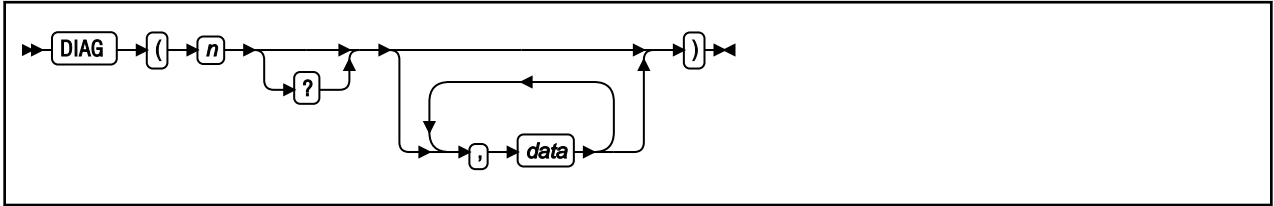
communicates with CP through a dummy DIAGNOSE instruction and returns data as a character string. (This interface is described in the discussion on the DIAGNOSE Instruction in *z/VM: CP Programming Services*.)

The *n* is the hexadecimal diagnose code to be run. You can omit leading zeros. The ? indicates that diagnostic messages are to be displayed if appropriate. The optional item, *data*, is dependent upon the specific diagnose code being run; it is generally the input data for the given diagnose.

Note: A DIAGNOSE function with incorrect parameters may in some cases result in a specification exception or a protection exception.

The data returned is in binary format; that is, it is precisely the data that the DIAGNOSE returns; no conversion is performed.

Note: The REXX built-in functions C2X and C2D can convert the returned data. Samples of the use of these functions are included in the descriptions of DIAGRC codes 0C and 60.

Codes are the same as for DIAGRC.

DIAGRC
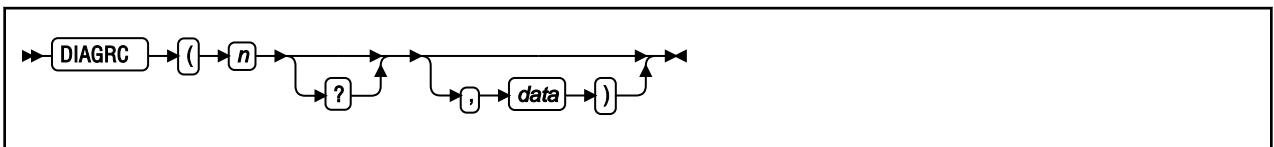
is similar to the DIAG function. The *n* is the hexadecimal diagnose code to be run. You can omit leading zeros. The use of quotation marks is optional but recommended. This is especially true for DIAGNOSE codes C, C8, and CC. The ? indicates that diagnostic messages are to be displayed if appropriate. The optional item, *data*, is dependent upon the specific diagnose code being run; it is generally the input data for the given diagnose.

In contrast to the DIAG function, the data returned in this function has a prefix of:

Character position	Contents
1 to 9	Return code from CP
10	A blank
11	Condition code from CP

Character position	Contents
12 to 16	Five blanks

The input and the returned data for each supported diagnose follow. If a diagnose code is not in this list, it is not supported.

DIAG(00)
   — Store Extended-Identification Code

DIAGRC(00)
   The value returned is a string, at least 40 characters in length, depending on the level of nesting of z/VM. The maximum amount of data returned is 200 bytes, but ordinarily 40 bytes of data are returned.

DIAG(08,*cpcommand*[,*sizebuf*])
   — Virtual Console Function

DIAGRC(08,*cpcommand*[,*sizebuf*])

Input is *cpcommand* (CP command) to be issued (240 bytes maximum), followed (optionally) by a third argument, *sizebuf*, which specifies the size (in bytes) of the buffer that will contain the result. This buffer size must be a nonnegative whole number; the default is 4096. When *sizebuf* is 0 then the command is not run.

Any command response is returned as the function value. If the response contains multiple lines, the character '15'X delimits them in the returned data.

Note that the command is passed to CP without any translation to uppercase. Thus commands specified as a quoted string (a good idea) must be in uppercase or CP does not recognize them. For example:

```
Diag(8,'query rdr all')  /* fails because CP has no   */
                         /* "query" command (only     */
                         /* "QUERY").                  */

Diag(8,query rdr all)    /* ordinarily works, but will*/
                         /* fail if "query", "rdr" or */
                         /* "all" are variables that  */
                         /* have been assigned values */
                         /* other than their own names*/

Diag(8,'QUERY RDR ALL')  /* is the best and safest.   */
```

DIAG(0C)
   — Pseudo Timer

DIAGRC(0C)
   The value returned is a 32-byte string containing the date, time, virtual time used, and total time used.

   For example, to display the virtual time:

```
Say 'Virtual time =' c2x(substr(diag('C'),17,8)) '(Hex)'

/* This results in a display of the form */

Virtual time = 00000000004BF959 (Hex)
```

   You can display the virtual time as a decimal value by using the C2D function:

```
Say 'Virtual time =' c2d(substr(diag('C'),17,8))

/* This results in a display of the form */

Virtual time = 4979033
```

DIAG(14,*acronym*,*rdrvaddr*,*addvals*)
   — Input File Spool Manipulation

DIAGRC(14,*acronym*,*rdrvaddr*,*addvals*)

The *acronym* is one of those described following. The *rdrvaddr* is the address of the virtual reader. The *addvals* are one or more additional and sometimes optional values associated with a given *acronym*. Acronym descriptions (following) include any additional, associated values as well.

The value returned is:

Character position	Contents
1	Condition code
2	A blank
3 to 6	Four bytes from register y+1
7 to 8	Two blanks
9 onwards	A return string (if any) whose length and content depend upon the function being performed.

Note: The PARSE instruction can assign these operands to suitable variables, as in the examples shown.

Acronym Descriptions:

The hexadecimal diagnose codes are in parentheses after the acronym name.

RNSB,*rdrvaddr*
— Read Next Spool Buffer; data record (X'0000')

There are no additional values associated with this acronym.

The return string is the 4096-byte spool file buffer. For example,

```
Parse value diag(14,'RNSB','00C'),
        with cc 2 . 3 Ryp1 7 . 9 buffer

/* will read the next spool buffer from the   */
/* card reader at address X'00C' and assign:  */
/*    CC = the condition code                 */
/*    RYP1 = the contents of register y+1     */
/*    BUFFER = the 4096-byte spool buffer     */
```

RNPRSFB,*rdrvaddr*[,*readnum*]
— Read Next PRint Spool File Block (X'0004')

The *readnum* may be used to specify the number of doublewords of the spool file block to be read. (See Note .)

The return string is the next spool file block of type PRT. Thus to read the next spool file block of type PRT from device X'00C':

```
Parse value diag(14,'RNPRSFB','00C',18),
        with cc 2 . 3 Ryp1 7 . 9 SFB

/* will read the next print spool file block from */
/* the card reader at address X'00C' and assign:  */
/*    CC = the condition code                     */
/*    RYP1 = the contents of register y+1         */
/*    SFB = the 144-byte spool file block         */
```

RNPUSFB,*rdrvaddr*[,*readnum*]
— Read Next PUnch Spool File Block (X'0008')

The *readnum* may be used to specify the number of doublewords of the spool file block to be read. (See Note .)

The return string is the next spool file block of type PUN.

Thus to read the next spool file block of type PUN from device X'00C':

```
Parse value diag(14,'RNPUSFB','00C',18),
          with cc 2 . 3 Ryp1 7 . 9 SFB

/* will read the next punch spool file block from */
/* the card reader at address X'00C' and assign:  */
/*    CC = the condition code                      */
/*    RYP1 = the contents of register y+1          */
/*    SFB = the 144-byte spool file block          */
```

SF,*rdrvaddr*,*spfileid*
— Select a File for processing (X'000C')

The *spfileid* specifies the spool file id.

There is no return string other than the condition code and Ry+1 value.

Thus to select spool file number 8159 for processing from device X'00C':

```
Parse value diag(14,'SF','00C',8159),
          with cc 2 . 3 Ryp1 7

/* will select a file for processing from the  */
/* card reader at address X'00C' and assign:   */
/*    CC = the condition code                   */
/*    RYP1 = the contents of register y+1       */
```

RPF,*rdrvaddr*,*newcopy*
— RePeat active File *nnn* times (X'0010')

The *newcopy* specifies the new copy count, the maximum being 255. If the count is greater than 255, it is set to 255.

There is no return string other than the condition code and Ry+1 value.

Thus to change the copy count for the active file on device X'00C' to 5:

```
Parse value diag(14,'RPF','00C',5),
          with cc 2 . 3 Ryp1 7

/* will repeat active file 5 times on the     */
/* card reader at address X'00C' and assign:  */
/*    CC = the condition code                  */
/*    RYP1 = the contents of register y+1      */
```

RSF,*rdrvaddr*
— ReStart active File at beginning (X'0014')

There are no additional values associated with this acronym.

The return string is the first 4096-byte spool file buffer.

Thus to reset the active file on device X'00C' to the beginning and read the first spool buffer:

```
Parse value diag(14,'RSF','00C'),
          with cc 2 . 3 Ryp1 7 . 9 buffer
```

BS,*rdrvaddr*
— BackSpace one record (X'0018')

There are no additional values associated with this acronym.

The return string is the 4096-byte spool file buffer.

Thus to read the previous spool buffer from the file active on device X'00C':

```
Parse value diag(14,'BS','00C'),
          with cc 2 . 3 Ryp1 7 . 9 buffer

/* will read the previous spool file buffer from */
/* the card reader at address X'00C' and assign: */
```

```
/*    CC = the condition code                */
/*    RYP1 = the contents of register y+1     */
/*    BUFFER = the first 4096 bytes of the file  */
```

RLSFB,*rdrvaddr*
— Read Last Spool File Buffer (X'0024')

There are no additional values associated with this subcode.

The specified device is checked for an already active file. The return string is the last 4096-byte spool file buffer if there is an active file. If there is no active file, cc=2 is set.

Thus to read the last spool file buffer from the card reader at address X'00C':

```
Parse value diag(14,'RLSFB','00C'),
        with cc 2 . 3 Ryp1 7 . 9 buffer

/* will read the last full page buffer from the card */
/* the card reader at address X'00C' and assign:     */
/*    CC = the condition code                        */
/*    RYP1 = the contents of register y+1            */
/*    buffer = the last 4096 byte spool buffer        */
```

SFPRNSB,*rdrvaddr,spfileid*
— Select a File for Processing and Read Next Spool Buffer (X'002C')

This subcode is a combination of an SF subcode followed by an RNSB.

The return string is a 4096-byte spool file buffer. Held files are skipped.

Thus to select and read the next spool file buffer from the card reader at address X'00C' for spool file 8159:

```
Parse value diag(14,'SFPRNSB','00C', 8159),
        with cc 2 . 3 Ryp1 7 . 9 buffer

/* will select and read the next 4096 page buffer */
/* from the card reader X'00C' buffer for spool   */
/* ID 8159.                                        */
/*    CC = the condition code                      */
/*    RYP1 = the contents of register y+1          */
/*    BUFFER = the next 4096-byte spool buffer     */
```

RSFDNPR,*n*[,*numwords*[,3800]]
— Retrieve Subsequent File Descriptor Not Previously Retrieved (X'0FFE')

The *n* is either 0 (to retrieve subsequent file descriptor not previously retrieved) or 1 (to reset the previously retrieved flags for all the file descriptors; then retrieve the first file descriptor). The optional *numwords* specifies the number of doublewords of spool file block data to be returned. (See Note "3" on page 129.) You can specify 3800, which is also optional, to cause 40 bytes of 3800 information to be included between the spool file block and the tag.

See Notes "1" on page 129 and "2" on page 129 for additional information.

Thus to obtain information about the next not previously retrieved file without regard to type, class, and so forth:

```
  Parse value diag(14,'RSFDNPR',0,18),
          with cc 2 . 3 Ryp1 7 . 9 SFB 153 . 205 tag

/* will read the spool file block           */
/* from the card reader at address X'00C' and */
/* assign:                                    */
/*    CC = the condition code                 */
/*    RYP1 = the contents of register y+1     */
/*    SFB = the 144-byte spool file block     */
/*    TAG = the tag data                      */
```

RSFD,*spfilenum*[,*numwords*[,3800]]
— Retrieve Subsequent File Descriptor (X'0FFF')

The *spfilenum* specifies the spool file number at which the search for the next file begins; it may be 0 to start at the beginning of the queue. The optional *numwords* specifies the number of doublewords of spool file block data to be returned. (See Note "3" on page 129.) 3800, also optional, may be specified to cause 40 bytes of 3800 information to be included between the spool file block and tag.

See Notes "1" on page 129 and "2" on page 129 for additional information.

Thus to obtain information about the next spool file without regard to type, class, and so forth:

```
Parse value diag(14,'RSFD',0,18,3800),
        with cc 2 . 3 Ryp1 7 . 9 SFB,
        153 data_3800 193 . 205 tag

/* will read the spool file block        */
/* from the card reader at address X'00C' and */
/* assign:                               */
/*    CC = the condition code            */
/*    RYP1 = the contents of register y+1 */
/*    SFB = the 144-byte spool file block */
/*    DATA_3800 = the 3800 data          */
/*    TAG = the tag data                 */
```

Notes on Diagnose X'14'

1. Because only one bit is provided to indicate that the length of return data is being explicitly stated and that 3800 data is being requested, if either is specified (on RSFD or RSFDNPR calls), 40 bytes of 3800 data are returned.

2. RSFD and RSFDNPR wait for a file being used by a system function. If, however, the file does not become available in the 250 millisecond time limit, the function returns a null string for DIAG, usual return code information for DIAGRC. For a discussion of possible causes for this condition, see the notes on "DIAGNOSE Code X'14'" in *z/VM: CP Programming Services*.

3. For RNPRSFB, RNPUSFB, RSFD, and RSFDNPR, the default number of doublewords of spool file block is 13; however, the length of the spool file in the current release of z/VM can be found in *z/VM: CP Programming Services*. The maximum number of doublewords that can be returned is 63; if a larger number is specified, only 504 bytes will be returned. The preceding examples do not necessarily return the entire spool file block. If *readnum* or *numwords* is changed in the preceding examples, the template on the parse instruction must also be changed.

DIAG(24,*devaddr*)
— Device Type and Features

DIAGRC(24,*devaddr*)
The input, *devaddr*, is the device address or number (or -1 for virtual console).

Note: DIAGNOSE X'24' is not valid for a 3390 DASD. Use DIAGNOSE X'210' instead.

The value returned is a 13-byte string of virtual and real device information:

Position	Contents
1 through 4	Virtual device information from Register y
5 through 8	Real device information from Register y+1
9 through 12	If -1 was specified, virtual console information from Register x
13	Condition code

DIAG(5C,*editstring*[,*headerlen*])
— Error Message Editing

DIAGRC(5C,*editstring*[,*headerlen*])
The *editstring* is a string to be edited according to the current EMSG setting. The *headerlen* is the length of the header used for the editing. If you do not specify *headerlen*, the default length is 10. The *headerlen* may not be longer than *editstring*.

The value returned is the edited message, which is a null string, the message code, the message text, or the entire input string, depending on the EMSG setting.

DIAG(60)
— Determine Virtual Storage Size

DIAGRC(60)
The value returned is the 4-byte virtual storage size.

You can display this value in hexadecimal with:

```
Say 'Virtual storage =' c2x(Diag(60))
```

resulting (for example) in display of the line:

```
Virtual storage = 00100000
```

Alternatively, you can display storage size in terms of K.However, depending on the virtual storage size, numeric digits may need to be set greater than the default of 9.

```
Say 'Virtual storage =' c2d(diag(60))/1024'K'
```

resulting (for example) in display of the line:

```
Virtual storage = 512K
```

You can compare the virtual storage size to a hexadecimal value:

```
Say diag(60) > '00100000'x
```

results in display of 1 for virtual machines greater than 1M in size and 0 for those 1M or less. You can express the same comparison in terms of megabytes:

```
Say c2d(diag(60))/(1024*1024) > 1
```

with the same results.

DIAG(64,*subcode*,*name*[,*subfunction*])
— Named Saved Segment Manipulation

DIAGRC(64,*subcode*,*name*[,*subfunction*])
The input *subcode* is a 1-character code indicating the function to perform, followed by a third argument, *name*, the name of the segment. Specify a fourth argument, *subfunction*, only with the **X** *subcode*. The *subfunction* is one of the following: FINDSPACE, FINDSKEL, or FINDSEGA.

For *subcodes* **F**, **L**, **O**, **P**, and **S**, the value returned is a 9-byte string consisting of the returned Rx and Ry values, and a single byte condition code. For *subcode* **X**, the value returned is a 1064-byte string. (The number of bytes is a decimal number.)

Note: For **X**, the returned buffer includes output *and input* buffers. The returned buffer contains: 4 bytes for Rx; 4 bytes for Ry; 1 byte for the condition code; 7 bytes reserved; and the input to the subcode, followed by the output.

The subcodes are:

F
Find starting and ending address of the named saved segment.

L
Load a saved segment in nonshared mode.

O
Load a saved segment in shared mode only if no overlay condition exists.

P
Release a saved segment from storage.

S

Load a saved segment in shared mode.

X

Call the *subfunction* FINDSPACE, FINDSKEL, or FINDSEGA.

For example, to find the load address of the segment SPFSEG and display the starting and ending addresses and the condition code:

```
spfsegaddr=diag(64,'F','SPFSEG')
Say 'Start:' c2x(substr(spfsegaddr,1,4)),
    ' End:' c2x(substr(spfsegaddr,5,4)),
    ' CC:' substr(spfsegaddr,9,1)

/* which displays:
            Start: 00230000   End: 0024FFFF   CC: 0  */
```

indicating that the segment loads from 230000 to 24FFFF, and is already loaded (cc=0).

Attention: The L and S functions should be used with care. It is the coder's responsibility to ensure that the loaded segment will not overlap virtual storage (see DIAG 60 preceding). CP will load a segment in the middle of your virtual storage if requested, so code carefully.

Note: You can use the CMS SEGMENT command instead of this function to load and purge a named segment. (See *z/VM: CMS Commands and Utilities Reference* for a description of the SEGMENT command.)

However, SEGMENT PURGE is not interchangeable with DIAG(64,P) or DIAGRC(64,P). This means you must purge a segment the same way you loaded it; for example, if you use DIAG(64,S,SEGONE) to load segment SEGONE, use DIAGRC(64,P,SEGONE) to purge segment SEGONE.

DIAG(8C)

— Access 3270 Display Device Information

DIAGRC(8C)

The value returned is a string that is variable in length. The string contains device-dependent information about the device (the virtual console). If the device is LAN attached to the host, then additional data may be appended by other LAN-attached devices. If the virtual machine is disconnected or the virtual console is a TTY device, then the returned string is null. You can determine the length of the returned string using the LENGTH(*string*) function.

The value returned is:

Byte	Contents
0	Flags:
	X'01' 14-bit addressing is available **X'20'** programmed symbol sets are available **X'40'** device has extended highlighting **X'80'** device has extended color
1	Number of partitions
2-3	Number of columns on the terminal
4-5	Number of rows on the terminal
6-n	Information returned to CP by the initial Write Structured Field Query Reply

Note: If the evaluation of a clause containing DIAGRC(8C) has as its target a disconnected device or a TTY type device, a NULL string will be returned with a return code of '0' and a condition code of '2'. This is instead of having a return code of '2' set in a register and a condition code of '0', as would be set if submitted directly to CP. Since the DIAG(8C) and DIAGRC(8C) are intended to obtain information about 3270-type devices, when we evaluate DIAG(8C) and DIAGRC(8C) within the REXX code we first issue a DIAG(24) to determine the console type. If the device is disconnected, we receive a cc=2, Rx +1 is not updated by CP, and we skip issuing the DIAG(8C) and DIAGRC(8C). We then preserve the condition code and return a NULL string.

DIAG(BC,*subcode,device*)
— Open and Query Spool File Characteristics

DIAGRC(BC,*subcode,device*)
The *subcode* is either CHAR or BIN. CHAR meaning return information in character format only. BIN meaning return information in character format where appropriate or binary format for numeric information.

The value returned is:

Character position	Contents
1	Condition code
2	A blank
3 to 6	Four bytes from register y+1
7 to 8	Two blanks
9 onwards	The return string from CP for DIAGNOSE Code X'BC'. See *z/VM: CP Programming Services* for details on the returned information. The REXX subcode CHAR corresponds to the CP X'00' subcode. The REXX subcode BIN corresponds to the CP X'04' subcode.

DIAG('C8',*langid*)
— SET CP's language

DIAGRC('C8',*langid*)
The value returned is a 5-byte string containing the *langid* that CP set.

If this DIAGNOSE code is issued from an exec and CMS is on an earlier version of CP, error message DMSREX475E (`Incorrect call to routine`) is issued and the exec terminates.

DIAG('CC',*langid,addr*)
— SAVE CP's language repository at address *addr*

DIAGRC('CC',*langid,addr*)
The value returned for the DIAG function is a null string. The *addr* must be on a page boundary.

If this DIAGNOSE code is issued from an exec and CMS is on an earlier version of CP, error message DMSREX475E (`Incorrect call to routine`) is issued and the exec terminates.

Message DMSREX475E also results if an unauthorized user ID tries to enter DIAGNOSE code X'CC'. Return code 20040 is set.

DIAG('F8',*spoolid*)
— Read Spool File Origin Data

DIAGRC('F8',*spoolid*)
The four-character *spoolid* identifies the spool file in question.

The value returned is a character string consisting of:

- A 2-byte version code (binary 0)
- A 2-byte spool ID (in binary)

- An 8-byte node ID
- An 8-byte user ID.

For example, assume the spool ID is 4321, the associated node ID is NODE7, and the user ID is SMITH. Then the statements:

```
ANS=DIAG('F8',4321)
say 'VERSION: ' C2D(SUBSTR(ANS,1,2))
say 'SPOOLID: ' C2D(SUBSTR(ANS,3,2))
say 'NODEID:  ' SUBSTR(ANS,5,8)
say 'USERID:  ' SUBSTR(ANS,13,8)
```

Would display:

```
VERSION:  0
SPOOLID:  4321
NODEID:   NODE7
USERID:   SMITH
```

For another example, assume the return code is 000000000, the condition code is 1, the spool ID is 4321, the associated node ID is NODE7, and the user ID is SMITH. Then the statements:

```
ANS=DIAGRC('F8',4321)
say 'RC:      ' STRIP(SUBSTR(ANS,1,9))
say 'CC:      ' SUBSTR(ANS,11,1)
say 'VERSION: ' C2D(SUBSTR(ANS,17,2))
say 'SPOOLID: ' C2D(SUBSTR(ANS,19,2))
say 'NODEID:  ' SUBSTR(ANS,21,8)
say 'USERID:  ' SUBSTR(ANS,29,8)
```

Would display:

```
RC:       0
CC:       1
VERSION:  0
SPOOLID:  4321
NODEID:   NODE7
USERID:   SMITH
```

If your system supports DIAGNOSE X'F8', the system obtains the spool file's origin data. If your system does not support DIAGNOSE X'F8', any such request terminates and results in error message DMSREX475E (Incorrect call to routine).

DIAG(210,*devaddr*)
— Retrieve Device Information

DIAGRC(210,*devaddr*)
The input, *devaddr*, is the device address or number. Although DIAGNOSE X'24' is valid for many device types, if your device address is a 3390 you should use this DIAGNOSE X'210' rather than DIAGNOSE X'24'.

The value returned is as follows:

Position	Contents
1 through 4	Virtual device information from word 1 of the VRDCB
5 through 8	Real device information from word 2 of the VRDCB
9 through 12	Device number or device address
13	Condition code (not applicable for other than DASD that supports RDC data)

DIAG(218)
— Retrieve Real CPU ID Information

DIAGRC(218)
The value returned is an 8-byte string containing the CPU ID of the processor where the diagnose run.

For example, to display the real CPU ID from a processor:

```
SAY 'Real CPU ID  =' c2x(diag(218))
```

Which displays:

```
Real CPU ID = 6512032890210000
```

Here is another way to look at it:
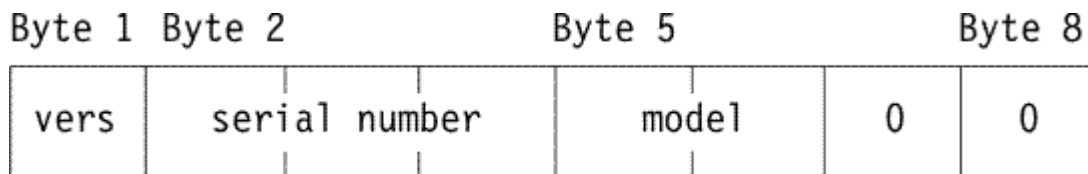
DIAG(270)
— Pseudo Timer Extended

DIAGRC(270)
The value returned is a string that is variable in length. The string contains identical information that would be obtained from the DIAGNOSE X'0C' (the date, time, virtual time used, and total time used) plus the date in the FULLDATE format followed by the ISODATE format.

The value returned is as follows:

Position	Contents
1 through 8	The date (MM/DD/YY)
9 through 16	The time of day (HH:MM:SS)
17 through 24	The virtual CPU time used
25 through 32	The total CPU time used
33 through 42	The date in FULLDATE format (MM/DD/YYYY)
43 through 48	6 X'00's (reserved)
49 through 58	The date in ISODATE format (YYYY-MM-DD)
59	X'01'
60	The user's date format
61	The system's date format
62 through 64	3 X'00's (reserved)

SOCKET
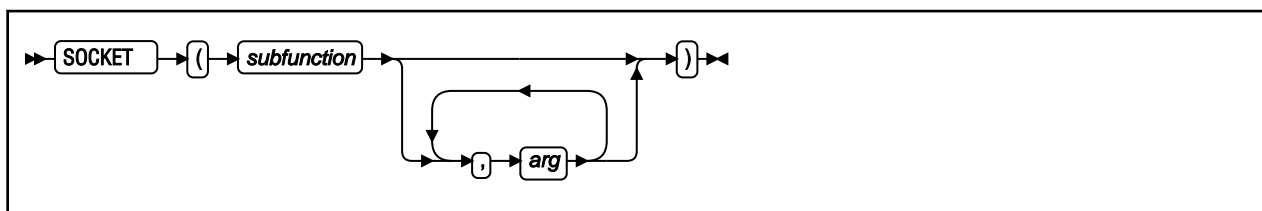
provides access to the TCP/IP socket interface. This allows you to use REXX to implement and test TCP/IP applications.

subfunction is the name of a REXX socket function and *arg* is a parameter on that socket function.

REXX socket functions are provided to:

- Process socket sets
- Initialize, change, and close sockets
- Exchange data
- Resolve names for sockets
- Manage configurations, options, and modes for sockets
- Translate data and do tracing

See "Tasks You Can Perform Using REXX Sockets" on page 221 and "REXX Socket Functions" on page 223.

A character string is returned from the SOCKET call 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.

Here are some examples:

```
SOCKET('GetHostId')        ->    '0 9.4.3.2'
SOCKET('Recv',socket)      ->    '35 EWOULDBLOCK Operation would block'
```

During initialization of the REXX Sockets module or when doing certain REXX socket functions, system messages may also be issued.

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

STORAGE
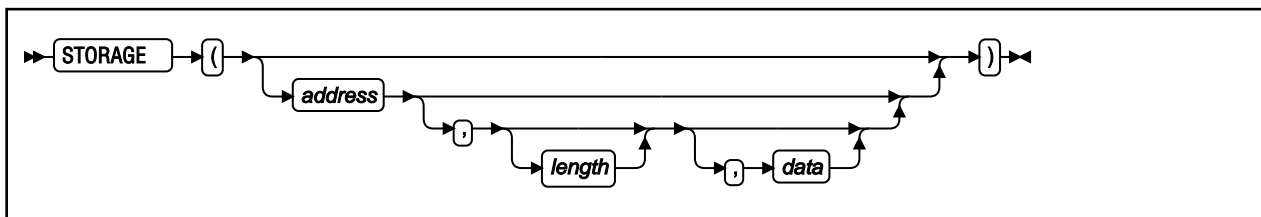
returns the current virtual machine size expressed as a hexadecimal string if no arguments are specified. Otherwise, returns *length* bytes from the user's memory starting at *address*. The *length* is in decimal; the default is 1 byte. The maximum *length* is 16 MB minus the length of the control information associated with an EVALBLOK. The *address* is a hexadecimal number. In an ESA/390 or ESA/XC virtual machine, the high-order bit of *address* is ignored. In an ESA or XA virtual machine that has been switched into z/Architecture mode, *address* can be 16 hexadecimal digits.

If you specify *data,* after the current value has been retrieved storage starting at *address* is overwritten with *data* (the *length* argument has no effect on this).

If *length* would imply returning storage beyond the virtual machine size, only those bytes up to the virtual machine size are returned. If an attempt is made to alter any bytes outside the virtual machine size, they are left unaltered.

⚠️ **Attention:** The STORAGE function allows any location in your virtual machine to be altered. Do not use this function without due care and knowledge.

Examples:

```
/* After DEF STOR 20M     */
STORAGE()       ->  '1400000'
```

```
/* Note that the following results vary from system to system.  */
STORAGE(200,32) ->   'z/VM V6.1.0  10/19/09 12:58'
```

STSI

```
>>--[ STSI ]--[ ( ]--------------------[ 0 ]----------------------[ ) ]--><
                    |                                          |
                    '--[ function code ]--[ , ]--[ selector 1 ]--[ , ]--[ selector 2 ]--'
```
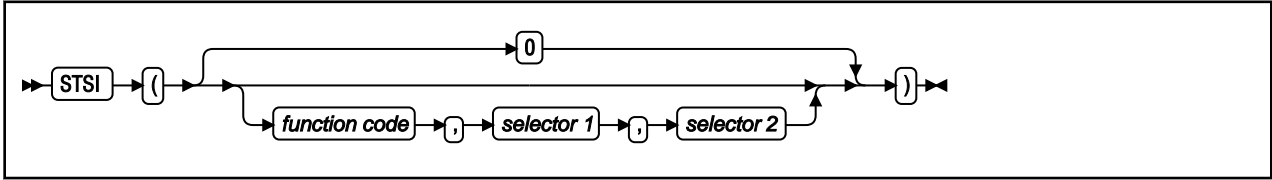
If no arguments or a value of zero are specified, then the current configuration level is returned as the first four bytes of a 4K byte hexadecimal string. If a function code and two selector codes are specified, then information related to the component or components of the configuration is returned as a 4K byte hexadecimal string. Refer to the z/Architecture Principles of Operation (publibfi.boulder.ibm.com/epubs/pdf/dz9zr011.pdf) for descriptions of the valid function and selector codes and for information on the contents of the data returned.

The syntax of the STSIUSE SAMPEXEC is:

```
>>--[ STSIUSE ]--><
```
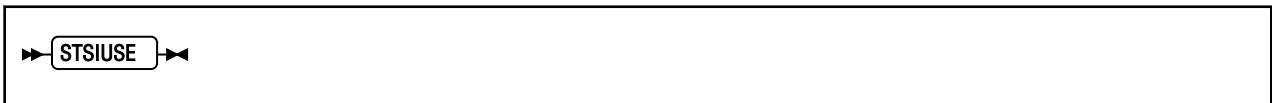
STSIUSE SAMPEXEC is delivered as a sample exec. Its purpose is to demonstrate possible parsings of the data returned by various calls to the CMS REXX STSI function. Some sample output may appear as:

```
stsiuse

STSI(0,0,0).........................................0.0.0.
Current-config-level number: 30000000

STSI(1,1,1).........................................1.1.1.
Manufacturer:             IBM
Type:                     2094
Model-Capacity Identifier:   738
Sequence Code:            0000000000029B9E
Plant of Manufacture:     02
Model:                    S38

STSI(1,2,1).........................................1.2.1.
Sequence Code:            0000000000029B9E
Plant of Manufacture:     02
CPU Address:              0000
```

Chapter 4. Parsing

The parsing instructions are ARG, PARSE, and PULL (see <u>"ARG" on page 27</u>, <u>"PARSE" on page 48</u>, and <u>"PULL" on page 53</u>).

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 <u>"Templates Containing String Patterns" on page 139</u> for details.)

Positional patterns
Indicate the character positions at which to split the source string. (See <u>"Templates Containing Positional (Numeric) Patterns" on page 140</u> 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

Parsing Value Instruction Example:

```
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 <u>"Using UPPER" on page 144</u> 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. For more information about parsing instructions, see <u>"Parsing Instructions Summary" on page 145</u>.

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 143 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 ("Combining String and Positional Patterns: A Special Case" on page 147) 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 143 for details.)

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.

```
character positions
     1            11            21                40
FIELDS:  | LASTNAME | FIRST | PSEUDONYM           |  end of
                                                      record
```
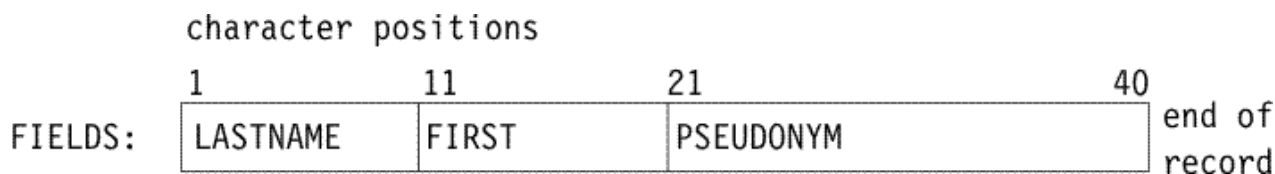
```
/* Parsing with absolute positional patterns in template        */
record.1='Clemens    Samuel    Mark Twain         '
record.2='Evans      Mary Ann  George Eliot       '
record.3='Munro      H.H.      Saki               '
do n=1 to 3
   parse var record.n lastname 11 firstname 21 pseudonym
   If 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 143.)

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 ("Combining String and Positional Patterns: A Special Case" on page 147) when a string pattern precedes a variable name and a positional pattern follows the variable name). The templates from the examples of absolute and relative positional patterns give the same results.

```
┌─────────┐   ┌─────────────────┐   ┌─────────────────┐   ┌─────────────────┐
│         │   │ lastname   11   │   │ firstname   21  │   │ pseudonym       │
│         │   │ lastname  +10   │   │ firstname  +10  │   │ pseudonym       │
└─────────┘   └─────────────────┘   └─────────────────┘   └─────────────────┘

(Implied      Put characters      Put characters      Put characters
starting      1 through 10        11 through 20       21 through end
point is      in lastname.        in firstname.       of string in
position      (Non-inclusive      (Non-inclusive      pseudonym.
1.)           stopping point      stopping point
              is 11 (1+10).)      is 21 (11+10).)
```

Only with positional patterns can a matching operation *back up* to an earlier position in the source string. Here is an example using absolute positional patterns:

```
/* Backing up to an earlier position (with absolute positional)  */
string='astronomers'
parse var string 2 var1 4 1 var2 2 4 var3 5 11 var4
say string 'study' var1||var2||var3||var4
/* Displays: "astronomers study stars"                    */
```

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

With relative positional patterns, a number preceded by a minus sign backs up to an earlier position. Here is the same example using relative positional patterns:

```
/* Backing up to an earlier position (with relative positional)  */
string='astronomers'
parse var string 2 var1 +2 -3 var2 +1 +2 var3 +1 +6 var4
say string 'study' var1||var2||var3||var4      /* same results  */
```

In the previous example, the relative positional pattern -3 backs up to the first character in the source string.

The templates in the last two examples are equivalent.

```
┌───────────┐   ┌───────────┐   ┌───────┐   ┌───────────┐   ┌───────────┐   ┌───────────┐
│  2        │   │ var1   4  │   │   1   │   │ var2   2  │   │ 4 var3  5 │   │ 11 var 4  │
│  2        │   │ var1 +2   │   │  -3   │   │ var2 +1   │   │ +2 var3 +1│   │ +6 var 4  │
└─────┬─────┘   └─────┬─────┘   └───┬───┘   └─────┬─────┘   └─────┬─────┘   └─────┬─────┘
      │               │             │             │               │               │
   Start          Non-         Go to 1.  Non-          Go to 4.        Go to 11.
   at 2.          inclusive    (4-3=1)   inclusive     (2+2=4).        (5+6=11).
                  stopping               stopping      Non-
                  point is 4             point is 2    inclusive
                  (2+2=4).               (1+1=2).      stopping
                                                       point is
                                                       5 (4+1=5).
```
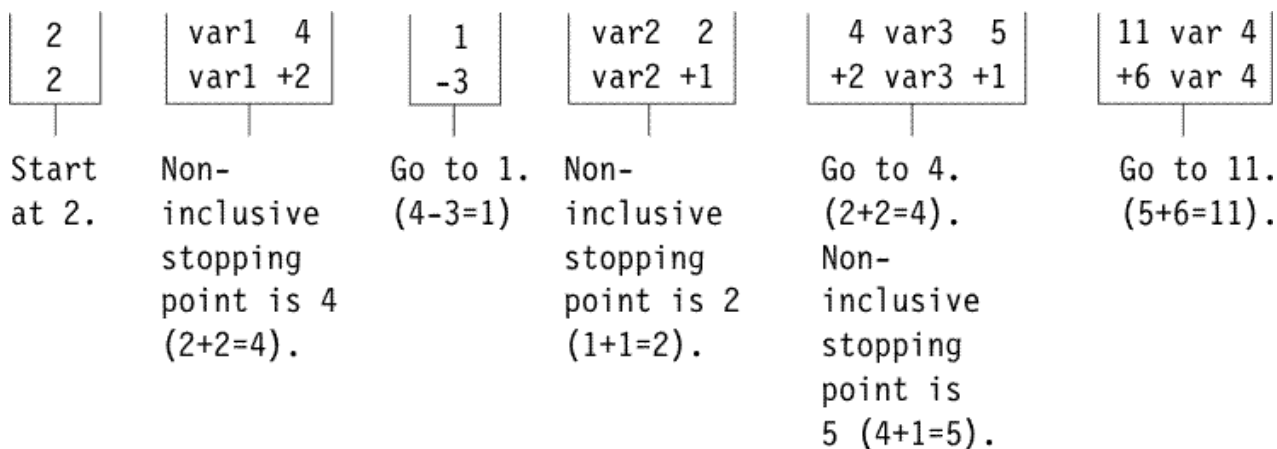
You can use templates with positional patterns to make multiple assignments:

```
/* 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.    */
```

When the value of a positional pattern evaluates to an integer less than 1 and the character position would be less than the string start, the effect is as if the value were equal to 1. That is, parsing cannot begin before the first character in a string.

When the value of a positional pattern evaluates to a position less than or equal to the previous character position, the parsed value is set to be from the previous character position to string end.

```
/* Combination absolute and relative positional patterns  */
string = 'abcdefghij'
parse var string 3 p1 -5 p2 +0 p3 6 p4 =2 p5
say p1 p2 p3 p4 p5
/* Displays:  "cdefghij abcdefghij abcde fghij bcdefghij" */
```

Here is how this template works:

```
┌───────────┐   ┌───────────┐   ┌───────────┐   ┌───────────┐   ┌───────────┐
│ 3 p1 -5   │   │ -5 p2 +0  │   │ -0 p3 6   │   │ 6 p4 =2   │   │ +2 p5     │
└─────┬─────┘   └─────┬─────┘   └─────┬─────┘   └─────┬─────┘   └─────┬─────┘
      │               │               │               │               │
```

Put	Starting at	Starting at	Starting at	Starting at
characters	1, put	1, put	6, put	2, put
3 through	characters	characters	characters	characters
string end	1 through	1 through	6 through	2 through
into p1.	string end	5 into p3.	string end	string end
(Stopping	into p2.	(Stopping	into p4.	into p5.
point is	(Stopping	point is	(Stopping	
string end	point is	the same	point is	
because -5	string end	as the	string end	
character	because +0	match	because =2	
position	character	start for	character	
is prior	position	p2.)	position	
to 3 which	is equal		is prior	
is match	to first		to match	
start.)	character		start.)	
	of string.)			

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 137.)

```
/* 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 137.)

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
PARSE UPPER LINEIN	PARSE LINEIN
PULL PARSE UPPER PULL	PARSE PULL
PARSE UPPER SOURCE	PARSE SOURCE

Converts alphabetic characters to uppercase before parsing	Maintains alphabetic characters in case entered
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	Next line from terminal input buffer
PARSE LINEIN	Next line in the default input stream.
PULL PARSE PULL	The string at the head of the external data queue. (If queue empty, uses default input, typically the terminal.)
PARSE SOURCE	System-supplied string giving information about the executing program.
PARSE VALUE	Expression between the keyword VALUE and the keyword WITH in the instruction.
PARSE VAR *name*	Parses the value of *name*.
PARSE VERSION	System-supplied string specifying the language, language level, and (three-word) date.

Parsing Instructions Examples

All examples in this section parse source strings into words.

ARG

```
/* ARG with source string named in REXX program invocation      */
/*  Program name is PALETTE.  Specify 2 primary colors (yellow,  */
/*   red, blue) on call.   Assume call is: palette red blue      */
arg var1 var2               /* Assigns: var1='RED'; var2='BLUE' */
If var1<>'RED' & var1<>'YELLOW' & var1<>'BLUE' then signal err
If var2<>'RED' & var2<>'YELLOW' & var2<>'BLUE' then signal err
total=length(var1)+length(var2)
SELECT;
  When total=7 then new='purple'
  When total=9 then new='orange'
  When total=10 then new='green'
  Otherwise new=var1                  /* entered duplicates */
END
Say new; exit                         /* Displays: "purple" */

Err:
say 'Input error--color is not "red" or "blue" or "yellow"'; exit
```

ARG converts alphabetic characters to uppercase before parsing. An example of ARG with the arguments in the CALL to a subroutine is in "Parsing Multiple Strings" on page 146.

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 LINEIN

```
parse linein 'a=' num1 'c=' num2    /* Assume: 8 and 9         */
sum=num1+num2                       /* Enter: a=8 b=9 as input */
say sum                             /*  Displays: "17"         */
```

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:      "CMS"    */
```

A PARSE VALUE example can be found in "Simple Templates for Parsing into Words" on page 137.

PARSE VAR examples are throughout the chapter, starting in "Simple Templates for Parsing into Words" on page 137.

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 139). But a template containing the sequence:

• string pattern

• variable name

• *relative* positional pattern

does *not* skip over the matching data. A relative positional pattern moves relative to the first character matching a string pattern. As a result, assignment includes the data in the source string that matches the string pattern.

```
/* Template containing string pattern, then variable name, then  */
/*  relative positional pattern does not skip over any data.     */
string='REstructured eXtended eXecutor'
parse var string var1 3 junk 'X' var2 +1 junk 'X' var3 +1 junk
say var1||var2||var3 /* Concatenates variables; displays: "REXX" */
```

Here is how this template works:

var1 3	junk 'X'	var2 +1	junk 'X'	var3 +1	junk
Put characters 1 through 2 in var1. (Stopping point is 3.)	Starting at 3, put characters up to (not including first 'X' in junk.	Starting with first 'X' put 1 (+1) character in var2.	Starting with char-acter after first 'X' put up to second 'X' in junk.	Starting with second 'X' put 1 (+1) character in var3.	Starting with char-acter after second 'X' put rest in junk.
var1='RE'	junk= 'structured e'	var2='X'	junk= 'tended e'	var3='X'	junk= 'ecutor'

Parsing with DBCS Characters

Parsing with DBCS characters generally follows the same rules as parsing with SBCS characters. Literal strings and symbols can contain DBCS characters, but numbers must be in SBCS characters. See "PARSE" on page 282 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.

The following figure shows a conceptual overview of parsing.

```
                    ┌──────────────────────────────────────────────┐
                    ▼                                               │
   ┌──────────────────────────────────────┐                        │
   │ START                                │                        │
   │ Token is first one in template.      │                        │
   │ Length=length(source string)         │                        │
   │ March start=1. Match end=1.          │                        │
   └──────────────────────────────────────┘                        │
                    │                                               │
   ┌─────────────►  ▼               yes                             │
   │   ┌──────────────────────┐  ───────►  ┌──────────────────────┐ │
   │   │ End of template?     │            │ Parsing complete.    │ │
   │   └──────────────────────┘            └──────────────────────┘ │
   │                    │ no                                        │
   │   ┌──────────────────────┐                                    │
   │   │ CALL Find Next       │                                    │
   │   │ Pattern.             │                                    │
   │   └──────────────────────┘                                    │
   │                    │                                           │
   │   ┌──────────────────────┐                                    │
   │   │ CALL Word Parsing.   │                                    │
   │   └──────────────────────┘                                    │
   │                    │                                           │
   │   ┌──────────────────────┐                                    │
   │   │ Step to next token.  │                                    │
   │   └──────────────────────┘                                    │
   │                    │                                           │
   │   ┌──────────────────────┐   yes    ┌──────────────────────┐  │
   │   │ Token a comma?       │ ──────►  │ Set next source      │──┘
   │   └──────────────────────┘          │ string and template. │
   │                    │ no             └──────────────────────┘
   └────────────────────┘
```
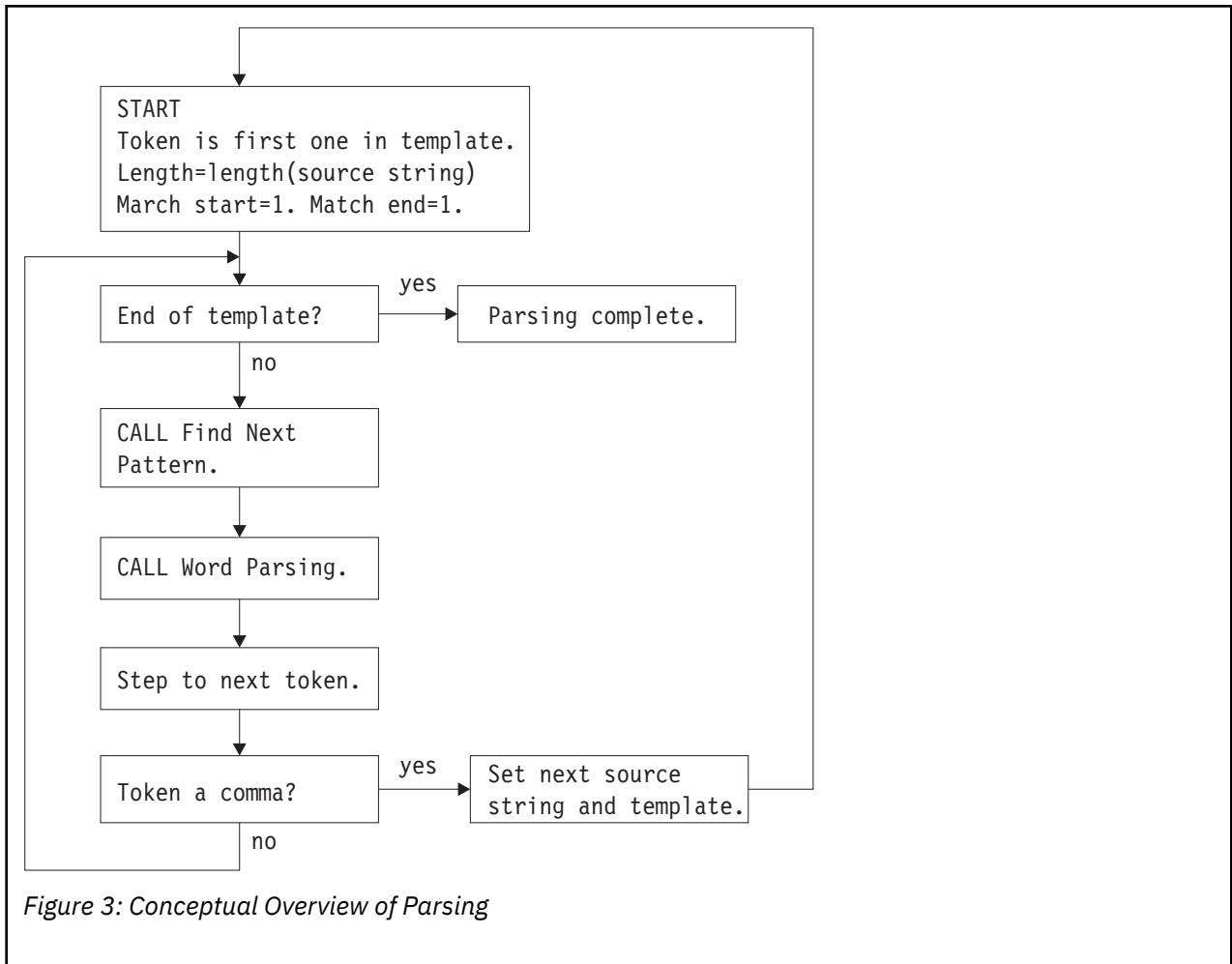
Figure 3: Conceptual Overview of Parsing

The next figure shows a conceptual view of finding the next pattern.
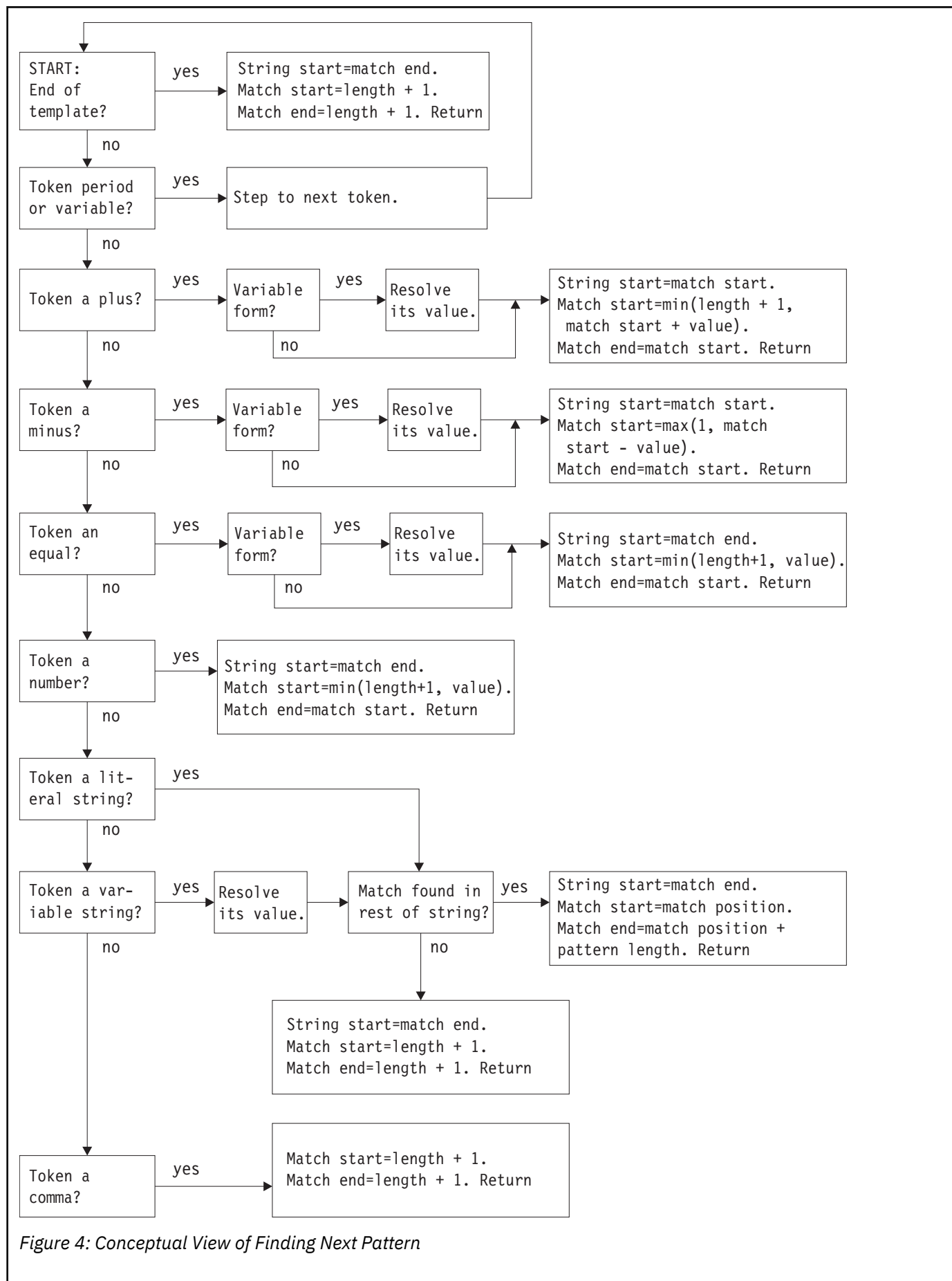
Figure 4: Conceptual View of Finding Next Pattern

The next figure shows a conceptual view of word parsing.

```
┌─────────────────────┐   no   ┌──────────────────────────┐
│ START:  Match end <= │──────→│ String end=match start.   │
│         string start?│        └──────────────────────────┘
└─────────────────────┘
         │ yes
         ▼
┌─────────────────────┐
│ String end=length + 1.│
└─────────────────────┘
         │
         ▼
┌──────────────────────────────────────────────────────────────────┐
│ Substring=substr(source string,string start,(string end-string start))│
│ Token=previous pattern.                                            │
└──────────────────────────────────────────────────────────────────┘
         │
         ▼
┌─────────────────────┐   no
│ Any more tokens?     │──────────────────────┐
└─────────────────────┘                       │
         │ yes                                 │
         ▼                                     │
┌─────────────────────┐                        │
│ Step to next token.  │                        │
└─────────────────────┘                        │
         │                                     │
         ▼                                     ▼
┌─────────────────────┐   no   ┌──────────────────────────┐
│ Token a variable or a│──────→│ Return.                   │
│ period?             │        └──────────────────────────┘
└─────────────────────┘
         │ yes
         ▼
┌─────────────────────┐   no
│ Any more tokens?     │──────────────────┐
└─────────────────────┘                   │
         │ yes                             ▼
         ▼                   ┌──────────────────────────┐
┌─────────────────────┐  no  │ Assign rest of substring │
│ Next token a variable│─────→│ to variable.             │
│ or a period?        │      └──────────────────────────┘
└─────────────────────┘
         │ yes
         ▼
┌─────────────────────┐   no   ┌──────────────────────────┐
│ Any substring left?  │──────→│ Assign null string to     │
└─────────────────────┘        │ variable.                 │
         │                     └──────────────────────────┘
         ▼
┌─────────────────────┐
│ Strip any leading    │
│ blanks.             │
└─────────────────────┘
         │
         ▼
┌─────────────────────┐   no   ┌──────────────────────────┐
│ Any substring left?  │──────→│ Assign null string to     │
└─────────────────────┘        │ variable.                 │
         │ yes                 └──────────────────────────┘
         ▼
┌─────────────────────┐   no   ┌──────────────────────────┐
│ Blank found in       │──────→│ Assign rest of substring │
│ substring?          │        │ to variable.             │
└─────────────────────┘        └──────────────────────────┘
         │ yes
         ▼
┌──────────────────────────────────────────────────────────┐
│ Assign word from substring to variable and step past blank.│
└──────────────────────────────────────────────────────────┘
```

Figure 5: Conceptual View of Word Parsing

Chapter 5. 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), or by division by 1.

- A zero result is always expressed as the single digit 0.

- Exponential form is used for a result depending on its value and the setting of NUMERIC DIGITS (the default is 9). If the number of places needed before the decimal point exceeds the NUMERIC DIGITS

setting, or the number of places after the point exceeds twice the NUMERIC DIGITS setting, the number is expressed in exponential notation:

```
1e6 * 1e6    ->    1E+12          /* not 1000000000000 */
1 / 3E10     ->    3.33333333E-11 /* not 0.0000000000333333333 */
```

Definition

A precise definition of the arithmetic facilities of the REXX language is given here.

Numbers

A **number** in REXX is a character string that includes one or more decimal digits, with an optional decimal point. (See "Exponential Notation" on page 158 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:
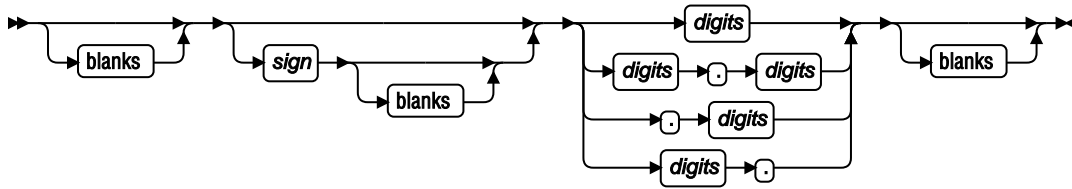
blanks
    are one or more spaces

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:
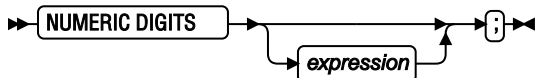
The *expression* is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) to which calculations are carried out. Results are rounded to that precision, if necessary.

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) 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:

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 160.) 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 157 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 9. You can use any of these for comparing numeric strings. However, you should not use ==, \==, ¬==, >>, \>>, ¬>>, <<, \<<, and ¬<< for comparing numbers because leading and trailing blanks and leading zeros are significant with these operators.

A comparison of numeric values is effected by subtracting the two numbers (calculating the difference) and then comparing the result with 0. That is, the operation:

```
A ? Z
```

where ? is any numeric comparison operator, is identical with:

```
(A - Z) ? '0'
```

It is, therefore, the *difference* between two numbers, when subtracted under REXX subtraction rules, that determines their equality.

A quantity called **fuzz** affects the comparison of two numbers. This controls the amount by which two numbers may differ before being considered equal for the purpose of comparison. The FUZZ value is set by the instruction:
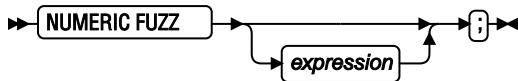
Here *expression* must result in a positive whole number or zero. The default is 0.

The effect of FUZZ is to temporarily reduce the value of DIGITS by the FUZZ value for each numeric comparison. That is, the numbers are subtracted under a precision of DIGITS minus FUZZ digits during the comparison. Clearly the FUZZ setting must be less than DIGITS.

Thus if DIGITS = 9 and FUZZ = 1, the comparison is carried out to 8 significant digits, just as though `NUMERIC DIGITS 8` had been put in effect for the duration of the operation.

Example:

```
Numeric digits 5
Numeric fuzz 0
say  4.9999 = 5    /* Displays "0"   */
say  4.9999 < 5    /* Displays "1"   */
Numeric fuzz 1
say  4.9999 = 5    /* Displays "1"   */
say  4.9999 < 5    /* Displays "0"   */
```

Exponential Notation

The preceding description of numbers describes "pure" numbers, in the sense that the character strings that describe numbers can be very long. For example:

```
10000000000 * 10000000000
```

would give

```
100000000000000000000
```

and

```
.00000000001 * .00000000001
```

would give

```
0.0000000000000000000001
```

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

For example:

```
numeric digits 5
say 54321*54321
```

would display 2950800000 in long form. This is clearly misleading, and so the result is expressed as
2.9508E+9 instead.

The definition of numbers is, therefore, extended as:
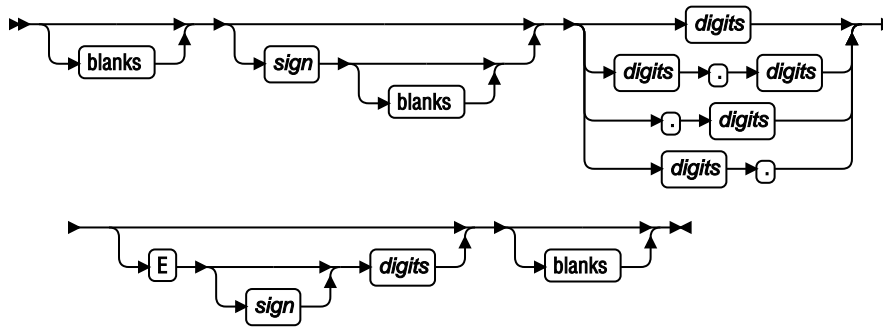
The integer following the E represents a power of ten that is to be applied to the number. The E can be in
uppercase or lowercase.

Certain character strings are numbers even though they do not appear to be numeric to the user.
Specifically, because of the format of numbers in exponential notation, strings, such as 0E123 (0 times 10
raised to the 123 power) and 1E342 (1 times 10 raised to the 342 power), are numeric. In addition, a
comparison such as 0E123=0E567 gives a true result of 1 (0 is equal to 0). To prevent problems when
comparing nonnumeric strings, use the strict comparison operators.

Here are some examples:

```
12E7   =    120000000           /* Displays "1" */
12E-5  =    0.00012             /* Displays "1" */
-12e4  =    -120000             /* Displays "1" */
0e123  =    0e456               /* Displays "1" */
0e123  ==   0e456               /* Displays "0" */
```

The preceding numbers are valid for input data at all times. The results of calculations are returned in
either conventional or exponential form, depending on the setting of NUMERIC DIGITS. If the number of
places needed before the decimal point exceeds DIGITS, or the number of places after the point exceeds
twice DIGITS, exponential form is used. The exponential form REXX generates always has a sign following
the E to improve readability. If the exponent is 0, then the exponential part is omitted—that is, an
exponential part of E+0 is never generated.

You can explicitly convert numbers to exponential form, or force them to be displayed in long form, by
using the FORMAT built-in function (see page "FORMAT" on page 87).

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:
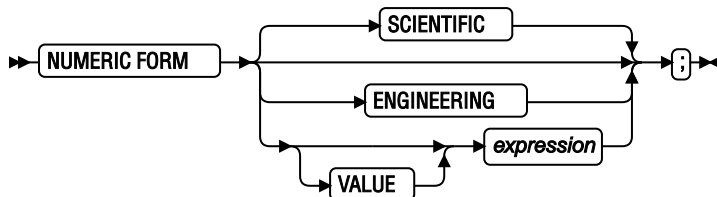
Scientific notation is the default.

```
/* 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, VM supports exponents in the range -999999999 through 999999999.

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

- Insufficient storage

  Storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail because of lack of storage. This is considered a terminating error as usual, rather than an arithmetic error.

Chapter 6. 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 29 and "SIGNAL" on page 59).
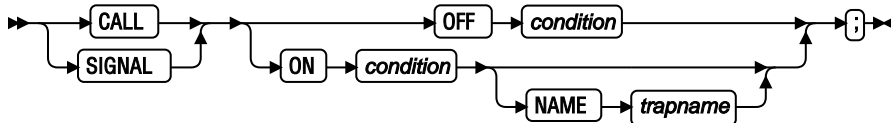
condition and *trapname* are single symbols that are taken as constants. Following one of these instructions, a condition trap is set to either ON (enabled) or OFF (disabled). The initial setting for all condition traps is OFF.

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. See note "3" on page 163.

CALL ON ERROR and SIGNAL ON ERROR trap all positive return codes, and negative return codes only if CALL ON FAILURE and SIGNAL ON FAILURE are not set. See "Exit Definitions" on page 196 for information about ERROR and the RXCMD exit.

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. See "Exit Definitions" on page 196 for information about FAILURE and the RXCMD exit.

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 CMS immediate command, HI (Halt Interpretation), creates a halt condition. See "Interrupting Execution and Controlling Tracing" on page 206.

See "Exit Definitions" on page 196 for information about halting and the RXHLT exit.

NOTREADY
raised if an error occurs during an input or output operation. See "Errors During Input and Output" on page 173. This condition is ignored if the NOTREADY condition trap is already in the delayed state.

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 Appendix A, "Error Numbers and Messages," on page 277) 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, NOTREADY, NOVALUE, or SYNTAX) is used.

For example, the instruction `call on error` enables the condition trap for the ERROR condition. If the condition occurred, then a call to the routine identified by the name ERROR is made. The instruction `call on error name commanderror` would enable the trap and call the routine COMMANDERROR if the condition occurred.

The sequence of events, after a condition has been trapped, varies depending on whether a SIGNAL or CALL is processed:

- If the action taken is a SIGNAL, execution of the current instruction ceases immediately, the condition is disabled (set to OFF), and the SIGNAL takes place in exactly the same way as usual (see "SIGNAL" on page 59).

    If any new occurrence of the condition is to be trapped, a new CALL ON or SIGNAL ON instruction for the condition is required to re-enable it when the label is reached. For example, if SIGNAL ON SYNTAX is enabled when a SYNTAX condition occurs, then, if the SIGNAL ON SYNTAX label name is not found, a usual syntax error termination occurs.

- If the action taken is a CALL (which can occur only at a clause boundary), the CALL is made in the usual way (see "CALL" on page 29) 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, FAILURE, or NOTREADY) 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).

Notes:

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, HALT, and NOTREADY traps can occur only at clause boundaries. If these conditions arise in the middle of an INTERPRET instruction, execution of INTERPRET may be interrupted and later resumed. Similarly, other instructions, for example, DO or SELECT, may be temporarily interrupted by a CALL at a clause boundary.

3. The state (ON, OFF, or DELAY, and any *trapname*) of each condition trap is saved on entry to a subroutine and is then restored on RETURN. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller. See the CALL instruction ("CALL" on page 29) for details of other information that is saved during a subroutine call.

4. The state of condition traps is not affected when an external routine is called by a CALL, even if the external routine is a REXX program. On entry to any REXX program, all condition traps have an initial setting of OFF.

5. While user input is processed during interactive tracing, all condition traps are temporarily set OFF. This prevents any unexpected transfer of control—for example, should the user accidentally use an uninitialized variable while SIGNAL ON NOVALUE is active. For the same reason, a syntax error during interactive tracing does not cause exit from the program but is trapped specially and then ignored after a message is given.

6. The system interface detects certain execution errors either before execution of the program starts or after the program has ended. SIGNAL ON SYNTAX cannot trap these errors.

Note that a **label** is a clause consisting of a single symbol followed by a colon. Any number of successive clauses can be labels; therefore, multiple labels are allowed before another type of clause.

Condition Information

When any condition is trapped and causes a SIGNAL or CALL, this becomes the current trapped condition, and certain condition information associated with it is recorded. You can inspect this information by using the CONDITION built-in function (see "CONDITION" on page 78).

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.

NOTREADY
> The fully-qualified name of the stream being manipulated when the error occurred and the NOTREADY condition was raised.

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 "RETURN" on page 56.)

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.

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. This may be followed, in CMS, by the following lines, so that by pressing ENTER you are placed in XEDIT as suggested previously:

```
call trace 'O'
address command 'DROPBUF 0'
parse source . . fn ft fm .
push 'Command :'sigl; push 'Command EMSG' errormsg
address cms 'Xedit' fn ft fm
exit rc
```

Chapter 7. Input and Output Streams

REXX defines only simple, character-oriented forms of input and output. In general, communication to or from the user is in the form of a stream of characters. You can manipulate these streams either character by character or line by line. These streams can be several things: minidisk files, SFS files, spool files (reader, printer, and punch), or the program stack. In addition to these character streams, an external data queue is defined for inter-program communication. You can access this queue only on a line-by-line basis. (The VM implementation of the queue is the program stack.)

In this discussion, input and output are described as though communicating with a human user. A character stream might, in fact, have a variety of sources or destinations. These may include files or displays. A character stream can be one of the following:

- **Transient** or dynamic, for example, the default input stream or the program stack
- **Persistent**, in a static form, for example, a file or data object.

Housekeeping for the character streams (opening and closing files, for example) is not explicitly part of the language because in most environments these operations are automatic; however, the STREAM built-in function is provided for miscellaneous stream commands for those operating environments that require them.

One default input stream and one default output stream are assumed.

Stream Formats

The I/O functions can read or write both variable-format and fixed-format streams. For minidisk and SFS files, new streams can be specified as variable or fixed depending on the characteristics of the stream. For example, SFS and minidisk files can be either variable or fixed, while printer, reader, and punch files can only be fixed. (See Table 2 on page 168.)

When writing to a fixed-format stream, the record being written will be padded with blanks if it is shorter than the logical record length of the output stream. If the record being written is longer than the logical record length of the output stream, the function will not write the line, and an appropriate error message will be returned.

Opening and Closing Streams

The STREAM function is provided as a means to explicitly open or close a stream. However, there are other ways of accomplishing an open or close. Any line or character I/O function will implicitly open a stream if it is not already open:

- CHARS and LINES will implicitly open the stream for WRITE if possible, READ otherwise.
- CHARIN and LINEIN will implicitly open the stream for READ.
- CHAROUT and LINEOUT will implicitly open the stream for WRITE.
- The STREAM function can be used to explicitly open a stream for READ or WRITE.

Each time an SFS file is opened, it is associated with a new work unit ID. This also applies to multiple opens of the same file. The stream will then remain open for subsequent I/O as long as an explicit close is not issued. A close can be done with the STREAM function or specific forms of CHAROUT and LINEOUT (as described in "CHAROUT (Character Output)" on page 76 and "LINEOUT (Line Output)" on page 90). If you do not explicitly close the stream, it remains open until the completion of the last active REXX program, at which time it is automatically closed. Prior to closing the stream, any remaining buffered data from character output operations is written out.

Note: The only files that can be opened multiple times are SFS and minidisk files.

Stream Names Used by the Input and Output Functions

All the input and output functions described in REXX may specify a *name*. This name defines the data stream or file that will be read from or written to. The following table shows the supported data streams and the associated stream name to be used in the REXX function call:

Table 2: Stream Names Used by the Input and Output Functions			
Data Stream	**Type**	**Length**	**Stream Name and Description**
Reader file	Transient	Fixed 80	*nnnn* RDRFILE CMSOBJECTS. where *nnnn* is the spool ID of the file. An asterisk (*) in the *nnnn* field can be used to specify the first file in the reader.
Punch	Transient	Fixed 80	VIRTUAL PUNCH CMSOBJECTS.
Printer	Transient	Variable 132	VIRTUAL PRINTER CMSOBJECTS.
SFS file	Persistent	Fixed or Variable	*filename filetype dirname* Asterisks may not be used anywhere in the file name or file type.
Minidisk or accessed SFS directory file	Persistent	Fixed or Variable	*filename filetype* or *filename filetype filemode* For an input operation, if no file mode or a file mode of asterisk (*) is supplied, the first instance of the file in the CMS search order will be used. For an output operation, *filemode* must be specified. Asterisks may not be used anywhere in the file name or file type.
Program Stack	Transient	Variable 255	PROGRAM STACK CMSOBJECTS. PROGRAM STACK CMSOBJECTS.LIFO PROGRAM STACK CMSOBJECTS.FIFO (Also referred to as the "external data queue".) LIFO and FIFO may be appended, to designate LIFO or FIFO stacking on output. If neither is specified, FIFO is the default and will be appended to the name of the stream when it is returned by the STREAM and CONDITION functions. When reading from the stack, lines are always read from the top. If the program stack has been opened LIFO, it cannot be used FIFO unless it is closed and reopened FIFO. Similarly, if the program stack has been opened FIFO, it cannot be used LIFO unless it is closed and reopened LIFO.

Table 2: Stream Names Used by the Input and Output Functions (continued)			
Data Stream	**Type**	**Length**	**Stream Name and Description**
Default Stream	Transient	Variable 255	"" (null string) For input, this reads from the terminal input buffer or from user input if the buffer is empty. For output, this displays on the user's console. The default stream is always open, therefore, you do not need to open it.

One or more blanks must separate the tokens of any stream name. If a stream name contains more than 254 characters, only the first 254 characters are kept. When doing I/O on a stream, certain error conditions will prevent any further I/O operations from succeeding. In these cases, you should close the stream, fix the error condition, and then you can continue processing.

In addition to the above names, all functions, except STREAM(*name*,'c','open'), accept one additional form of *name*. This is the unique identifier that is returned when the stream is opened with the STREAM function. This identifier is unique for each opening of the same named stream, provided that the stream can be opened more than one time. An example of obtaining and using this unique identifier is:

```
/* Open the file for write. */
parse value stream('TEST FILE A','C','OPEN WRITE') with ok file_handle
if ok ¬= 'READY:' then signal open_error
number_of_lines = lines(file_handle)
```

The file name, file type, and file mode for a minidisk file or for an SFS file are used exactly as supplied, so they should usually be given in uppercase. For all other pieces of any object name, case is insignificant.

Unit Record Device Streams

The unit record devices (virtual reader, virtual printer, and virtual punch) are accessed as transient streams. This means that you cannot specify a *start* position or *line* number for these streams. You can only write to the end of the stream (for printer or punch), or read from the next line (for the reader). The following information is specific to each device:

READER
> Both LINEIN and CHARIN may be used to get characters from the reader; LINES may be used to get the number of card images in the file, and CHARS may be used to determine if more characters exist. The reader must be defined at address '00C'.

PRINTER
> Both LINEOUT and CHAROUT are supported, and will write a line to the virtual printer of 132 characters or more. You may write out lines that have a length less than the width of the printer (padding to the printer size is not necessary). The printer must be defined at address '00E'.

PUNCH
> Both LINEOUT and CHAROUT are supported, and will write a line to the virtual punch of 80 characters. If the output line is more than 80 characters, nothing will be written, and an error will be indicated. If you write fewer than 80 characters, blanks will be added on the right to complete the card image. The punch must be defined at address '00D'.

Note: You should be aware of how CP handles spool files. In particular, if you open a reader file and do not read any records or only partially read the file, CP will discard the entire file when the spool file is closed. In order to prevent this, you can send a hold command to CP, using the SPOOL, CHANGE, or CLOSE commands. See *z/VM: CP Commands and Utilities Reference* for details.

The Input and Output Model

The model of input and output for REXX consists of three logically distinct parts, namely:

1. One or more character input streams
2. One or more character output streams
3. One external data queue.

REXX instructions and built-in routines manipulate these three elements as follows:

Character Input Streams

A character input stream is a serial character stream generated by user interaction, or having the characteristics of a stream so generated. Characters or lines or both can be read from a character input stream. Characters may be added to the end of some streams.

Here are the instructions that govern the use of input streams:

- The CHARIN function or the LINEIN function can directly read any named input stream as characters or lines, respectively.
- The instructions PULL and PARSE PULL can read the default input stream as lines if the external data queue is empty. (PULL is the same as PARSE PULL except that with PULL uppercase translation takes place).
- The PARSE LINEIN instruction can read lines from the default input stream regardless of the state of the external data queue. Usually, however, you would use PULL or PARSE PULL to read the default input stream.

The **read position** in a character input stream is the position from which the next character or line will be read. In a persistent stream, the language processor knows the current read position. You can modify the read position by using the STREAM function, however, you can only set it to the beginning of a line in the stream. You cannot position to a specific character offset in the stream.

- The CHARS function returns 1 if there are more characters currently available in an input character stream from the read position through the end of the stream. Returns 0 if no more characters are available.
- The LINES function returns the number of lines that remain between the current read position and the end of the stream. The number returned may include a partial line if CHARIN has been used to read some of the characters in the line.

In a transient stream, the CHARS and LINES functions can determine only if data is present in the stream.

Character Output Streams

A character output stream is a serial character stream to which characters or lines or both can be written. Character output streams provide for output from a REXX program. The:

- CHAROUT function can write any output stream in character form
- LINEOUT function can write any output stream as lines
- SAY instruction can write the default output stream as lines.

The **write position** in a character output stream is the position at which the next character or line will be written. The language processor knows the current write position in a stream. This is independent of the read position. You can modify the write position by using the STREAM function, however, you can only set it to the beginning of a line in the stream. You cannot position to a specific character offset in the stream.

The write position is usually the end of the stream (for example, when the stream is first opened) so that data can be appended to the end of the stream. For persistent streams, however, the STREAM function can direct sequential output starting at some arbitrary write position.

Note: After data has been placed in a transient character output stream, it is no longer accessible to REXX.

Physical and Logical Lines

CMS is not a character-based system, so some conventions have had to be adopted to support character-based input and output. The character functions are useful, however, and do allow CMS files and other data to be handled as character streams, even though actual (physical) input and output operations take place line by line.

Files in CMS have physical lines, each of which may contain arbitrary data (that is, all 256 character codes). Line operations read complete physical lines from the stream and write complete passed output data without any regard to the contents of the data. However, in order to properly support character operations, some manipulation of the data is necessary. REXX must know if line end characters are significant to the user or if they are not. An option on the OPEN command of the STREAM function, TEXT or BINARY, tells REXX if data contents are significant.

The BINARY option signifies that any character code could be in the stream and therefore, character operations will not manipulate the data in any way. Data read in is passed to the user without regard for or indication of line ends. Data to be written out is accumulated until enough characters are obtained to fill the output buffer (size can be defined by the LRECL parameter on OPEN), and then a line is written.

The TEXT option signifies that LINEEND characters are not included in the data stream, and line ends should be noted. Line end characters (as defined by the LINEEND parameter on OPEN) will be inserted at line ends in the data passed to the user on character input operations. On character output, the data to be written is scanned for the LINEEND character, and physical lines are written as appropriate. The LINEEND character is never written to the output stream. If a LINEEND character is not found, characters are accumulated until LRECL characters have been given and then the line is written out. For fixed length streams, lines will be padded with blanks as necessary.

When streams are either written and read with the line functions or with the character functions, the scheme of reading and writing LINEEND characters will work correctly. In general, problems may arise only when line and character functions are mixed. In this case, there is one problem to be aware of: **a stream can be written (with the LINEOUT function, or with other CMS tools) that has LINEEND characters embedded within lines. If the stream is copied with the character functions, and the BINARY option is not specified, such lines will be split at those characters**.

If part of a line has been read with the CHARIN function, then a subsequent call to the LINEIN function will return the remainder of that line. Similarly, if a line has been partly written with the CHAROUT function, then a call to LINEOUT will add to and complete the line. The general rule is that any call to a line function will treat the data manipulated as a unit and will affect a single physical line only.

The STREAM Function

The STREAM function determines the state of an input or output stream or carries out specific operations, described by *stream commands*. This stream command mechanism allows REXX programs to open and close selected streams for read-only or read and write operations, to move the read and write positions within a stream, and to access specific information about the stream.

The z/VM operating system, unlike some other systems, allows certain data streams to be opened multiple times. The recommended procedure on z/VM is to explicitly open the stream so that the user is able to obtain the unique identifier for that particular opening of the stream. This is especially important when a named data stream can be opened more than one time, and the unique identifier is needed to reference the different stream openings. When a stream is implicitly opened, the user is unable to obtain the unique identifier.

See for details.

External Data Queue—the General REXX SAA Model

The external data queue is a list of character strings that only line operations can access. It is external to REXX programs in that other programs can have access to the queue whenever REXX relinquishes control. The VM implementation of the queue is the program stack.

The external data queue, therefore, forms a language-defined channel of communication between programs. Data in the queue is arbitrary: no characters have any special meaning or effect.

Apart from the explicit REXX operations described here, no detectable change to the queue occurs during the execution of a REXX program, except when control leaves the program (for example, when an external command or routine is called). Here are the REXX queuing operations:

- The PULL or PARSE PULL instructions can remove lines from the queue. When the queue is empty, these instructions read lines from the default input stream. In this way, the external data queue can be a source of user input, provided that PULL or PARSE PULL reads the input as lines.
- The PUSH instruction can stack lines at the head of the queue.
- The QUEUE instruction can add lines to the tail of the queue.
- The QUEUED function returns the number of lines currently in the queue.

External Data Queue—VM Extensions

In addition to the general model, VM allows access to the program stack through the I/O functions CHARIN, CHAROUT, CHARS, LINEIN, LINEOUT, and LINES. This means that character operations are possible on the external data queue. The REXX keyword commands listed in the preceding section do indeed support only line operations, but CHARIN and CHAROUT can be used to read or write portions of a line. Also unlike the general REXX model, line end characters are significant when using CHARIN and CHAROUT on the external data queue (the program stack).

Implementation

Usually, the dialog between a REXX program and a user takes place on a line-by-line basis and is, therefore, carried out with the SAY and PULL (or PARSE PULL) instructions. This technique considerably enhances the usability of many programs, because they can be converted to programmable dialogs by using the external data queue to provide the input the user usually types. You should use the PARSE LINEIN instruction only when it is necessary to bypass the external data queue.

When a dialog is not on a line-by-line basis, use the explicitly serial interfaces the CHARIN and CHAROUT functions provide. These functions are especially important for input and output in transient character streams, such as keyboards or printers.

General I/O Information

Opening and closing of persistent streams, such as files, is largely automatic. Generally speaking, a stream is opened upon the first call of a line or character function and remains open until explicitly closed with the CHAROUT, LINEOUT or STREAM functions, or until all REXX programs end. A stream can also be opened or closed explicitly. This can be done with the STREAM function, or through specific use of the other I/O functions. For example, invoking the LINEOUT function with just the name of a stream (and no output line) closes the named stream.

All input and output to a given stream should be done exclusively with REXX or exclusively with the CMS supplied routines. REXX needs to maintain its own control blocks that are different from the ones CMS maintains. Mixing types of I/O can cause unpredictable results. One situation in particular, however, could happen under usual circumstances. A stream could get closed by CMS while REXX is actively doing I/O on the stream. This would occur when a shared file system file is being used, and a rollback happens. The file will no longer be open to REXX, and any attempted I/O on that file will generate an error with a special reason code. In this case, REXX will release the control block for that file and consider the file closed.

REXX uses data buffering while doing I/O, and, therefore, you must understand the implications of this. Data written with the CHAROUT function is placed in the I/O buffer (a temporary storage area for data being transferred to, or from, a stream) and perhaps not immediately written to the output stream. Several things can cause the data to be written immediately:

- A line end character is given
- Either I/O pointer is moved
- The stream is closed
- The I/O buffer is needed for an input operation
- A full buffer of data is accumulated.

You must pay particular attention to this buffering concept when using the Shared File System and coordinating the update of resources. Before you start any commit or rollback processing, you should ensure that all data in appropriate buffers is written out. Data remaining in a buffer during commit or rollback processing will be ignored for that operation.

It is also important to note that there is only one buffer for input and output. This is especially important when requesting both input from and output to the same stream. If the buffer contains data because of an input operation and an output operation is requested, the output data will replace the input data in the buffer. A subsequent read would first cause the output data to be written (if this had not already been done) and then the input record would be re-read. As mentioned earlier, there are several things that cause buffered output data to be written. This flip-flop of input and output data can cause performance degradation.

The z/VM implementation of input and output uses CSL routines to perform the actual I/O. The return codes and reason codes generated by the CSL routines that perform the I/O are documented in Appendix F, "Input and Output Return and Reason Codes," on page 307. For the lower level routine codes not listed there, see the following books:

- *z/VM: CMS Callable Services Reference* and *z/VM: CP Messages and Codes*: For SFS files, minidisk files, or the program stack.
- *z/VM: CMS Macros and Functions Reference*: For spool files (reader, punch, and printer).
- *z/VM: CP Programming Services*: For CP diagnose codes. (The value in the reason code field is actually the condition code.)
- *z/VM: CP Commands and Utilities Reference*: For CP commands.

Errors During Input and Output

REXX offers considerable flexibility in the handling of errors during input or output. This is provided in the form of a NOTREADY condition, which CALL ON and SIGNAL ON instructions can trap, and by the STREAM function, which can elicit further information. (See Chapter 6, "Conditions and Condition Traps," on page 161 for a more detailed discussion of SIGNAL ON and CALL ON.)

When an error occurs during an input or output operation, the function being called continues without interruption (for example, an output function returns a nonzero count). Depending on the nature of the operation, a program has the option of raising the NOTREADY condition. The NOTREADY condition is similar to the ERROR and FAILURE conditions associated with commands in that it does not cause a terminating error if the condition is raised but is not trapped.

After NOTREADY has been raised, the following possibilities exist:

NOTREADY condition is not being trapped	Execution continues without interruption. The NOTREADY condition remains in the OFF state.
SIGNAL ON NOTREADY is trapping the NOTREADY condition	The NOTREADY condition is raised, execution of the current clause ceases immediately, and the SIGNAL takes place as usual for condition traps.

CALL ON NOTREADY is trapping the NOTREADY condition	The NOTREADY condition is raised, but execution of the current clause is not halted—the NOTREADY condition is put into the delayed state, and execution continues until the end of the current clause.
	While execution continues, input functions that refer to the same stream may return the null string, and output functions may return an appropriate count, depending on the form and timing of the error. At the end of the current clause, the CALL takes place as usual for condition traps.
NOTREADY condition is in the DELAY state	(This occurs when CALL ON NOTREADY is trapping the NOTREADY condition, which has already been raised.) Execution continues, and the NOTREADY condition remains in the DELAY state.

After the NOTREADY condition has been raised and is in DELAY state, the CONDITION function (with the Description option) returns the name of the stream being processed when the stream error occurred. If the stream is a default stream and has no defined name, then the null string is returned.

The STREAM function then shows that the state of the stream is ERROR or NOTREADY, and you can get additional information on the state of the stream by specifying the Description option on the STREAM function.

Examples of Input and Output

In most circumstances, communication with a user running a REXX program is through the default input and output streams. For a question and answer dialog, the recommended technique is to use the SAY and PULL instructions (use PARSE PULL if case-sensitive input is required).

More generally, though, it is necessary to write to or read from streams other than the default. For example, to copy the contents of one file to another, one might use the following program:

```
/* This routine copies the stream or file named by  */
/* the first argument to the stream or file named   */
/* by the second, as lines.                         */
parse arg inputname, outputname

do while lines(inputname)>0
  call lineout outputname, linein(inputname)
  end
```

As long as lines remain in the named input stream, a line is read and is then immediately written to the named output stream. It is easy to modify this program so that it filters the lines in some way before writing them.

To illustrate how character and line operations might be mixed in a communications program, this example converts a character stream into lines:

```
/* This routine collects characters from the stream */
/* named by the first argument until a line is      */
/* complete, and then places the line on the        */
/* external data queue.                             */
/* The second argument is the single character that */
/* identifies the end of a line.                    */
parse arg inputname, lineendchar

buffer=''      /* zero-length character accumulator */
do forever
  nextchar=charin(inputname)
  if nextchar=lineendchar then leave
  buffer=buffer||nextchar          /* add to buffer */
  end
queue buffer /* place it on the external data queue */
```

Here each line is built in a variable called BUFFER. When the line is complete (for example, when the user presses Enter) the loop is ended and the contents of BUFFER are placed on the external data queue. The program then ends.

Summary of Instructions and Functions

CHARIN

This function reads zero or more single-byte characters from a character input stream. You can specify a start position of 1 to read from the beginning of a persistent stream. (See "CHARIN (Character Input)" on page 75.)

CHAROUT

This function writes zero or more single-byte characters to a character output stream. You can specify a start position of 1 to write from the beginning of a persistent stream. (See "CHAROUT (Character Output)" on page 76.)

CHARS

This function returns an indication if characters currently remain in a character input stream. (See "CHARS (Characters Remaining)" on page 77.)

LINEIN

This function reads zero lines or one line from a character input stream. You can specify a line number for persistent streams. (See "LINEIN (Line Input)" on page 89.)

LINEOUT

This function writes zero lines or one line to a character output stream. You can specify a line number for persistent streams. (See "LINEOUT (Line Output)" on page 90.)

LINES

This function returns the number of lines currently remaining in a character input stream. (See "LINES (Lines Remaining)" on page 91.)

PARSE LINEIN

This instruction reads one line from the default input stream. (See "PARSE" on page 48.)

PARSE PULL

This instruction reads one line from the external data queue. If the queue is empty it reads a line from the default input stream instead. (See "PARSE" on page 48.)

PULL

This instruction is the same as PARSE PULL except that the string read is translated to uppercase. (See "PULL" on page 53.)

PUSH

This instruction writes one line to the head of the external data queue, as in a stack. (See "PUSH" on page 54.)

QUEUE

This instruction writes one line to the tail of the external data queue. (See "QUEUE" on page 55.)

QUEUED

This function returns the number of lines currently available in the external data queue. (See "QUEUED" on page 93.)

SAY

This instruction writes one line to the default output stream. (See "SAY" on page 57.)

STREAM

This function returns a string describing the state of, or the result of an operation upon, a named character stream. (See "STREAM" on page 96.)

Chapter 8. System Interfaces

This chapter is addressed mainly to assembler language programmers and system programmers. It describes:

1. Calls to and from the language processor. A general description of calls to and from the REXX programs (from the CMS command line, from another exec, and so on) with an indication of the type of parameter list used in each case.

2. The CMS EXEC interface that receives calls to exec programs and passes them to the appropriate language processor.

3. Parameter lists. Details, at assembler language level, of the parameter lists used for calls to and from the language processor.

4. Function Packages. How to write a function or subroutine that the language processor can call and how to put it into a Function Package.

5. The EXECCOMM subcommand, which allows other programs to read and alter REXX variables and extract other information.

6. How the language processor sets and tests the flags in the exec flag control byte so as to obey the CMS immediate commands HI (Halt Interpretation), TS (Trace Start), and TE (Trace End).

Calls to and from the Language Processor

When called, the language processor can process either the Tokenized PLIST (Parameter List) or an Extended PLIST. When calling, the language processor generates both PLISTs. The language processor uses a special parameter list (subsequently referred to in this manual as the six-word Extended PLIST) for function calls and subroutine calls. The contents of the General Register 1 high order byte (Byte 0) define the format of the PLIST the caller passes.

Note: The general formats for CMS PLISTs (parameter lists) are described in the *z/VM: CMS Application Development Guide for Assembler*. The Extended PLIST and the six-word Extended PLIST are described later in this chapter.

Calls Originating from the CMS Command Line

To call a REXX language program, you can enter on the command line:

- Just the name of the program (*execname*) and the argument string. In this case, if IMPEX is ON (the default) and if the file execname EXEC exists, CMS issues the command EXEC, using the original command line as the argument string. If IMPEX is OFF, you cannot call the exec this way and must specify the word exec explicitly.

   Note: If ABBREV is ON (the default), CMS Command Line Processing searches the synonym tables.

- The command EXEC followed by the name of the REXX language exec (and the argument string, if any).

   Note: In this case synonyms are not recognized.

In both cases, CMS uses CMSCALL to call the exec. Register 0 points to the Extended PLIST and the user call-type information is a X'0B', indicating that:

- This is a CMS environment.

- CMS used the full CMS search order.

- An Extended PLIST is available.

CMS passes control to the language processor through the EXEC command handler (described under "The CMS EXEC Interface" on page 183).

Calls Originating from the XEDIT Command Line

To call a REXX macro that is stored in a file with a file type of XEDIT, you can enter on the XEDIT command line:

- Just the name of the macro and the argument string (if any). In this case, XEDIT runs the subcommand MACRO, using the original command line as the argument string. Note that if the macro has the same name as an XEDIT built-in command, the macro is not called unless MACRO is set ON (which is *not* the default).

- The command MACRO followed by the name of the REXX macro (and the argument string, if any). This always calls the specified macro, if it exists.

In both cases XEDIT checks to see if the macro is already loaded into storage. If not, it loads the macro if it exists, constructing an Extended PLIST, a File Block, and a Program Descriptor List. (For more information on Program Descriptor List, see "The File Block" on page 188.) Word 4 of the Extended PLIST points to the File Block and the user call-type information is a X'01'. CMS passes control to the language processor through the EXEC command handler (see "The CMS EXEC Interface" on page 183).

If you enter the name of the macro (*macroname* ...) on the XEDIT command line and the file `macroname` XEDIT is not found and IMPCMSCP is set ON, XEDIT assumes that an exec or a CMS command is being called and tries the full CMS search order for the command, as though the command had been entered from the CMS command line. In this case, the user call-type information is a X'0B' as usual.

Calls Originating from CMS Execs

Calls from CMS execs must explicitly call the exec, for example, EXEC name. Only the Tokenized PLIST is available. If the called exec is written in REXX, the CMS EXEC interface constructs an argument string from the Tokenized PLIST. The user call-type information is dependent upon the setting of the &CONTROL statement—X'0D' if MSG was specified (default), and X'0E' if NOMSG was specified.

Calls Originating from EXEC 2 Programs

Calls originating from EXEC 2 programs must explicitly call the exec, for example, EXEC name. However, EXEC 2 provides both the Tokenized PLIST and the Extended PLIST. The user call-type information is a X'01', which signifies that the Extended PLIST is available. An EXEC 2 program may also use &SUBCOMMAND CMS to simulate a call originating from the CMS command line.

Calls Originating from Alternate Format Exec Programs

When processing a call to a REXX exec, an alternate format exec processor may pass a Tokenized PLIST to the CMS EXEC interface, or it may pass both Tokenized and Extended PLISTS. On entry to the language processor, both PLISTS are available and the call-type information is X'01'. If the alternate format exec processor only passed a Tokenized PLIST, the Extended PLIST will point to an argument string constructed from data in the Tokenized PLIST.

Alternate format exec processors may also simulate the calls originating from the CMS command line by using the CMS SUBCOM environment.

Calls Originating from a Clause That Is an Expression

For a REXX clause that is an expression, the resulting string is issued as a command to whichever environment is currently selected (See "Commands to External Environments" on page 17). The PLIST format used depends on the environment selected (by default or by the ADDRESS instruction).

If the environment is COMMAND (or null), the command is issued directly: CMSCALL is issued with a CALLTYP of EPLIST (X'01'). (Note to EXEC 2 users: this is the way that EXEC 2 issues commands.)

If the environment is a valid PSW, the call is handled as described in "Non-SVC Subcommand Invocation" on page 189.

For all other environments, such as CMS, XEDIT, and so on, the call is handled using CMSCALL with a CALLTYP of SUBCOM (X'02'). For CMS, this results in the command being handled the same as from the CMS command line (same search order, same PLIST structure). In all cases, Register 1 points to a Tokenized PLIST that contains:

- The name of the subcommand entry point that is to be given control (8 characters long), such as CMS, XEDIT, and so on
- Two fullwords containing -1 (that is, X'FFFFFFFF').

Register 0 points to the Extended PLIST, containing the following pointers:

- To the environment name
- To the beginning of the argument string
- To the character after the end of the argument string
- A zero (indicating no File Block is given).

For example, if the following statement is in a REXX program:

```
address gkb 'calculate inverse'
```

then:

- R1 would point to the area containing an 8 character string 'GKB ' followed by two fullwords of -1
- R0 would point to an area containing all of the following:
  - A pointer to the string 'GKB '
  - A pointer to the string 'calculate inverse'
  - A pointer to the character after the e in inverse
  - A zero

Note that whether the environment is CMS or COMMAND, CMS Command Line Processing performs no cleanup after the command has been run, and interrupts are not canceled.

Calls Originating from a CALL Instruction or a Function Call

The language processor does not issue a command when processing a CALL instruction or function call to an external routine. The called routine may be a MODULE, a Nucleus Extension, or a REXX program; all use the same PLIST, but the language processor provides an FBLOCK only when the routine is called through the EXEC interface. The search order for external routines is described in "Search Order" on page 68.

In XA and XC virtual machines, the language processor can call modules above the 16MB line. The module can also pass data residing above the 16MB line back to the language processor because, in XA and XC virtual machines, REXX execs and XEDIT macros can reside above the 16MB line. AMODE 31 and ANY programs are called in 31-bit mode if they are called from the language processor in an XA or XC virtual machine. The following apply in an XA and XC virtual machine:

- REXX modules run in 31-bit addressing mode.
- REXX allocates REXX control blocks above the 16MB line.
- REXX handles interfaces between REXX programs and applications, regardless of their addressing mode.

If the module the language processor is calling has an AMODE of 24, the language processor calls the module in 24-bit mode and the following are copied below the 16MB line:

- The six-word extended PLIST.
- The argument list pointed to by the fifth word in the six-word extended PLIST.
- The strings the argument list points to.
- The sixth word in the PLIST.

In all cases, the user call-type information is a X'05', indicating that the six-word Extended PLIST is used. Word 5 of this PLIST points to the argument list (see Figure 6 on page 186). Word 6 points to a fullword location in USER storage, which is zero on entry and stores the address of an EVALBLOK if a result is returned. A routine that does not return a result must leave this location unchanged.

A routine called as a function *must* return a result, but a routine called as a subroutine need not. The caller sets Register 0 Bit 0 to:

0 if the routine is called as a function
1 if the routine is called as a subroutine.

(If the called routine is an exec written in REXX, you can obtain this information by using the PARSE SOURCE instruction, described in "PARSE" on page 48.)

If the REXX program is being called as a function, it must end with a RETURN or EXIT instruction with an expression, and the resulting string is returned in the form of an EVALBLOK. This EVALBLOK will be a 24-bit address if the caller is in 24-bit mode.

Calls Originating from a MODULE

REXX may be called from a user MODULE using any of the standard forms of PLIST:

- Only the Tokenized PLIST: The user call-type information is a X'00'. Register 0 is not used.
- The Extended PLIST: The user call-type information is a X'01'. Register 1 must point to a doubleword-aligned 16-byte field, containing

```
CL8'EXEC'
CL8'execname'
```

The rest of the Tokenized PLIST is not inspected. Register 0 must point to an Extended PLIST. The FBLOCK may be provided if desired (see "The File Block" on page 188).

- The six-word Extended PLIST: The user call-type information is X'05'. Other conditions are the same as for the Extended PLIST. You should use this form if more than one argument string is to be passed to the exec or if the exec is being called as a function. (Note that if the exec returns data in an EVALBLOK, it is the responsibility of the caller to free that storage.)

Note: You should use the CMSCALL macro to make your calls. CMSCALL has parameters that allow you to set up your PLISTs and your user call-type information. For example, if you use the COPY option, CMSCALL lets you pass a PLIST that resides above the 16MB line back to REXX. See *z/VM: CMS Application Development Guide for Assembler* for more information on the CMSCALL macro.

Calls Originating from an Application Program

An application program written in a language such as VS FORTRAN or OS/VS COBOL can call REXX using a callable services library (CSL) routine. Calling this routine is useful when the application program needs to call a CMS or CP command.

An application program can call a REXX exec through the CSL routine DMSCCE. The following is an example of the general call format for DMSCCE:
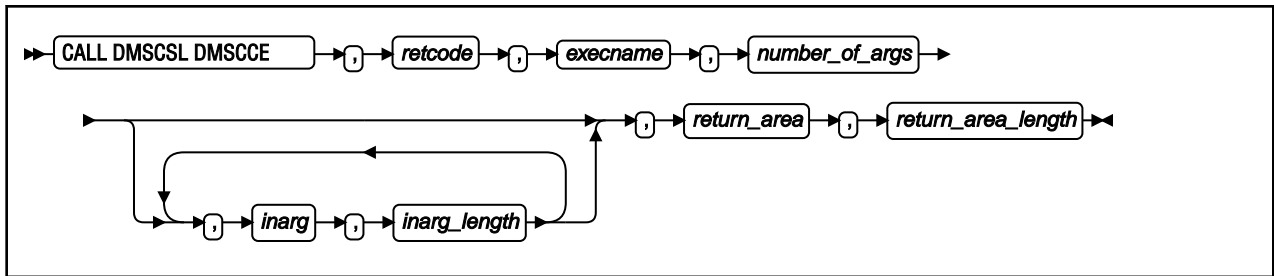
call to DMSCSL
> is the language-dependent format for invoking a callable services library (CSL) routine. The following list shows the general call format for calling DMSCCE using DMSCSL in the languages[3] that support CSL.

Assembler
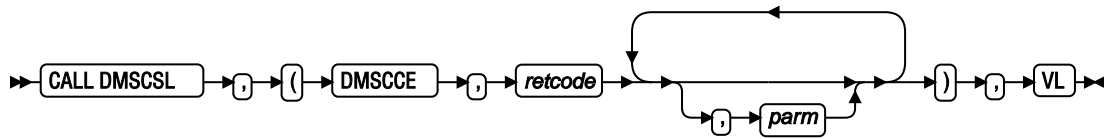
C
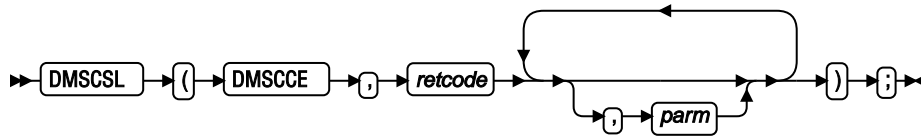
OS/VS COBOL or VS COBOL II
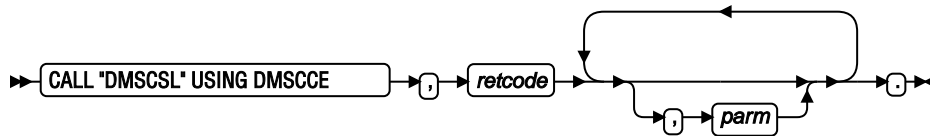
VS FORTRAN
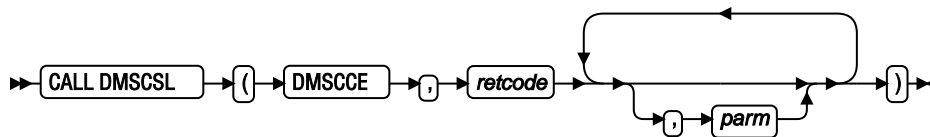
VS Pascal
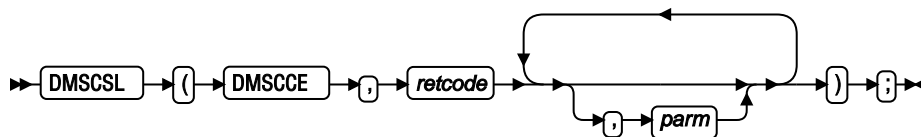
PL/I

---
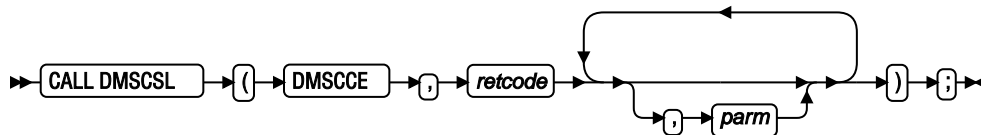
[3] It is not appropriate to use this CSL routine, DMSCCE, in a REXX program.

Additional language-specific statements may be necessary so that language compilers can provide the proper assembler interface. Other programming notation, such as variable declarations, is also language-dependent.

DMSCCE
is the name of the CSL routine being called. The value DMSCCE can be passed directly or in a variable. Note that you must pad two blanks on the right because the CSL routine name must be eight characters in length.

retcode
is a signed 4-byte binary variable to hold the return code from DMSCCE.

execname
is the name of the REXX EXEC being called. This field must be an 8-byte character string padded with blanks on the right if necessary, and it is used for input only.

number_of_args
is the number of input argument character strings being passed to the REXX exec. A maximum of 10 input character strings is allowed on a call. (See Usage Note "3" on page 183.) This field must be a 4-byte binary number, and it is used for input only.

inarg1 ... inargn
are the character string arguments passed to the REXX exec. These fields are used for input only.

inarg1_length ... inargn_length
are the lengths of the corresponding character string arguments. These fields must be 4-byte binary numbers, and they are used for input only.

return_area
is a buffer area to receive data from the REXX exec. This field must be a fixed-length character string, and it is used for output only.

return_area_length
on input, this is the length of *return_area*; on output, this is the length of the data returned in *return_area*. (See Usage Note "4" on page 183.) It must be a 4-byte binary integer.

For more information on calling REXX using a callable services library routine, see *z/VM: CMS Callable Services Reference*.

Usage Notes:

1. This routine is useful when the application needs to call some CMS or CP command. The REXX exec issues the CP or CMS command and passes the results back to the application program.

2. An example of a good way to use DMSCCE is to issue a FILEDEF command from an application program. A REXX exec named DATADEF issues the FILEDEF command. The following code fragment from a PL/I program shows an example of this:

```
  ⋮
 /* Declares for parameters of CALL statement */
   DCL DMSCCE    CHAR(8)       INIT('DMSCCE'),
       RETCODE   FIXED BIN(31) INIT(0),
       DATADEF   CHAR(8)       INIT('DATADEF'),
       ONE       FIXED BIN(31) INIT(1),
       ARG       CHAR(37) INIT('INFILE DISK FILENAME FILETYPE A (PERM'),
       ARGL      FIXED BIN(31) INIT(37),
       RET       CHAR(10)      INIT(' '),
       RETL      FIXED BIN(31) INIT(10);

 /* Call statement to DATADEF EXEC */
   CALL DMSCSL (DMSCCE,RETCODE,DATADEF,ONE,ARG,ARGL,RET,RETL);
  ⋮
```

After the application program issues the preceding CALL statement, the FILEDEF command is run using the arguments supplied in the "ARG" parameter.

Note: Using DMSCCE to issue a FILEDEF command is especially useful if your application program calls the SAA file-related functions OPEN, READ, WRITE, or CLOSE. Your program can be portable

across different IBM systems when you use SAA functions; however, a program must issue a FILEDEF before calling an SAA file-related function.

An application program can use the VM-specific shared file system routines to perform an OPEN, READ, WRITE, or CLOSE, but the program would not be portable across systems.

3. Although you cannot specify more than 10 arguments on a call to DMSCCE, an argument string can represent multiple variables. For example, you could pass ʼvar1 var2 var3 var4 var5ʼ as a single argument string, and this single string can be parsed into five separate variables.

4. If the data returned from the REXX exec is longer than the length of *return_area*, the data is truncated and a return code of 200 is issued.

Return Codes:

0
 Usual completion.

20
 Incorrect CMS character in EXEC name.

28
 The REXX exec specified on the call does not exist.

112
 The number of parameters passed on the call was incorrect.

118
 The parameter list passed to the routine was not in a valid format.

123
 The number of arguments passed to the REXX exec exceeded the number specified in *number_of_args*.

200
 The data returned in *return_area* has been truncated. (The *return_area_length* variable contains the length of the data before it was truncated.)

10*nn*
 The data type for parameter *nn* is incorrect.

20*nn*
 The length for parameter *nn* is incorrect.

Calls Originating from CMS Pipelines

CMS Pipelines can call a REXX program to process and transport data in a pipeline. This program is called a user-written stage and has a file type of REXX to distinguish it from other exec procedures. You can store a user-written stage using the EXECLOAD command or install a user-written stage in a shared segment like all other REXX programs.

When a user-written stage is called from CMS Pipelines, the default environment to which commands are passed is CMS Pipelines, not the CMS environment. To issue CMS or CP commands from a user-written stage, use the REXX ADDRESS instruction.

The CMS EXEC Interface

All calls to the CMS command EXEC are first processed by the CMS EXEC interface, which builds any necessary argument strings and also selects the language processor that is to process the program.

This selection is done by inspecting the call-type and PLISTs, and reading up to 255 bytes of the first line of the program file.

1. If a six-word PLIST is available (call-type X'05'), the first line of the program is read to determine whether the program should be passed to an alternate format exec processor or to the language processor.

   If the program is not an alternate format exec, it is assumed to be a REXX exec.

2. If an Extended PLIST containing an FBLOCK pointer is available (call-type X'01' or X'0B' and word 4 of the Extended PLIST is not 0), the first line of the program is read to determine whether the program should be passed to an alternate format processor, the language processor, or the EXEC 2 processor.

   If the program is not an alternate format exec and not a REXX exec, it is assumed to be an EXEC 2 exec.

3. If an Extended PLIST containing an FBLOCK is not available, the first line of the program is read to determine whether the program should be passed to an alternate format exec processor, the language processor, the EXEC 2 processor, or the CMS EXEC processor.

   If the program is not an alternate format exec, not a REXX exec, and not an EXEC 2 exec, it is assumed to be a CMS EXEC exec.

   The forms of the different kinds of execs are mutually exclusive.

   • REXX execs must begin with a begin-comment delimiter /∗.

   • EXEC 2 execs must begin with the string &TRACE.

   • CMS EXEC execs must begin with a comment delimiter ∗.

   • Alternate format execs cannot begin with a REXX or CMS EXEC comment and must contain the string EXECPROC in positions 5 through 12. The string in positions 13 through 20 is assumed to be the name the alternate format exec processor.

   Once the exec type (alternate format, REXX, EXEC 2, or CMS EXEC) has been determined, the CMS EXEC interface calls the appropriate language processor. For alternate format execs, this is done using CMSCALL.

The Extended Parameter List

The language processor may be called with an Extended PLIST (in addition to the 8-byte Tokenized PLIST) that allows the following possibilities:

• One or more arbitrary parameter strings (mixed case and untokenized) may be passed to the language processor, and one string may be returned from it when execution ends.

• A file other than that defined in the Tokenized PLIST may be used. (The file type, for example, need not be EXEC).

• A default target for commands (other than CMS) can be specified. If a file type other than EXEC or blanks is specified, then it is stored in the File Block. The language processor can then use the information in the File Block to send commands to the appropriate environment.

• A program that exists in storage may be run (instead of first being read from a file). This in-storage execution option may be used for improved performance when a REXX program is being run repeatedly.

• A default target for commands may be specified that overrides the default derived from the file type.

Using the Extended Parameter List

To use the Extended PLIST, both Register 1 and Register 0 are used. Register 1 points to the Tokenized PLIST. The first token of this PLIST must be CL8'EXEC', and the second token must contain the name of the exec or macro to be processed unless a FBLOCK that specifies the name is provided.

The user call-type information may have the following values:

X'01' or X'0B'

Extended PLIST available. The argument string defined by words 2 and 3 (BEGARGS and ENDARGS) of the Extended PLIST finds the called name of the program and the argument string passed to the language processor. The first two tokens of the Tokenized PLIST are used.

X'05'

A language processor call (for example, originating from a CALL instruction or a function call to a REXX external routine). The six-word Extended PLIST is available. The argument list pointed to by Word 5 of the PLIST is used for the strings accessed by the ARG instruction and the ARG function. Only the first token of the Tokenized PLIST is used. If the argument list is specified, only the first word of the BEGARGS/ENDARGS string is used (for the called name of the program).

Any other value

(for example, X'00') Only the Tokenized PLIST is available.

Note: You should use the CMSCALL macro to make your calls. CMSCALL has parameters that allow you to set up your user call-type information. Register 0 points to the Extended PLIST. The Extended PLIST has the form:

```
EPLIST DS  0F                PLIST with pointers:
       DC  A(COMVERB)        -> CL5'EXEC '
       DC  A(BEGARGS)        -> start of Argstring
       DC  A(ENDARGS)        -> character after end of
*                                 the Argstring
       DC  A(FBLOCK)         -> File Block, described following.
*                               (if there is no File Block,
*                                this pointer must be 0)
```

The six-word Extended PLIST (which exists only if the user call-type information is X'05') is the previous four pointers followed by two additional pointers:

```
       DC  AL4(ARGLIST)      -> Argument list.
*                               If there is no argument
*                               list, this pointer is 0,
*                               and BEGARGS/ENDARGS are
*                               used for the ARG string.
       DC  A(SYSFUNRT)       -> RETURN location.  This location:
*                               - contains a zero on entry;
*                               - will be unchanged if
*                                 no result is returned;
*                               - will contain the address of an
*                                 EVALBLOK if a result is returned.
```

The following example shows the use of the Extended PLIST.

```
************* This is the sample assembler program used to call
************* the REXX function using the six-word extended PLIST


* Sample program to call a REXX function
CALLFUN  CSECT
         USING *,R12
         STM   R14,R12,0(R13)  Save registers
         ST    R13,R13SAVE     Keep address of save area
*
         CMSCALL PLIST=PLIST,EPLIST=EPLIST,CALLTYP=FUNCTION
*                             Call the REXX program
         LINEWRT DATA=MESSAGE  Type message line
         L     R10,SYSFUNRT    Get the address of the EVALBLOK
         USING EVALBLOK,R10    Get addressability to it
         LA    R2,EVDATA       Get address of the result
         L     R3,EVLEN        Get result length
         DROP  R10
         LINEWRT DATA=((R2),(R3))  Display the result
         L     R13,R13SAVE     Get address of save area
         LM    R14,R12,0(R13)  Restore registers
         XR    R15,R15         Give zero return code
         BR    R14             Return to caller
*
*  PLIST, EPLIST and constants
*
         DS    0D
PLIST    DC    CL8'EXEC'        Declare the Tokenized PLIST
         DC    CL8'MYEXEC'
*                               Declare the Extended PLIST
EPLIST   DC    A(COMVERB)
         DC    A(BEGARGS)
         DC    A(ENDARGS)
         DC    F'0'
         DC    A(ARGLIST)
         DC    A(SYSFUNRT)
*
COMVERB  DC    CL5'EXEC '
BEGARGS  EQU   *
         DC    CL7'MYEXEC '
ARG1     DC    C'This is the 1st arg string '
ARG2     DC    C'This is the 2nd '
ARG3     DC    C'This is the 3rd string.'
ARG4     DC    C'This is the 4th '
ENDARGS  EQU   *
```

Figure 6: SAMPLE CALL (Part 1 of 2)

```
*
ARGLIST  DS    0F              Adlen pairs for the arguments
         DC    A(ARG1,L'ARG1)
         DC    A(ARG2,L'ARG2)
         DC    A(ARG3,L'ARG3)
         DC    A(ARG4,L'ARG4)
         DC    F'-1'           fence
         DC    F'-1'
SYSFUNRT DC    F'0'            RETURN area
R13SAVE  DS    F               pointer to save area
MESSAGE  DC    C'+++This is the returned string from MYEXEC:'
*
EVALBLOK DSECT
EVNEXT   DS    F               Reserved
EVSIZE   DS    F               size in DW's
EVLEN    DS    F               length of data in bytes
EVPAD    DS    F               (reserved)
*
LEVALBLO EQU   *-EVALBLOK      length of basic area
EVDATA   DS    C               ****    Start of data...
         REGEQU                register equates
         END CALLFUN


*************** This is the REXX function


/* MYEXEC EXEC - sample function program */
say 'In MYEXEC'
parse arg in1,in2,in3,in4
say 'Arg string 1 = *'in1'*'
say 'Arg string 2 = *'in2'*'
say 'Arg string 3 = *'in3'*'
say 'Arg string 4 = *'in4'*'
say 'Leaving MYEXEC....'
exit 'This is the exit string'


*************** this is the console log

callfun
In MYEXEC
Arg string 1 = *This is the 1st arg string *
Arg string 2 = *This is the 2nd *
Arg string 3 = *This is the 3rd string.*
Arg string 4 = *This is the 4th *
Leaving MYEXEC....
+++This is the returned string from MYEXEC:
This is the exit string
Ready; T=0.01/0.01 14:23:54
```

Figure 7: SAMPLE CALL (Part 2 of 2)

The **argument list** consists of an **Adlen** (Address/Length) pair for each argument string. The final value pair is followed by two fullwords containing -1 (that is, X'FFFFFFFF'). There is no limit to the number of strings when the language processor is called, but note that the language processor itself provides only from zero to 20 argument strings.

If the argument list is given, the simple argument string (as defined by BEGARGS and ENDARGS) is not used for the ARG instruction or the ARG built-in function.

Note: The argument list and the strings it defines must be in privately owned storage. This means that the language processor need not copy the data strings before using them (as is done for the BEGARGS/ENDARGS string, when it is used).

The **result** of a subroutine or function call using the six-word Extended PLIST is returned in a block of USER storage allocated by CMS Storage Management; this has the following storage assignments and values:

```
*-- DSECT for the returned data block -----------------*
EVALBLOK DSECT
EVNEXT   DS    F          Reserved
```

```
EVSIZE   DS   F           Total block size in DW's
EVLEN    DS   F           Length of Data (in bytes)
EVPAD    DS   F           Reserved -- should be set to 0
EVDATA   DS   C...        The returned character string
```

A result may only be returned if the called routine ends cleanly, with a Register 15 return code of 0.

Note: The EVALBLOK can be either above or below the 16MB line if the caller is AMODE 31 (31-bit addressing). The EVALBLOK must be below the 16MB line if the caller is AMODE 24 (24-bit addressing).

The File Block

Word 4 of the Extended PLIST described previously points to the FBLOCK. It is only needed if the language processor is to run a non-EXEC file or is to run from storage, or is to have an address environment that is not the same as its file type. If it is not required, word 4 of the Extended PLIST should be set to 0. For the format of the FBLOCK macro, refer to the data areas and control blocks information in the IBM z/VM Internet Library at IBM: z/VM Internet Library (www.ibm.com/vm/library).

Note: GCS does not support the FBLENAME portion of the FBLOCK extension. The GCS FBLOCK macro reserves the fifth and sixth fullwords.

The descriptor list for an in-storage program looks like this:

```
** Descriptor list for in-storage program
PROG  DS  0F                ** In storage program **
      DC  A(line1),F'len1'  Address, length of line 1
      DC  A(line2),F'len2'  Address, length of line 2
      ....
      DC  A(lineN),F'lenN'  Address, length of line N
PGEND EQU *
```

Note:

1. The in-storage program lines need not be contiguous, because each is separately defined in the descriptor list.

2. For in-store execution, the *filename* is still required in the File Block, because this determines the logical program name. The *filetype* similarly sets the default command environment, unless the name in the extension block explicitly overrides it.

3. If the extension length is >= 4 fullwords, the 3rd and 4th fullwords (FBLPREF) form an 8-character environment address that overrides the default address set from the *filetype* in the File Block and thus forms the initial ADDRESS to which commands are issued. This new address may be all characters (for example, blank, CMS, or some other environment name), or it may be a PSW for non-SVC subcommand execution—described in "Non-SVC Subcommand Invocation" on page 189. It may be cleared to 8X'00' if not required.

4. If the extension length is >= 6 fullwords, the 5th and 6th fullwords (FBLENAME) form an 8-character environment name that is used for the default address unless this is a non-SVC command execution. In this case, the 3rd and 4th fullwords (FBLPREF) are used as a PSW for non-SVC subcommand execution—described in "Non-SVC Subcommand Invocation" on page 189. PARSE SOURCE and the ADDRESS built-in function return the environment name, and the PSW in the 4th and 5th fullwords calls subcommands.

5. Exits are defined at language processor invocation by means of a specified FBLOCK extension. The FBLOCK extension contains a pointer in the seventh fullword of the extension block that points to the exit. The eighth fullword of the extension block passes a user word value that is returned to the parameter list when an exit is entered. See "REXX Exits" on page 195 for a description of exits.

6. When an EXEC or XEDIT macro has been loaded into storage through the EXECLOAD command and the EXEC or XEDIT macro is invoked through a CMSCALL on which a FBLOCK is supplied, the high-order bit of the FBLOCK address (Word 4 of the Extended PLIST) must be set on in order for the usage count reported by the EXECMAP command to be incremented. A high-order bit of 1 in the address of the FBLOCK indicates that the EXEC is in storage as a result of an EXECLOAD command.

Function Packages

Functions and subroutines can be written in REXX or in any other language that has an interface that conforms to the six-word Extended PLIST described previously. Those routines not written in REXX may be supplied simply as a file with a file type of MODULE. For a further improvement in performance, routines that are called frequently may be loaded as Nucleus Extensions, or placed in a Function Package.

A function package contains the code for functions that are candidates for loading as nucleus extensions. The first time a function in one of the three packages known to the language processor (RXUSERFN and RXLOCFN and RXSYSFN) is called, a call to the package with a LOAD request causes the package to load itself as a Nucleus Extension (if it is not already in storage). The entry point to the particular function required is then declared as a Nucleus Extension by the package. On subsequent calls, the code for the function is directly available using CMSCALL and the extra processing for loading the package MODULE is avoided. The functions in a package usually share common code and subroutines. For an example of a function package, see Appendix D, "Example of a Function Package," on page 297.

See "Search Order" on page 68 for the full search order of external routines.

All external routines are called using the six-word Extended PLIST defined previously in "Using the Extended Parameter List" on page 184. If the called routine is not an exec or macro (that is, EXEC does not process it), then word 4 is zero. Word 5 points to the list of arguments, and word 6 points to a location that may be used to return the address of an EVALBLOK that will contain the result of the function or subroutine. If the routine is being called as a subroutine (rather than as a function), so that it need not return a result, then the top bit of R0 is set to indicate this. Otherwise the routine should return a result—the language processor raises an error if it does not.

During calculation of the result, the routine may use the argument strings (which reside in USER storage the language processor owns) as work areas, without fear of corrupting internal REXX values.

External function packages must be able to respond to a call of the form:

```
RXnameFN   LOAD   RXfname
```

(which is issued using just the Tokenized PLIST, with the user call-type information being X'00').

When the package RXnameFN is called with this request, if RXfname is contained within the package, RXnameFN:

- Loads itself, if necessary
- Installs the nucleus extension entry point for the function
- Returns with a return code 0.

Otherwise, the return code is 1. This allows the language processor to automatically load the function packages and entry points when necessary.

Non-SVC Subcommand Invocation

When a command is issued to an environment, there is an alternative non-SVC fast path available for issuing commands. This mechanism may be used if an environment wishes to support a minimum-overhead subcommand call.

The fast path is used if the current eight-character environment address has the form of a PSW (signified by the fourth byte being X'00'). This address may be set using the Extended PLIST (see previous description) or by using the ADDRESS instruction if the PSW has been made available to the exec in some other way. Note that if a PSW is used for the default address, the PARSE SOURCE string uses ? as the name of the environment unless an environment name has also been provided. The PSW must be in a correct format for the addressing mode you are running in: a 370 PSW for 24-bit addressing, and an XA or XC PSW for 31-bit addressing.

The definition of the interface follows:

1. The language processor passes control to the routine by executing an LPSW instruction to load the eight-byte environment address. On entry to the called program the following registers are defined:

    Register 0
    Extended PLIST as per usual subcommand call. First word contains a pointer to the PSW used; second and third words define the beginning and end of the command string; and the fourth word is 0.

    Register 1
    Tokenized PLIST. First doubleword contains the PSW used; second doubleword is 2F'-1'. Note that the top byte of Register 1 does not have a flag.

    Register 2
    is the original Register 2 as encountered on the initial entry to the language processor's external interface. This register is intended to allow for the passing of private information to the subcommand entry point, typically the address of a control block or data area. This register is safe only if the exec is called with a BALR to the entry point contained at label AEXEC in NUCON; otherwise the SVC processor alters this register.

    Register 13
    points to an 18 fullword save area.

    Register 14
    contains the return address.

    (*All* other registers are undefined.)

2. It is the called program's responsibility to save Registers 9 through 12 and to restore them before returning to the language processor. All other registers may be used as work registers.

3. On return to the language processor, Registers 9 through 12 must be unchanged (see Note 2 preceding), and Register 15 should contain the return code (which is placed in the variable RC as usual). Contents of other registers are undefined. The language processor sets the storage key and mask that it requires.

Note: The EXECCOMM subcommand entry point is always set up when execution of a REXX program begins, even if the exec is called through BALR. This results in a subcommand block being added to the SUBCOM chain.

Direct Interface to Current Variables

The language processor provides an interface whereby called commands can easily access and manipulate the current generation of REXX variables. Variables may be inspected, set, or dropped, and if required all active variables can be inspected in turn. The language processor's own routines manipulate internal work areas: therefore, user programs do not need to know anything of the structure of the variables' access method. The interface code checks names for validity, and optionally substitution into compound symbols is carried out according to usual REXX rules. Certain other information about the program that is running is also made available through the interface.

The EXECCOMM interface to REXX is 31-bit capable. That is, the address in the parameter list can be above the 16MB line. However, programs supporting calls from both EXEC 2 and REXX need to ensure that all areas reside below the 16MB line.

The interface works as follows:

When the language processor starts to process a new program, the program first sets up a **subcommand entry point** called EXECCOMM. When the language processor calls a program (command, subcommand, or external routine), the program may in turn use the current EXECCOMM entry point to set, fetch, or drop REXX variables, using the language processor's internal mechanisms. Part of the language processor carries out all changes to pointers, allocation of storage, substitution of variables in the name, and so forth, and therefore isolates user programs from the internal mechanisms of the language processor.

To access variables, EXECCOMM is called using both the Tokenized and the Extended PLIST (see also "The Extended Parameter List" on page 184). CMSCALL is issued with R1 pointing to the usual Tokenized PLIST, and the user call-type information set to X'02', as this is a subcommand call.

The R1 PLIST: Register 1 must point to a PLIST which consists of the 8-byte string EXECCOMM .

The R0 PLIST: Register 0 must point to an Extended PLIST. The first word of the PLIST must contain the value of Register 1 (without the user call-type information in the high-order byte). No argument string can be given, so the second and third words must be identical (for example, both 0). The fourth word in the PLIST must point to the first of a chain of one or more request blocks; see the following.

On return from the CMSCALL, **Register 15** contains the return code from the entire set of requests. The possible return codes are:

POSITIVE
Entire PLIST was processed. Register 15 is the composite OR of bits 0-5 of the SHVRET bytes of the Request Block (SHVBLOCK).

0
Entire PLIST processed successfully.

-1
Incorrect entry conditions (for example, BEGARGS ¬= ENDARGS, or EXECCOMM is being called when the language processor is active).

-2
Insufficient storage was available for a requested SET. (See the note in Chapter 1, "REXX General Concepts," on page 1.) Processing was ended (some of the request blocks may remain unprocessed— their SHVRET bytes are unchanged).

-3
(from SUBCOM). No EXECCOMM entry point found; for example, not called from inside a REXX program.

The Request Block (SHVBLOCK)

Each request block in the chain must be structured as the SHVBLOCK macro. For the format of the SHVBLOCK macro, refer to the data areas and control blocks information in the IBM z/VM Internet Library at IBM: z/VM Internet Library (www.ibm.com/vm/library).

A typical calling sequence using fully relocatable and read-only code might be:

```
      LA  R0,EPLIST           -> Extended PLIST, same format as
                                 the R0 PLIST described previously.
                              -> Set up the call using CMSCALL.
                                 CMSCALL will take care of the
                                 user call-type information,
                                 set up the address of the
                                 Extended PLIST and Tokenized
                                 PLIST and set up the
                                 error routine address.
      CMSCALL EPLIST=(R0),PLIST=EXNAME,CALLTYP=SUBCOM,ERROR=DISASTER
      BM  DISASTER            Where to go if return code not equal to 0
  .
  .
  .
 EXNAME    DC  CL8'EXECCOMM'        Tokenized PLIST
           DC  XL8'FFFFFFFFFFFFFFFF'  Fence for PLIST copy
```

Function Codes (SHVCODE)

Three function codes (D, F, and S) may be given in either lowercase or uppercase:

Lowercase
(The **Symbolic** interface). The names must be valid REXX symbols (in mixed case if desired), and usual 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 forth) are permitted following a valid REXX stem.

Note: The **Direct** interface, which is also provided (in part) by EXEC 2, should be used in preference to the **Symbolic** interface whenever generality is desired.

The other function codes, N and P, must always be given in uppercase. The specific actions for each function code are as follows:

D and d

Drop variable. The SHVNAMA/SHVNAML adlen describes the name of the variable to be dropped. SHVVALA/SHVVALL are not used. The name is validated to ensure that it does not contain incorrect characters, and the variable is then dropped, if it exists. If the name given is a stem, all variables starting with that stem are dropped.

F and f

Fetch variable. The SHVNAMA/SHVNAML adlen describes the name of the variable to be fetched. SHVVALA specifies the address of a buffer into which the data is to be copied, and SHVBUFL contains the length of the buffer. The name is validated to ensure that it does not contain incorrect characters, and the variable is then located and copied to the buffer. The total length of the variable is put into SHVVALL, and, if the value was truncated (because the buffer was not big enough), the SHVTRUNC bit is set. If the variable is shorter than the length of the buffer, no padding takes place. If the name is a stem, the initial value of that stem (if any) is returned.

SHVNEWV is set if the variable did not exist before the operation, and in this case the value copied to the buffer is the derived name of the variable (after substitution and so forth); see "Compound Symbols" on page 14.

N

Fetch Next variable. This function can search through all the variables known to the language processor (that is, all those of the current generation, excluding those *hidden* by PROCEDURE instructions). The order in which the variables are revealed is not specified.

The language processor maintains a pointer to its list of variables: this is reset to point to the first variable in the list whenever a host command is issued, or any function other than N or P is run through the EXECCOMM interface.

Whenever an N (Next) function is run, the name and value of the next variable available are copied to two buffers supplied by the caller.

SHVNAMA specifies the address of a buffer into which the name is to be copied, and SHVUSER contains the length of that buffer. The total length of the name is put into SHVNAML, and if the name was truncated (because the buffer was not big enough) the SHVTRUNC bit is set. If the name is shorter than the length of the buffer, no padding takes place. The value of the variable is copied to the user's buffer area using exactly the same protocol as for the Fetch operation.

If SHVRET has SHVLVAR set, the end of the list of known variables has been found, the internal pointers have been reset, and no valid data has been copied to the user buffers. If SHVTRUNC is set, either the name or the value has been truncated.

By repeatedly executing the N function (until the SHVLVAR flag is set) a user program can locate all the REXX variables of the current generation.

P

Fetch private information. This interface is identical with 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), and the following names are recognized:

ARG

Fetch primary argument string. The first argument string that would be parsed by the ARG instruction is copied to the user's buffer.

PARM

Fetch the number of argument strings. The number of argument strings supplied to the program is placed in the caller's buffer. The number is formatted as a character string.

Note: When specifying PARM, each letter *must* be supplied.

PARM.*n*

Fetch the *n*th argument string. Argument string *n* is placed in the caller's buffer. Returns a null string if argument string *n* cannot be supplied (whether omitted, null, or fewer than *n* argument strings specified). Parm.1 returns the same result as ARG.

Note: When specifying PARM.*n*, 'PARM.' *must* be supplied.

SOURCE

Fetch source string. The source string, as described for PARSE SOURCE in "PARSE" on page 48, is copied to the user's buffer.

VERSION

Fetch version string. The version string, as described for PARSE VERSION in "PARSE" on page 48, is copied to the user's buffer.

S and s

Set variable. The SHVNAMA/SHVNAML adlen describes the name of the variable to be set, and SHVVALA/SHVVALL describes the value to be assigned to it. The name is validated to ensure that it does not contain incorrect characters, and the variable is then set from the value given. If the name is a stem, all variables with that stem are set, just as though this was a REXX assignment. SHVNEWV is set if the variable did not exist before the operation.

Note:

1. EXEC 2 supports only the S (Set) and F (Fetch) functions. Other requests are rejected.

2. The interface is enabled only during the execution of commands (including CMS subcommands) and external routines (functions and subroutines). An attempt to call the EXECCOMM entry point asynchronously results in a return code of -1 (Invalid entry conditions).

3. While the EXECCOMM request is being serviced, interrupts are enabled for most of the time.

Using Routines from the Callable Services Library

When REXX calls another program that is written in another programming language (Assembler, OS/VS COBOL, VS FORTRAN, VS Pascal, PL/I, C), that program can access and manipulate the current generation of REXX variables by using routines that reside in z/VM's supplied callable services library. The following list describes these CSL routines:

- DMSCDR—causes REXX to drop a REXX variable or group of variables.

- DMSCGR—gets the value of a variable known to an active REXX procedure.

- DMSCGS—gets special REXX values.

- DMSCGX—gets the names and values of all variable known to an active REXX procedure one at a time.

- DMSCSR—sets the value of a variable for an active REXX procedure.

These CSL routines use the EXECCOMM interface described earlier in this section. See *z/VM: CMS Callable Services Reference* for more information about coding these CSL routines.

Example:

The following example shows a REXX exec named TEST invoking a VS FORTRAN program named GETNXT. Once called, GETNXT calls the CSL routine DMSCGX to get the value of all the REXX variables from the TEST EXEC and then displays those values.

```
/* This is a sample REXX exec that sets some variables and
   then calls a FORTRAN program called GETNXT          */
A    = 12
B.1  = 0.5
C    = 3.5E6
D.   = -2
D.1  = 5
D.2  = -4
E    = '123456789ABCDEF'
LAST_GET_NEXT_VAR = 'CHAR STRING'
'LOAD GETNXT'
'START'
```

Figure 8: TEST EXEC

```
C  This is the VS FORTRAN program GETNXT to get the values of all
C  REXX variables from the TEST EXEC, store them in an array,
C  and then display the variables with their values.
C  GETNXT calls the CSL routine 'DMSCGX' to get the values.
C
       PROGRAM GETNXT
C
C DMSCSL - external interface routine to call CSL routine
       EXTERNAL       DMSCSL
C
C Declare all parameters for the CSL call.
C This accommodates 20 variables with names + values up to 25 characters
       INTEGER        RTCODE,VARLEN,BUFLEN,ACVLEN,ACBLEN
       CHARACTER*25   VARNAM(20)
       CHARACTER*25   BUFFER(20)
C
C Input length of buffer and variable length for all variables
       BUFLEN = 25
       VARLEN = 25
C
C Initialize the return code
       RTCODE = 0
       J = 20
C
C Keep getting the next variable until they are all depleted
C      (RC=206) or until you get 20 variables.
       DO 10 I = 1, J
C
C          Initialize the next variable and value
           VARNAM(I) = ' '
           BUFFER(I) = ' '
C
C          Make the call to 'DMSCGX'
         CALL DMSCSL('DMSCGX  ',RTCODE,VARNAM(I),VARLEN,BUFFER(I),
     1                BUFLEN,ACVLEN,ACBLEN)
C
C          Display results
           IF (RTCODE .EQ. 206) THEN
               WRITE (6,31) ' RTCODE = ',RTCODE
               GO TO 40
               END IF
           WRITE (6,30) ' ',VARNAM(I), ' = ',BUFFER(I)
   10      CONTINUE
   40      CONTINUE
   30      FORMAT (A1,A25,A3,A25)
   31      FORMAT (A10, I4)
           END
```

Figure 9: VS FORTRAN Program—GETNXT FORTRAN

After executing the TEST EXEC, here is the output that is displayed at your terminal:

```
DMSLIO740I Execution begins...
A                          = 12
E                          = 123456789ABCDEF
```

```
C                          = 3.5E6
RC                         = 0
D.2                        = -4
D.1                        = 5
LAST_GET_NEXT_VAR          = CHAR STRING
B.1                        = 0.5
RTCODE =  206
```

REXX Exits

This set of exits to the z/VM REXX/VM Interpreter allows applications to tailor the REXX environment. The exits fall into two categories:

Initialization/Termination
    routines called at startup and termination of a program

System services
    routines called to provide host environment services to the language processor.

These exits are provided in both CMS and GCS. For the most part, the interfaces are identical between the two. The following description focuses on CMS, with the GCS differences noted.

Invocation of the Language Processor by an Application Program

The exits are defined at language processor invocation by means of a specified FBLOCK extension. See "The File Block" on page 188 for a description of the format of the file block. The FBLOCK extension contains a pointer in the seventh fullword of the extension block that points to the exit. The eighth fullword of the extension block passes a user word value that is copied from the FBLOCK to the exit routines parameter list (RXIUSER field).

The exit vector is a list of doubleword tokens, with a doubleword fence signaling the end of the list. Each token consists of a code in the first halfword identifying an exit and the address in the second fullword indicating the address of the exit. The second halfword of the doubleword token is reserved.

You can specify the following exits in the list. The RXITDEF macro establishes the equated values for each of these exit names and for their associated subfunctions.

RXFNC
    Process external functions

RXCMD
    Process host commands

RXMSQ
    Manipulate session queue

RXSIO
    Session I/O

RXMEM
    Memory services

RXHLT
    Halt processing

RXTRC
    Test external trace indicator

RXINI
    Initialization processing

RXTER
    Termination processing.

Invocation of the Exits by the Language Processor

This section explains the three parts to the exits:

- How invocation is handled
- The return conditions
- Exit definitions.

Call Conditions

Note: If a GCS problem state application calls the language processor, then the exit routines are entered in a problem state, enabled for interrupts, and enabled with the storage key of the original application program. If a GCS supervisor state application calls the language processor, then the exits are entered in a supervisor state, key zero, enabled for interrupts. If a CMS application calls the language processor, then the exits are called in supervisor state, nucleus key, and enabled for interrupts. In an XA or XC virtual machine, the exit will be called in AMODE 31.

The following registers are defined on entry:

Register 1
 A pointer to the exit parameter list. This parameter list varies with each entered exit. Details on the format of this parameter list for each exit are described later.

Register 13
 A pointer to a 20 fullword register save area.

Register 14
 The return address.

Register 15
 The entry point address.

The exit parameter list consists of several fields that all the exits commonly use, followed by fields that are specific to each exit. The common information includes:

- RXIEXIT - the exit being called
- RXISUBFN - the subfunction requested for that exit
- RXIUSER - the optional fullword of user data
- RXICFLAG - a flag byte used to control exit processing
- RXIFFLAG - a flag byte used for exit specific communication
- RXIPLEN - the length of the parameter list.

See "Exit Definitions" on page 196 for the format of the control blocks.

Return Conditions

On return from the exit, register 15 contains the exit return code and the parameter list is updated with the appropriate results. The return code in register 15 signals one of three actions:

RC=0
 Successful handling of the service. The parameter list has been updated as appropriate for that exit.

RC=1
 Exit chooses not to handle the service request. The language processor handles the request by the default means.

RC=-1
 An irrecoverable error occurred during processing of this request. REXX error 48 (`Failure in system service`) is raised.

The exit routines must save registers 0-14 upon invocation and restore them before returning to their caller.

Exit Definitions

The RXITDEF macro establishes the equated values for each of the exit names and for their associated subfunction names. The RXITPARM macro establishes the mapping DSECT for these parameter lists. The EPLIST and EVALBLOK mapping is further described in "Using the Extended Parameter List" on page 184.

Also, you can use the EPLIST and the EVALBLOK macros to provide the mapping for each of these DSECTS.

RXFNC

Process external functions.

RXFNCCAL

Call an external function

```
RXIEXIT   DS   H       Exit code = 2
RXISUBFN  DS   H       Exit subfunction = 1
RXIUSER   DS   F       User word
RXICFLAG  DS   X       Exit processing control flags
RXIFFLAG  DS   X       Exit specific flags
RXFFERR   EQU  X'80'   Incorrect call to routine
RXFFNFND  EQU  X'40'   Routine not found
RXFFSUB   EQU  X'20'   Subroutine call
RXIPLEN   DS   H       Length of PLIST in bytes
RXIRESRV  DS   F       Reserved for IBM use
RXFFNC    DS   A       Pointer to the routine name
RXFFNCL   DS   F       Length of the routine name
RXFARGS   DS   A       Pointer to argument list
RXFRET    DS   A       Pointer to EVALBLOK for
*                      function RETURN result
```

On entry to the exit, the fields RXFFNC and RXFFNCL define the name of the called function. The field RXFARGS points to the arguments to the function. The flag RXFFSUB is on if the routine is called by means of a CALL rather than as a function.

On return from the exit, values in RXIFFLAG indicate the status of the function processing. If neither RXFFERR nor RXFFNFND is on, then the routine has been successfully called and has run successfully. The field RXFRET may have the address of an EVALBLOK containing the returned result. The flag RXFFERR indicates that the parameters supplied to the routine are somehow incorrect. The language processor returns error 40, `Incorrect call to routine`.

The flag RXFFNFND in RXIFFLAG indicates that the exit could not locate the routine with the given name. The language processor returns error 43, `Routine not found`. If the routine is called as a function and a result is not returned, then the language processor returns error 44, `Function did not return data`. If the routine is called as a subroutine, then the returned result is optional.

The exit allocates the EVALBLOK containing the result, and the language processor returns the storage. The maximum size for an EVALBLOK is 16MB.

Note: The EXECCOMM interface is enabled during calls to the RXFNC exits.

RXCMD

Process host commands.

RXCMDHST

Call a host command.

```
RXIEXIT   DS   H     Exit code = 3
RXISUBFN  DS   H     Exit subfunction = 1
RXIUSER   DS   F     User word
RXICFLAG  DS   X     Exit processing control flags
RXIFFLAG  DS   X     Exit specific flags
RXCFFAIL  EQU  X'80' Command FAILURE occurred
*                    (trappable with SIGNAL ON
                     or CALL ON FAILURE)
RXCFERR   EQU  X'40' Command ERROR occurred
*                    (trappable with SIGNAL ON
                     or CALL ON ERROR)
RXIFEVAL  EQU  X'01' Return code returned in EVALBLOK
RXIPLEN   DS   H     Length of PLIST in bytes
RXIRESRV  DS   F     Reserved for IBM use
RXCADDR   DS   CL8   Current ADDRESS setting
RXCCMD    DS   A     Pointer to the command
RXCCMDL   DS   F     Length of the command
RXCRETC   DS   A     Pointer to return code buffer
RXCRETCL  DS   F     Length of return code buffer
```

On entry to the exit, the fields RXCRETC and RXCRETCL define a buffer that contains a value used for the return code in character format (that is, a numeric return code formatted as a character string). The return code may have a nonnumeric value if desired. On return from the exit, RXCRETCL contains the length of the data placed in the buffer that RXCRETC points to.

If the buffer supplied is too small for the returning value, then the value is alternately returned in an EVALBLOK. In this case, the exit supplies an EVALBLOK and stores the address of the block in RXCRETC. The flag RXIFEVAL is then turned on to indicate that an EVALBLOK has been provided. If the value is returned in an EVALBLOK, the language processor releases the EVALBLOK storage. The maximum size of an EVALBLOK is 16MB.

The exit uses the flags RXCFFAIL and RXCFERR to indicate that an ERROR or FAILURE condition has occurred. The exit controls the definition of what constitutes an ERROR or FAILURE of a command. Under the default command processor, a negative return code is a FAILURE condition and a positive return code is an ERROR condition.

When the RXCMD exit is not being used:

- CALL ON ERROR and SIGNAL ON ERROR trap all positive return codes
- CALL ON ERROR and SIGNAL ON ERROR trap all negative return codes if neither CALL ON FAILURE nor SIGNAL ON FAILURE is set
- If set, CALL ON FAILURE and SIGNAL ON FAILURE trap negative return codes.

When the RXCMD exit is being used:

- CALL ON ERROR and SIGNAL ON ERROR trap the RXCFERR flag that the RXCMD exit returns
- CALL ON ERROR and SIGNAL ON ERROR trap the RXCFFAIL flag if neither CALL ON FAILURE nor SIGNAL ON FAILURE is set.
- If set, CALL ON FAILURE and SIGNAL ON FAILURE trap the RXCFFAIL flag.

Note: The EXECCOMM interface is enabled during calls to the RXCMD exits.

RXMSQ

Manipulate session queue (the external data queue used for PUSH, QUEUE, and PULL).

The exit routine must support a number of subfunction codes. (Note that the code must be supported even if the subfunction itself is not supported, for example, RC=1 can be returned to indicate that the exit routine did not handle the request.) The subfunction request code is contained in the field RXISUBFN. The remainder of the parameter list depends on the particular subfunction called. The RXMSQ subfunctions and their parameter lists are:

RXMSQPLL

Pull a line from the session queue.

```
RXIEXIT   DS   H     Exit code = 4
RXISUBFN  DS   H     Exit subfunction = 1
RXIUSER   DS   F     User word
RXICFLAG  DS   X     Exit processing control flags
RXIFFLAG  DS   X     Exit specific flags
RXMFEMPT  EQU  X'40' Queue was empty
RXIFEVAL  EQU  X'01' String returned in EVALBLOK
RXIPLEN   DS   H     Length of PLIST in bytes
RXIRESRV  DS   F     Reserved for IBM use
RXMRETC   DS   A     Pointer to return buffer
RXMRETCL  DS   F     Length of return buffer
```

On entry to the exit, the fields RXMRETC and RXMRETCL define a buffer that returns a value for the line removed from the session queue. If the buffer supplied is too small, then the value is returned in an EVALBLOK. In this case, the exit supplies an EVALBLOK and stores the address of the block in RXMRETC. The flag RXIFEVAL is then turned on to indicate that an EVALBLOK has been provided. If the value is returned in an EVALBLOK, the language processor releases the EVALBLOK storage. The maximum size of an EVALBLOK is 16MB.

Although the CMS and GCS program stacks are limited to 255 bytes, the session queue that the RXMSQ exits provide has no such limitation.

On return from the exit, the RXMFEMPT flag indicates that there is no data on the queue, and no data has been returned. The contents of the buffer should be ignored.

RXMSQPSH
Place a line on the session queue.

```
RXIEXIT   DS   H      Exit code = 4
RXISUBFN  DS   H      Exit subfunction = 2
RXIUSER   DS   F      User word
RXICFLAG  DS   X      Exit processing control flags
RXIFFLAG  DS   X      Exit specific flags
RXMFLIFO  EQU  X'80'  Stack the line LIFO
RXIPLEN   DS   H      Length of PLIST in bytes
RXIRESRV  DS   F      Reserved for IBM use
RXMVAL    DS   A      Pointer to line to stack
RXMVALL   DS   F      Length of line to stack
```

The line placed on the queue is the result of evaluating the expression specified on a PUSH or QUEUE instruction. This string can be any length up to 16MB. The exit truncates this string if the exit has a restriction on the maximum width of the queue. The flag RXMFLIFO indicates the stacking order.

RXMSQSIZ
Return the number of lines in the session queue.

```
RXIEXIT   DS   H      Exit code = 4
RXISUBFN  DS   H      Exit subfunction = 3
RXIUSER   DS   F      User word
RXICFLAG  DS   X      Exit processing control flags
RXIFFLAG  DS   X      Exit specific flags
RXIPLEN   DS   H      Length of PLIST in bytes
RXIRESRV  DS   F      Reserved for IBM use
RXMQSIZE  DS   F      Number of lines in stack
```

On return from the exit, RXMQSIZE contains the size of the data queue as a 32-bit unsigned number.

RXSIO
Session I/O.

Note: The EXTERNALS built-in function always returns a value of zero when the RXSIO exit has been specified.

The exit routine must support a number of subfunction codes. (Note that the code must be supported even if the subfunction itself is not supported, for example, RC=1 can be returned to indicate that the exit routine did not handle the request.) The subfunction request code is contained in the field RXISUBFN. The remainder of the parameter list depends on the particular subfunction called. The RXSIO subfunctions and their parameter lists are:

RXSIOSAY
Write a line to the character output stream. Called for SAY instruction to display output.

```
RXIEXIT   DS   H      Exit code = 5
RXISUBFN  DS   H      Exit subfunction = 1
RXIUSER   DS   F      User word
RXICFLAG  DS   X      Exit processing control flags
RXIFFLAG  DS   X      Exit specific flags
RXIPLEN   DS   H      Length of PLIST in bytes
RXIRESRV  DS   F      Reserved for IBM use
RXSVAL    DS   A      Pointer to line to display
RXSVALL   DS   F      Length of line to display
```

The line displayed at the terminal results from evaluating the expression specified on a SAY instruction. This string can be any length up to the size of the terminal (either by default system processing or by a call to RXSIOTLL). The exit truncates this string if the string is too long.

RXSIOTRC

TRACE output processing. Call to output TRACE results.

```
RXIEXIT   DS   H    Exit code = 5
RXISUBFN  DS   H    Exit subfunction = 2
RXIUSER   DS   F    User word
RXICFLAG  DS   X    Exit processing control flags
RXIFFLAG  DS   X    Exit specific flags
RXIPLEN   DS   H    Length of PLIST in bytes
RXIRESRV  DS   F    Reserved for IBM use
RXSVAL    DS   A    Pointer to line to display
RXSVALL   DS   F    Length of line to display
```

The line to be displayed at the terminal is the result of a traced line. This string may be any length up to the size of the terminal (as determined by default system processing or by a call to RXSIOTLL). The exit truncates this string if the string is too long.

RXSIOTRD

Read from character input stream.

```
RXIEXIT   DS   H      Exit code = 5
RXISUBFN  DS   H      Exit subfunction = 3
RXIUSER   DS   F      User word
RXICFLAG  DS   X      Exit processing control flags
RXIFFLAG  DS   X      Exit specific flags
RXIFEVAL  EQU  X'01'  String returned in EVALBLOK
RXIPLEN   DS   H      Length of PLIST in bytes
RXIRESRV  DS   F      Reserved for IBM use
RXSRETC   DS   A      Pointer to return buffer
RXSRETCL  DS   F      Length of return buffer
```

On entry to the exit, the fields RXSRETC and RXSRETCL define a buffer that can contain a value used for the line read from the terminal. If the buffer supplied is too small for the value to be returned, then the value may alternately be returned in an EVALBLOK. In this case, the exit supplies an EVALBLOK and stores the address of the block in RXSRETC. The flag RXIFEVAL is then turned on to indicate that an EVALBLOK has been provided. If the value is returned in an EVALBLOK, the language processor releases the EVALBLOK storage. The maximum size of an EVALBLOK is 16MB.

RXSIODTR

Debug read from character input stream.

```
RXIEXIT   DS   H    Exit code = 5
RXISUBFN  DS   H    Exit subfunction = 4
RXIUSER   DS   F    User word
RXICFLAG  DS   X    Exit processing control flags
RXIFFLAG  DS   X    Exit specific flags
RXIPLEN   DS   H    Length of PLIST in bytes
RXIRESRV  DS   F    Reserved for IBM use
RXSRETC   DS   A    Pointer to return buffer
RXSRETCL  DS   F    Length of return buffer
```

On entry to the exit, the fields RXSRETC and RXSRETCL define a buffer that returns a value for the line read from the terminal. If the buffer supplied is too small, then a return code of -1 is returned.

RXSIOTLL

Return maximum line length in bytes.

```
RXIEXIT   DS   H    Exit code = 5
RXISUBFN  DS   H    Exit subfunction = 5
RXIUSER   DS   F    User word
RXICFLAG  DS   X    Exit processing control flags
RXIFFLAG  DS   X    Exit specific flags
RXIPLEN   DS   H    Length of PLIST in bytes
RXIRESRV  DS   F    Reserved for IBM use
RXSSIZE   DS   F    Size of terminal in bytes
```

On return from the exit, RXSSIZE contains the width of the terminal as a 32-bit unsigned number.

The LINESIZE built-in function uses this value, and it breaks up lines created by SAY and TRACE. The RXSIOSAY and RXSIOTRC functions must be capable of handling lines of this length.

RXMEM

Memory services.

The exit routine must support a number of subfunction codes. (Note that the code must be supported even if the subfunction itself is not supported, for example, RC=1 can be returned to indicate that the exit routine did not handle the request.) The field RXISUBFN contains the subfunction request code. The remainder of the parameter list depends on the particular subfunction called. The RXMEM subfunctions and their parameter lists are:

RXMEMGET

Allocate memory.

```
RXIEXIT   DS   H     Exit code = 6
RXISUBFN  DS   H     Exit subfunction = 1
RXIUSER   DS   F     User word
RXICFLAG  DS   X     Exit processing control flags
RXIFFLAG  DS   X     Exit specific flags
RXMFLO24  EQU  X'80' Storage must be allocated
*                    below the 16MB line.
RXIPLEN   DS   H     Length of PLIST in bytes
RXIRESRV  DS   F     Reserved for IBM use
RXMSSIZE  DS   F     Size of storage to be
*                    allocated (in doublewords)
RXMADDR   DS   A     Address of allocated storage
```

On entry to the exit, RXMSSIZE contains the size of the block of storage to be allocated. On exit, RXMADDR should contain the address of the allocated storage. Out-of-storage conditions are reflected by setting RXMADDR to zero on return from the exit. The flag RXMFLO24 indicates that the storage must be allocated below the 16MB line.

RXMEMRET

Deallocate memory.

```
RXIEXIT   DS   H     Exit code = 6
RXISUBFN  DS   H     Exit subfunction = 2
RXIUSER   DS   F     User word
RXICFLAG  DS   X     Exit processing control flags
RXIFFLAG  DS   X     Exit specific flags
RXIPLEN   DS   H     Length of PLIST in bytes
RXIRESRV  DS   F     Reserved for IBM use
RXMSSIZE  DS   F     Size of storage to be
*                    released (in doublewords)
RXMADDR   DS   A     Address of storage to be
*                    released
```

On entry, the fields RXMSSIZE and RXMADDR contain the length and address of the storage to be released.

Note: Because calls to external functions and other exits can result in the language processor's obtaining blocks of storage that were not allocated by calls to the RXMEMGET exit, the RXMEMRET exit should be prepared to handle these conditions. If desired, a return code of 1 can be used to cause a block of storage to be released by usual system means.

Note: Because the memory services provided by the RXMEM exits cannot support releasing storage in increments other than those allocated, the language processor does not release partial storage if the RXMEM exit has been specified.

RXHLT

Halt processing.

The exit routine must support a number of subfunction codes. (Note that the code must be supported even if the subfunction itself is not supported, for example, RC=1 can be returned to indicate that the exit routine did not handle the request.) The subfunction request code is contained in the field RXISUBFN. The remainder of the parameter list depends on the particular subfunction called.

When the RXHLT exit is not being used, the CMS immediate command HI (Halt Interpretation) causes a halt condition. If the RXHLT exit is being used, a halt condition is recognized when the exit routine sets the RXHFHALT flag.

The RXHLT subfunctions and their parameter lists are:

RXHLTCLR

Clear Halt indicator.

```
RXIEXIT   DS   H      Exit code = 7
RXISUBFN  DS   H      Exit subfunction = 1
RXIUSER   DS   F      User word
RXICFLAG  DS   X      Exit processing control flags
RXIFFLAG  DS   X      Exit specific flags
RXIPLEN   DS   H      Length of PLIST in bytes
RXIRESRV  DS   F      Reserved for IBM use
```

This exit has no inputs or outputs. It signals the exit that handles HALT processing that the condition has been recognized and should be cleared.

RXHLTTST

Test Halt indicator.

```
RXIEXIT   DS   H      Exit code = 7
RXISUBFN  DS   H      Exit subfunction = 2
RXIUSER   DS   F      User word
RXICFLAG  DS   X      Exit processing control flags
RXIFFLAG  DS   X      Exit specific flags
RXHFHALT  EQU  X'80' HALT condition occurred
RXIPLEN   DS   H      Length of PLIST in bytes
RXIRESRV  DS   F      Reserved for IBM use
RXHSTR    DS   A      Pointer to EVALBLOK
*                     containing an optional
*                     HALT string
```

On return from this exit, RXHFHALT indicates whether a HALT condition has occurred. The exit can also return a string that is available as CONDITION('Description') for a CALL ON HALT or SIGNAL ON HALT condition trap. This string is returned in an EVALBLOK. In this case, the exit supplies an EVALBLOK and stores the address of the block in RXHSTR. If the value is returned in an EVALBLOK, the language processor releases the EVALBLOK storage. The maximum size of an EVALBLOK is 16MB.

RXTRC

Trace services.

RXTRCTST

Test external trace indicator.

```
RXIEXIT   DS   H      Exit code = 8
RXISUBFN  DS   H      Exit subfunction = 1
RXIUSER   DS   F      User word
RXICFLAG  DS   X      Exit processing control flags
RXIFFLAG  DS   X      Exit specific flags
RXTFTRAC  EQU  X'80' External TRACE setting
RXIPLEN   DS   H      Length of PLIST in bytes
RXIRESRV  DS   F      Reserved for IBM use
```

On return from this exit, RXTFTRAC indicates whether an external trace condition has occurred.

RXINI

Initialization processing.

RXINIEXT

Perform external initialization.

```
RXIEXIT   DS   H      Exit code = 9
RXISUBFN  DS   H      Exit subfunction = 1
RXIUSER   DS   F      User word
RXICFLAG  DS   X      Exit processing control flags
RXIFFLAG  DS   X      Exit specific flags
```

```
RXIPLEN   DS   H     Length of PLIST in bytes
RXIRESRV  DS   F     Reserved for IBM use
```

This exit has no inputs or outputs. It is called before the first instruction of the program is interpreted. The EXECCOMM interface is enabled when this exit is called.

There are more possibilities for initialization that are not available on GCS. See "Additional Exit Provided in VM" on page 204 for more information.

RXTER
Termination processing.

RXTEREXT
Perform External Termination.

```
RXIEXIT   DS   H     Exit code = 10
RXISUBFN  DS   H     Exit subfunction = 1
RXIUSER   DS   F     User word
RXICFLAG  DS   X     Exit processing control flags
RXIFFLAG  DS   X     Exit specific flags
RXIPLEN   DS   H     Length of PLIST in bytes
RXIRESRV  DS   F     Reserved for IBM use
```

This exit has no inputs or outputs. It is called after the last instruction of the program is interpreted. The EXECCOMM interface is enabled when this exit is called.

There are more possibilities for termination that are not available on GCS. See "Additional Exit Provided in VM" on page 204 for more information.

Usage Notes

When using these exits to customize the REXX environment, you may need to consider the following limitations and restrictions:

1. When an application program calls the language processor, the exit vector could contain codes that are not defined or are reserved for IBM use. These are ignored.

2. The exit vector could contain the same code more than once. When obtaining the storage for the exit vector, the first occurrence of the RXMEM exit is used. The first occurrence of the REXMEM exit is also used to release the storage for the exit vector. In all other cases, including the RXMEM exit, the last occurrence of an exit code is the one used.

3. Upon return to the language processor from one of the exits, the return code might be a nonzero return code other than the documented (+)1 or -1. All negative return codes are treated the same as a -1 and all positive return codes are treated the same as a (+)1.

4. The EXTERNALS built-in function always returns a value of zero when the RXSIO exit has been specified.

5. The RXMEM exit is not used to obtain or release storage when called on GCS under any of these three circumstances:

   - The REXX language processor has been called through SVC 202 from a problem state program with exits specified. Storage is obtained for a register save area, and then the SYNCH macro switches to problem state. This area is retained throughout the processing of the EXEC but is released just before the language processor returns control back to the SVC handler to return control to the calling program.

   - The EXECCOMM routine has been called through SVC 203 from a problem state program with exits specified. Storage is obtained for a register save area, and then the SYNCH macro switches to problem state. This area is retained throughout the processing of the EXECCOMM request, but is released just before EXECCOMM returns back to the SVC handler to return control to the calling program.

   - The RXMEM exit is being called to obtain storage for the main work area. Temporary storage is obtained for the RXMEM parameter list, and this is released when the work area has been obtained. The same sequence is used when calling RXMEM to release the storage the work area occupies.

> **Note:** Remember that the FIRST occurrence of the RXMEM exit found in the exit list obtains and release the storage for the work area. The LAST occurrence of the RXMEM exit found in the exit list is used for all other storage management requests.

Other information that you may need to be aware of follows.

1. The RXINI, RXTER, RXCMD, and RXFNC exits may use the EXECCOMM interface. EXECCOMM gives back a return code of -1 when it is called from any other exit routine.

2. The descriptions of some of the exits state that a result may be returned in an EVALBLOK "if the supplied buffer is too small." In fact, the result may be returned through an EVALBLOK regardless of the size of the supplied buffer. It MUST be returned through an EVALBLOK if the supplied buffer is too small, providing the routine ends with a zero return code.

3. The actual behavior of the RXTRCTST and RXHLT exits is that they are called one time before the first clause is run, and then one time after each clause is run. The intent of these exits is for them to be called on a clause boundary between clauses.

Additional Exit Provided in VM

An additional exit function to the z/VM REXX/VM interpreter is provided and is available only to CMS. The REXEXIT macro is used in an application program to create and maintain a list of user specified global exits for REXX programs.

Global exits allow applications running on VM to further tailor the REXX environment. System exits are defined and called for just one exec. In contrast, global exits are called for all REXX execs and remain defined until explicitly cleared or an IPL or ABEND occurs.

Global exits are defined by using the REXEXIT macro. A global exit can be defined specifying INIT=YES to indicate that it should be called during REXX initialization. Specifying TERM=YES indicates that it should be called during REXX termination processing.

The REXEXIT macro has three main parameters: SET, CLR, and QUERY. The SET parameter declares the named entry point as a user exit. NAME is used with SET to name the exit routine to be defined. In addition, ENTRY is used with SET to define the entry point of the exit routine.

The CLR parameter deletes the named user exit from the list of exits. The QUERY parameter queries the named global exit. NAME is also used with CLR and QUERY to name the exit routine to be cleared or queried.

For a more complete description of the REXEXIT macro, see *z/VM: CMS Macros and Functions Reference*.

Chapter 9. Debug Aids

In addition to the TRACE instruction ("TRACE" on page 61), there are the following debug aids.

- The interactive debug facility
- The CMS immediate commands:
  - HI – Halt Interpretation
  - TS – Trace Start
  - TE – Trace End
- The CMS HELP command.

Interactive Debugging of Programs

The debug facility permits interactively controlled execution of a program.

Changing the TRACE action to one with a prefix **?** (for example, TRACE ?A or the TRACE built-in function) turns on interactive debug and indicates to the user that interactive debug is active. Further TRACE instructions in the program are ignored, and the language processor pauses after nearly all instructions that are traced at the console (see the following for the exceptions). When the language processor pauses, indicated by a VM READ or unlocking of the keyboard, three debug actions are available:

1. **Entering a null line** (with no characters, even blanks) makes the language processor continue execution until the next pause for debug input. Repeatedly entering a null line, therefore, steps from pause point to pause point. For TRACE ?A, for example, this is equivalent to single-stepping through the program.

2. **Entering an equal sign (=)** with no blanks makes the language processor rerun the clause last traced. For example: if an IF clause is about to take the wrong branch, you can change the value of the variable(s) on which it depends, and then rerun it.

   Once the clause has been rerun, 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). If an instruction has a syntax error in it, a standard message is displayed and you are prompted for input again. Similarly all the other SIGNAL conditions are disabled while the string is processed to prevent unintentional transfer of control.

   During execution of the string, no tracing takes place, except that nonzero return codes from host commands are displayed. Host commands are always run (that is, 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 a TRACE instruction was entered. In this latter case, the language processor immediately alters the tracing action (if necessary) and then continues executing until the next pause point (if any). To alter the tracing action (from All to Results, for example) and then rerun the instruction, you must use the built-in function TRACE (see the TRACE function). For example, CALL TRACE I changes the trace action to **I** and allows re-execution of the statement after which the pause was made. Interactive debug is turned off, when it is in effect, if a TRACE instruction uses a prefix, or at any time when a TRACE 0 or TRACE with no options is entered.

   You can use the numeric form of the TRACE instruction to allow sections of the program to be run without pause for debug input. TRACE n (that is, positive result) allows execution to continue, skipping the next n pauses (when interactive debug is or becomes active). TRACE -n (that is, negative result) allows execution to continue without pause and with tracing inhibited for n clauses that would otherwise be traced.

The trace action selected by a TRACE instruction is saved and restored across subroutine calls. This means that if you are stepping through a program (say after using TRACE ?R to trace Results) and then enter a subroutine in which you have no interest, you can enter TRACE 0 to turn tracing off. No further instructions in the subroutine are traced, but on return to the caller, tracing is restored.

Similarly, if you are interested only in a subroutine, you can put a TRACE ?R instruction at its start. Having traced the routine, the original status of tracing is restored and (if tracing was off on entry to the subroutine) tracing (and interactive debug) is turned off until the next entry to the subroutine.

You can switch tracing on (without modifying a program) by using the command SET EXECTRAC ON. You can also turn tracing on or off asynchronously (that is, while a program is running) by using the TS and TE immediate commands. See "Interrupting Execution and Controlling Tracing" on page 206 for the description of these facilities.

Since any instructions may be run in interactive debug, you have considerable control over execution.

Some examples:

```
Say expr      /* displays the result of evaluating the    */
              /* expression.                               */

name=expr     /* alters the value of a variable.           */

Trace 0       /* (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          /* stops execution of the program.           */

Do i=1 to 10; say stem.i; end
              /* displays ten elements of the array stem.       */
```

Note that while in interactive debug, pauses may occur because of PULL statements as well as because of interactive debug. For programs containing PULL statements, it is important to be aware of the reason for each pause. In programs, PULL statements are often paired with SAY statements. The user should enter the data for the PULL at the pause after the trace line for the PULL (the pause specifically for entering data for the PULL). The user should not enter the data at the pause after the corresponding SAY statement (this is an interactive debug pause). **Exceptions**: Some clauses cannot safely be re-run, and, therefore, the language processor does not pause after them, even if they are traced. These are:

- Any repetitive DO clause, on the second or subsequent time around the loop.
- All END clauses (not a useful place to pause in any case).
- All THEN, ELSE, OTHERWISE, or null clauses.
- All RETURN and EXIT clauses.
- All SIGNAL and CALL clauses (the language processor pauses after the target label has been traced).
- Any clause that raises a condition that CALL ON or SIGNAL ON traps (the pause takes place after the target label for the CALL or SIGNAL has been traced).
- Any clause that causes a syntax error. (These may be trapped by SIGNAL ON SYNTAX, but cannot be re-run.)

Interrupting Execution and Controlling Tracing

You can interrupt the language processor during execution in several ways:

- The HI (Halt Interpretation) immediate command causes all currently executing REXX programs to stop, as though there has been a syntax error. This is especially useful when an editor macro gets into a loop, and it is desirable to halt it without destroying the whole environment (as HX, Halt Execution, would do). When an HI interrupt causes a REXX program to stop, the program stack is cleared. An HI interrupt may be trapped by using SIGNAL ON HALT ("SIGNAL" on page 59).

- The TS (Trace Start) immediate command turns on the external tracing bit. If the bit is not already on, TS puts the program into usual interactive debug and you can then run REXX instructions and so forth as usual (for example, to display variables, EXIT, and so forth). This too is useful when you suspect that a REXX program is looping—you can enter TS, and inspect and step the program before deciding whether to allow the program to continue or not.

- The TE (Trace End) immediate command turns off the external tracing bit. If it is not already off, this has the effect of executing a TRACE O instruction. This can be useful to stop tracing when not in interactive debug (as when tracing was started by issuing SET EXECTRAC ON and interactive debug was subsequently stopped by issuing TRACE ?).

The System External Trace Bit:

Before each clause is run, an external trace bit, owned by CMS, is inspected. You can turn the bit on with the TS immediate command, and turn it off with the TE immediate command. You can also alter the bit by using the SET EXECTRAC command (described later). CMS itself never alters this bit, except that it is cleared on return to CMS command level.

The language processor maintains an internal *shadow* of the external bit, which therefore allows it to detect when the external bit changes from a 0 to a 1, or vice-versa. If the language processor sees the bit change from 0 to 1, **?** (interactive debug) is forced on and the tracing action is forced to R if it is A, C, E, F, L, N, or O. The tracing action is left unchanged if it is I, R, or S.

Similarly, if the shadow bit changes from 1 to 0, all tracing is forced off. This means that tracing may be controlled externally to the REXX program: you can switch on interactive debug at any time without modifying the program. The TE command can be useful if a program is tracing clauses without being in interactive debug (that is, after SET EXECTRAC ON, TRACE ? was issued). You can use TE to switch off the tracing without affecting any other output from the program.

If the external bit is on upon entry to a REXX program, the SOURCE string is traced (see "PARSE" on page 48) and interactive debug is switched on as usual—hence with use of the system trace bit, tracing of a program and all programs called from it, can be easily controlled.

The internal *shadow* bit is saved and restored across internal routine calls. This means that (as with internally controlled tracing) it is possible to turn tracing on or off locally within a subroutine. It also means that if a TS interrupt occurs during execution of a subroutine, tracing is also switched on upon RETURN to the caller.

You can use the CMSFLAG(EXECTRAC) function and the command QUERY EXECTRAC to test the setting of the system trace bit.

The command SET EXECTRAC ON turns on the trace bit. Using it before invoking a REXX program causes the program to be entered with debug tracing immediately active. If issued from inside a program, SET EXECTRAC ON has the same effect as TRACE ?R (unless TRACE I or S is in effect), but is more global in that all programs called are traced, too. The command SET EXECTRAC OFF turns the trace bit off. Issuing this when the bit is on is equivalent to the instruction TRACE O, except that it has a system (global) effect.

Note: SET EXECTRAC OFF turns off the system trace bit at any time; for example, if it has been set by a TS immediate command issued while not in a REXX program.

Chapter 10. Reserved Keywords and Special Variables

You can use keywords as ordinary symbols in many situations where there is no ambiguity. The precise rules are given here.

There are three special variables: RC, RESULT, and SIGL.

Reserved Keywords

The free 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 referred to as keywords. Examples of REXX keywords are the WHILE in a DO instruction and the THEN (which acts as a clause terminator in this case) following an IF or WHEN clause.

Apart from these cases, only simple symbols that are the first token in a clause and that are not followed by an = or : are checked to see if they are instruction keywords. You can use the symbols freely elsewhere in clauses without their being taken as keywords.

It is not, however, recommended for users to run host commands or subcommands with the same name as REXX keywords (QUEUE, for example). This can create problems for any programmer whose REXX programs might be used for some time and in circumstances outside his or her control, and who wishes to make the program absolutely *watertight*.

In this case, a REXX program may be written with (at least) the first words in command lines enclosed in quotation marks.

Example:

```
'ERASE' filename filetype filemode
```

This also has the advantage of being more efficient, and, with this style, you can use the SIGNAL ON NOVALUE condition to check the integrity of an exec.

An alternative strategy is to precede such command strings with two adjacent quotation marks, which concatenates the null string on to the front.

Example:

```
''Erase filename filetype filemode
```

A third option is to enclose the entire expression (or the first symbol) in parentheses.

Example:

```
(Erase filename filetype filemode)
```

More important, the choice of strategy (if it is to be done at all) is a personal one by the programmer. The REXX language does not impose it.

Special Variables

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

RC

is set to the return code from any run host command (or subcommand). Following the SIGNAL events, SYNTAX, ERROR, and FAILURE, RC is set to the code appropriate to the event: the syntax error number (see appendix on error messages, Appendix A, "Error Numbers and Messages," on page 277) or the command return code. RC is unchanged following a NOVALUE or HALT event.

Note: Host commands run manually from debug mode do not cause the value of RC to change.

RESULT

is set by a RETURN instruction in a subroutine that has been called if the RETURN instruction specifies an expression. If the RETURN instruction has no expression on it, RESULT is dropped (becomes uninitialized.)

SIGL

contains the line number of the clause currently executing when the last transfer of control to a label took place. (A SIGNAL, a CALL, an internal function invocation, or a trapped error condition could cause this.)

None of these variables has an initial value. You can alter them, just as with any other variable, and they may be accessed as described under the "Direct Interface to Current Variables" on page 190. 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 "PARSE" on page 48). The latter consists of the string CMS followed by the call type and then the file name, file type, and file mode of the file being run. These are followed by the name by which the program was called and the initial (default) command environment.

In addition, PARSE VERSION (see "PARSE" on page 48) makes available the version and date of the language processor code that is running. The built-in functions TRACE and ADDRESS return the current trace setting and environment name, respectively.

Finally, you can obtain the current settings of the NUMERIC function by using the DIGITS, FORM, and FUZZ built-in functions.

Chapter 11. Some Useful CMS Commands

There are a number of CMS commands that can be especially useful to REXX programmers. Some can access and change REXX variables.

DESBUF
: Clears the console and program stack input and output buffers. DROPBUF should be used in most cases instead of DESBUF because DESBUF also drops output buffers (which could be undesirable).

DROPBUF
: Eliminates only the most recently created program stack buffer or a specified program stack buffer and all the buffers created after it.

EXECDROP
: Purges storage-resident EXECs.

EXECIO
: Reads and writes CMS files. Issues CP commands, placing the output that would usually appear on the screen in the program stack. Reads from the virtual reader. Writes to the virtual printer and virtual punch.

EXECLOAD
: Loads an exec or XEDIT macro into storage and prepares it for execution.

EXECMAP
: Lists storage-resident execs.

EXECOS
: Cleans up after OS, VSAM and Vector programs, and should be used if more than one OS or VSAM program is called between returns to CMS command level.

EXECSTAT
: Provides the status of a specified exec.

EXECUPDT
: An extension to the UPDATE command, EXECUPDT modifies a REXX program file with one or more update files. The input files must have fixed length, 80-column records. The result is an executable, V-format program file.

FINIS
: Closes one or more files on a disk or in a Shared File System (SFS) directory.

GLOBALV
: Saves exec data (variables) from one invocation to the next.

IDENTIFY
: Displays or stacks user ID, node ID, RSCS ID, date, time, time zone, and day of the week.

LISTFILE
: Lists information about CMS files in accessed directories and on accessed minidisks.

MAKEBUF
: Creates a new buffer within the program stack.

NUCXLOAD
: Installs specific types of modules as nucleus extensions.

NUCXMAP
: Displays or stacks information about currently defined nucleus extensions.

PARSECMD
: Parses and translates an exec's arguments.

PIPE
: Calls CMS Pipelines to process a series of programs or stages. A series of stages is called a pipeline. The PIPE command runs a pipeline. Each stage manipulates or handles data by:

- Using the stages and pipeline subcommands provided by CMS Pipelines. Several of the stages allow you to set and retrieve the value of REXX variables.
- Extending the set of CMS Pipelines stages by writing your own stages in the REXX language.
- User-written stages in the Assembler language can call REXX programs.

PROGMAP
Displays or stacks information about programs currently in storage.

QUERY
See SET in this list. (See also the CMSFLAG function.)

SEGMENT
Manages saved storage by: reserving CMS storage for a saved segment that will reside within a virtual machine, loading or purging a saved segment, or releasing storage previously reserved for a saved segment.

SENTRIES
Determines the number of lines currently in the program stack. When you issue a SENTRIES command, CMS returns the number of lines in the program stack (but not the terminal input buffer) as a return code. The REXX functions QUEUED() and EXTERNALS() can also provide information about the console stack.

SET
ABBREV, IMPEX, IMPCP, INSTSEG modify the search order; CMSTYPE controls output to the screen (including output generated by the SAY instruction); EXECTRAC controls tracing.

XEDIT
When used as an Editor, additional subcommands (macros) may be written in REXX. XEDIT may also be used to write and read menus (full screen displays). In both applications, XEDIT variables may be assigned to REXX variables using the EXTRACT subcommand of XEDIT.

XMITMSG
Retrieves messages from a repository file. These messages can then be displayed.

For more details on these CMS commands, refer to the *z/VM: CMS Commands and Utilities Reference*.

Chapter 12. Invoking Communications Routines

ADDRESS CPICOMM

You can use the ADDRESS CPICOMM statement in your REXX program to call program-to-program communications routines. These communications routines must be part of Common Programming Interface (CPI) Communications.

CPI Communications routines are described in the *Common Programming Interface Communications Reference*.

In z/VM, all communications routines are stored in the VMLIB callable services library.

Here is the format to use when calling a CPI Communications routine from a REXX program:
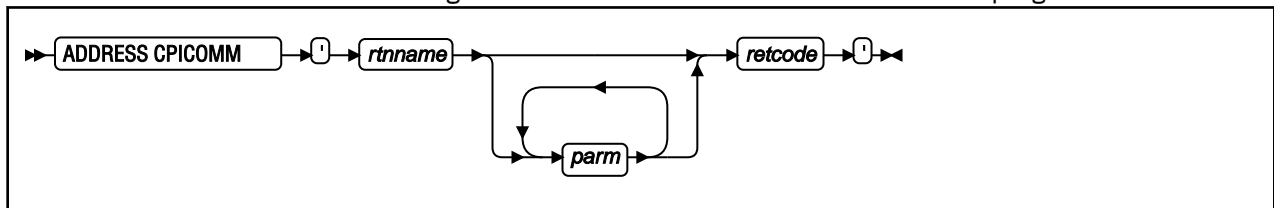
rtnname
> is the name of the CPI Communications routine to be called.

parm
> is the name of one or more parameters to be passed to the CPI Communications routine. The number and type of these parameters are routine-dependent. A parameter being passed must be the name of a variable.
>
> ADDRESS CPICOMM uses the EXECCOMM interface to build a properly-formatted parameter list before completing the call to the routine.

retcode
> is the name of a variable to receive the return code from the CPI Communications routine. The value returned in this variable is always greater than or equal to zero. Return codes are documented for individual CPI Communications routines in the *Common Programming Interface Communications Reference*.

> **Note:**

1. If REXX successfully calls the CPI Communications routine, then the REXX variable *RC* contains a zero return code. See "Return Codes" on page 214 for a list of return code values that can be stored in the REXX variable *RC*. Any value in *retcode* is the return code from the called CPI Communications routine and its value is greater than or equal to zero.

2. If REXX detects an error and is *not* able to successfully call the CPI Communications routine, then the REXX variable *RC* contains a negative return code. See "Return Codes" on page 214 for a list of return code values that can be stored in the REXX variable *RC*. Any value in *retcode* is meaningless because the CPI Communications routine was not successfully called.

Usage Notes

1. Do *not* use ADDRESS CPICOMM to call other types of CSL routines. Instead, use one of the following:

   - To call resource recovery routines, use ADDRESS CPIRR. For more information, see Chapter 13, "Invoking Resource Recovery Routines," on page 215.

- To call OPENVM-type CSL routines (such as OpenExtensions callable services), use ADDRESS OPENVM. For more information, see Chapter 14, "Invoking OPENVM Routines," on page 217.
- To call other CSL routines, use the CSL function. For more information, see "CSL" on page 120.

2. Only character string and signed binary data can be passed to a CPI Communications routine. If a routine's parameter is defined as a signed binary number, the ADDRESS CPICOMM function makes the necessary translations to and from the routine. However, ADDRESS CPICOMM cannot translate a number in exponential notation to signed binary. Use the NUMERIC instruction to ensure that exponential notation is not used.

3. When a CPI Communications routine returns data in a buffer variable, the data is left-justified and may have trailing blanks. You can use the STRIP function of REXX to extract data from the buffer.

4. See *z/VM: CMS Application Development Guide*, which contains scenarios and examples for using ADDRESS CPICOMM in a z/VM environment.

5. When calling Send_Data (CMSEND), the buffer you specify on the call cannot contain more than 32,767 bytes of data. If a buffer has more than 32,767 bytes, you must partition it and pass it in units of 32,767 or fewer bytes.

Return Codes

The list below shows the possible return codes from ADDRESS CPICOMM. The following return code values are in the REXX variable *RC*.

104
Insufficient virtual storage available

-3
Routine does not exist

-7
Routine not loaded

-9
Insufficient virtual storage available (See Chapter 1, "REXX General Concepts," on page 1.)

-10
Too many parameters specified

-11
Not enough parameters specified

-20
Callable Services Library internal error: invalid call

-22
Callable Services Library internal error: parameter list contains more than one argument

-24
REXX internal error: EXECCOMM fetch failure

-25
REXX internal error: EXECCOMM set failure

-26*nnn*
Parameter number *nnn* was too long

-27*nnn*
Parameter number *nnn* has an invalid value for the data type of that parameter

-28*nnn*
Parameter number *nnn* is an invalid REXX variable name

Note: For the last three return codes, the parameters are numbered serially, corresponding to the order in which they are coded. The routine name is always parameter number 001, the next parameter is 002, and so forth. When the return code is between -20 and -28*nnn* inclusive, the error can occur only when using the ADDRESS CPICOMM interface from REXX.

Chapter 13. Invoking Resource Recovery Routines

ADDRESS CPIRR

You can use the ADDRESS CPIRR statement in your REXX program to call CPI Resource Recovery routines.

CPI Resource Recovery routines are described in the *Common Programming Interface Resource Recovery Reference*.

In z/VM, all CPI Resource Recovery routines are stored in the VMLIB callable services library.

Here is a format to use when calling a CPI Resource Recovery routine from a REXX program:

rtnname
> is the name of the CPI Resource Recovery routine to be called. It is either SRRCMIT or SRRBACK. SRRCMIT is the CPI Resource Recovery routine that does a commit. SRRBACK is the CPI Resource Recovery routine that does a rollback. See *Common Programming Interface Resource Recovery Reference* for more information about SRRCMIT and SRRBACK.

retcode
> is the name of a variable to receive the return code from the CPI Resource Recovery routine. The value returned in this variable is always greater than or equal to zero. The *Common Programming Interface Resource Recovery Reference* documents return codes for individual CPI Resource Recovery routines.

> **Note:**

> 1. If REXX successfully calls the CPI Resource Recovery routine, then the REXX variable *RC* contains a zero return code. See "Return Codes" on page 216 for a list of return code values that can be stored in the REXX variable *RC*. Any value in *retcode* is the return code from the called CPI Resource Recovery routine, and its value is greater than or equal to zero.

> 2. If REXX detects an error and is *not* able to successfully call the CPI Resource Recovery routine, then message 1292S is issued and CMS abends with the abend code X'ACB'.

Usage Notes

1. Do *not* use ADDRESS CPIRR to call other types of CSL routines. Instead, use one of the following:

   - To call CPI Communications routines, use ADDRESS CPICOMM. For more information, see Chapter 12, "Invoking Communications Routines," on page 213.

   - To call OPENVM-type CSL routines (such as OpenExtensions callable services), use ADDRESS OPENVM. For more information, see Chapter 14, "Invoking OPENVM Routines," on page 217.

   - To call other CSL routines, use the CSL function. For more information, see "CSL" on page 120.

2. If you issue:

```
ADDRESS CPIRR 'SRRBACK retcode'
```

   and the rollback of resources fails, then CMS abends with the abend code X'ACB'. See *z/VM: CP Messages and Codes* for more information about abend codes.

Return Codes

The list below shows the return code from ADDRESS CPIRR. The following return code value is in the REXX variable *RC*:

0
　　Resource recovery routine was run and control returned to REXX

Chapter 14. Invoking OPENVM Routines

ADDRESS OPENVM

You must use the ADDRESS OPENVM statement in your REXX program to call OPENVM-type CSL routines, such as OpenExtensions callable services. OPENVM routines may not follow the usual CSL conventions, such as providing return and reason codes as the first two parameters.

To determine if a routine is an OPENVM routine, you can use the CMS CSLMAP or CSLLIST command. For information about these commands, see *z/VM: CMS Commands and Utilities Reference*.

Here is the format to use when calling an OPENVM routine from a REXX program:
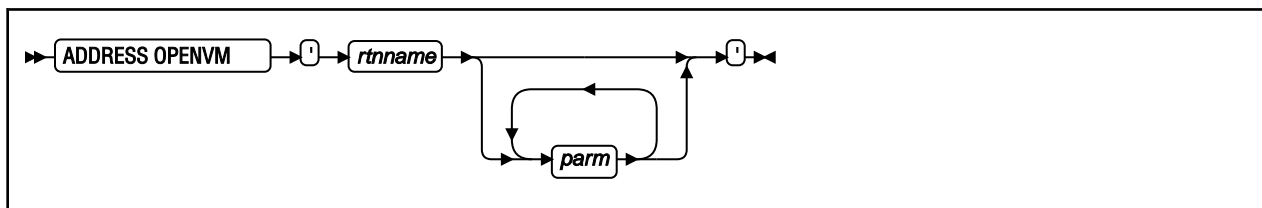
rtnname
> is the name of the OPENVM routine to be called.

parm
> is the name of one or more parameters to be passed to the OPENVM routine. The number and type of these parameters are routine-dependent. A parameter being passed must be the name of a variable.
>
> ADDRESS OPENVM uses the EXECCOMM interface to build a properly-formatted parameter list before completing the call to the routine.

Usage Notes

1. ADDRESS OPENVM must be used to call OPENVM routines. It may also be used to call other CSL routines that are not OPENVM routines.

2. If REXX successfully calls the OPENVM routine, then the REXX variable *RC* contains a zero return code. See "Return Codes" on page 217 for a list of return code values that can be stored in the REXX variable *RC*. The values of *retvalue*, *retcode*, and *retreason* in the parameter list are set by the routine.

3. If REXX detects an error and is *not* able to successfully call the OPENVM routine, then the REXX variable *RC* contains a negative return code. See "Return Codes" on page 217 for a list of return code values that can be stored in the REXX variable *RC*. If *RC* is not zero after the call, then the values of any output parameters are meaningless because the call to the OPENVM routine was not successfully completed.

Return Codes

The following list shows the possible return codes from ADDRESS OPENVM. These values are returned in the REXX variable *RC*. They reflect the ability of the interface to complete the call to the OPENVM routine. These return codes are in addition to any values returned from the specific OPENVM routine being called.

0
> Routine was completed and control returned to REXX

-7
> Routine not loaded from the callable services library

-8
> Routine was dropped from the callable services library

-9
> Insufficient virtual storage available (See Chapter 1, "REXX General Concepts," on page 1.)

-10
> More parameters than allowed were specified

-11
> Fewer parameters than required were specified

-20
> Invalid call

-22
> Invalid REXX argument

-23
> Subpool create failure

-24
> REXX EXECCOMM fetch failure

-25
> REXX EXECCOMM set failure

-26*nnn*
> Incorrect data length for parameter number *nnn*. Possible reasons are:
>
> - The passed length parameter was greater than the maximum allowed for the parameter
> - A length value greater than 65535 was supplied for a variable-length character or bit string
> - The length specified for an output variable-length character or bit string parameter is greater than the size the variable was initialized to before the call
> - A binary or length input parameter is too big.

-27*nnn*
> Incorrect data or data type for parameter number *nnn*. Possible reasons are:
>
> - A binary value was passed that contained a nonnumeric character
> - A binary value was passed that contained a nonnumeric character or had an initial character that was not numeric, +, -, or ' '
> - A parameter defined as unsigned binary (with a length of 4, 3, 2, or 1) was supplied as a negative value
> - A value supplied for a bit string parameter contained characters other than 0 or 1
> - At least one of the stemmed variables containing values for an input column was not defined.

-28*nnn*
> Incorrect variable name for parameter number *nnn*. Possible reasons are:
>
> - An incorrect variable name was specified for an output variable
> - A quoted literal value was supplied as the name for an output variable
> - A quoted literal value was supplied as the name for a table column stemmed variable name
> - No second quote character was passed for a quoted literal
> - The second quote character for a quoted literal was followed by a character that was not a blank.

-29*nnn*
> Incorrect length value (for example, a negative value) was specified for a length parameter, parameter number *nnn*.

Note: For the last four return codes, the parameters are numbered serially, corresponding to the order in which they are coded. The routine name is always parameter number 001, the next parameter is 002, and so forth.

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.

The SOCKET external function uses the TCP/IP Inter-User Communications Vehicle (IUCV) API to access the TCP/IP internet socket interface. The REXX socket functions are similar to socket calls in the C programming language.

This chapter contains the following sections:

- "Programming Hints and Tips for Using REXX Sockets" on page 219
- "SOCKET External Function" on page 220
- "Tasks You Can Perform Using REXX Sockets" on page 221
- "REXX Socket Functions" on page 223
- "REXX Sockets System Messages" on page 264
- "REXX Sockets Return Codes" on page 265
- Chapter 16, "Sample Programs," on page 271

For general information about sockets, see *z/VM: TCP/IP Programmer's Reference*.

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 (depending on the limit imposed by the OPTIONS MAXCONN statement in the CP Directory entry. The *subtaskid* for a socket set identifies the socket set and should be set to a string value that resembles the application's purpose. This identifier is displayed in the output of a NETSTAT SOCKET command.

- 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'. The SOCKET function resolves the host name into an IP address using the following:

  - If the host name does not contain a dot, then for each value specified as DOMAINSEARCH, as well as the value specified as DOMAINORIGIN in TCPIP DATA, the domain string is appended to the host name, and the host name is resolved using the procedure described below.

  - If the host name contains a dot, but not as the last character, then resolution is attempted using the supplied host name, using the procedure described below, before resorting to appending values to the host name from the domain search list as described above.

  - If the host name ends in a dot, then the dot is removed, and the name lookup proceeds as follows:

    Host name resolution

    1. If DOMAINLOOKUP (in TCPIP DATA) is set to DNS (the default), then each IP address specified in TCPIP DATA as an NSINTERADDR is sent a DNS query in turn, until either the name is resolved or the list of NSINTERADDRs is exhausted.

    2. If the host name is not resolved by an DNS server, the HOSTS SITEINFO file is interrogated to find the IP address.

    3. If DOMAINLOOKUP is set to FILE, the order of steps 1 and 2 is reversed.

    4. If DOMAINLOOKUP is set to DNSONLY, then only step 1 is performed.

5. If DOMAINLOOKUP is set to FILEONLY, then only step 2 is performed.

For additional information on host name resolution, see the descriptions of the DOMAINSEARCH and DOMAINLOOKUP statements in the *z/VM: TCP/IP Planning and Customization*.

- 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 35 (EWOULDBLOCK) or 36 (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.

- If your application uses the GiveSocket and TakeSocket functions to transfer a socket from a concurrent server program to a child server program, both server programs must agree on a mechanism for exchanging client IDs and the socket ID to be transferred. The child server program must also signal the concurrent server program when the TakeSocket function has successfully completed. The concurrent server program can then close the socket.

- The socket options SO_ASCII and SO_EBCDIC identify the socket's data type for use by the REXX socket program. Setting SO_EBCDIC on has no effect. Setting SO_ASCII on causes all incoming data on the socket to be translated from ASCII to EBCDIC and all outgoing data on the socket to be translated from EBCDIC to ASCII. REXX Sockets uses the following hierarchy of translation tables:

subtaskid TCPXLBIN *
userid TCPXLBIN *
STANDARD TCPXLBIN *
RXSOCKET TCPXLBIN *
Internal tables

The first four tables are data sets, and they are searched in the order in which they are listed. If no files are found, the internal tables corresponding to the ISO standard are used.

SOCKET External Function
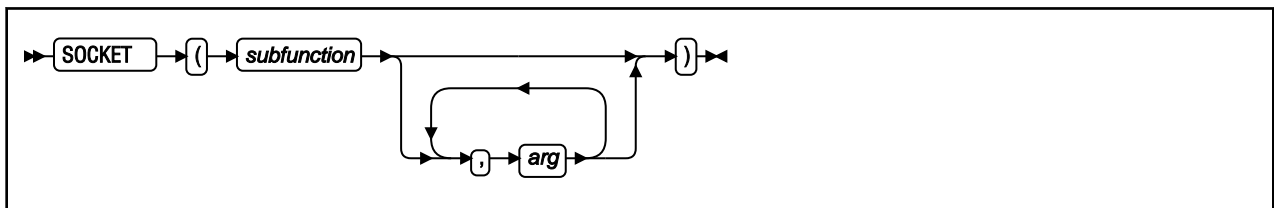
The first parameter in the SOCKET function, *subfunction*, identifies a REXX socket function. The REXX socket function may have additional arguments. REXX socket functions are provided for:

- 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
- Translating data and doing tracing

See "Tasks You Can Perform Using REXX Sockets" on page 221 and "REXX Socket Functions" on page 223.

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)
    '35 EWOULDBLOCK Operation would block'

For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

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 264.

The description of each REXX socket function in this chapter 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 3 on page 221.

Table 3: 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 4 on page 221.

Table 4: 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

Table 4: REXX socket functions for creating, connecting, changing, and closing sockets (continued)	
Function	**Purpose**
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 5: 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
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 6: REXX socket functions for resolving names and other identifiers	
Function	**Purpose**
GetClientId	Gets the calling program's TCP/IP identifier
GetDomainName	Gets the domain name for the host processor
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

Table 6: REXX socket functions for resolving names and other identifiers (continued)	
Function	**Purpose**
GetProtoByName	Gets the network protocol number for a protocol name
GetProtoByNumber	Gets the network protocol name for a protocol number
GetServByName	Gets the port and network protocol name for a service name
GetServByPort	Gets the service name and network protocol name for a port

- **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 7: 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
Cancel	Cancels an outstanding Select function
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 8: REXX socket functions for translating data and doing tracing	
Function	**Purpose**
Translate	Translates data from one type of notation to another
Trace	Enables or disables tracing facilities

REXX Socket Functions

This section describes the REXX socket functions, which are listed alphabetically.

Accept

▶▶── SOCKET ──(── ' ── ACCEPT ── ' ── , ── *socketid* ──) ──◀◀
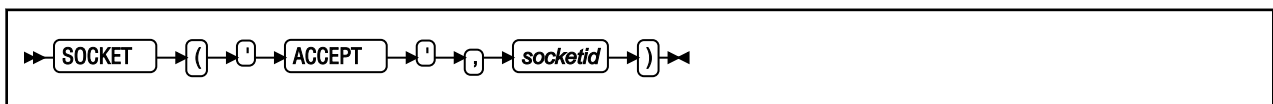
Purpose

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 35 (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.

Parameters

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.

Return Values

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.

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Bind
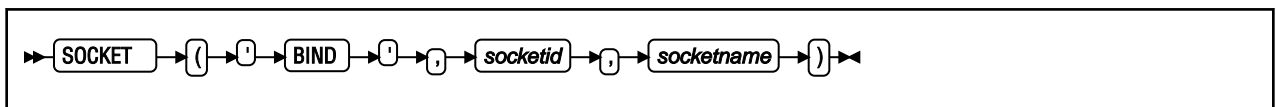
Purpose

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.

Parameters

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 of the socket. This must be either a non-negative integer between 0 and 65535 or INPORT_ANY.

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

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Cancel
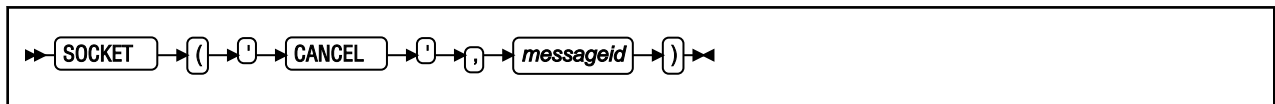
Purpose

Use the Cancel function to cancel an outstanding Select function. The *messageid* used by the Cancel function is obtained from a prior Select function with the IDENTIFY option specified.

The Cancel function can be used in conjunction with asynchronous Select function calls. Each Select can be issued with the IDENTIFY option, and the returned *messageid* can be saved for subsequent Cancel functions. The outstanding Select calls may be terminated by using the Cancel function with the corresponding *messageid*. If the Select function completes and a Cancel function is used, the Cancel will fail with a return code 2018 (EREQUESTNOTACTIVE).

Parameters

messageid
> is the message identifier from a previously issued Select request using the IDENTIFY option. For more information on the IDENTIFY option, see the Select functionon page "IDENTIFY" on page 249.

Return Values

If successful, this function returns a string containing 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
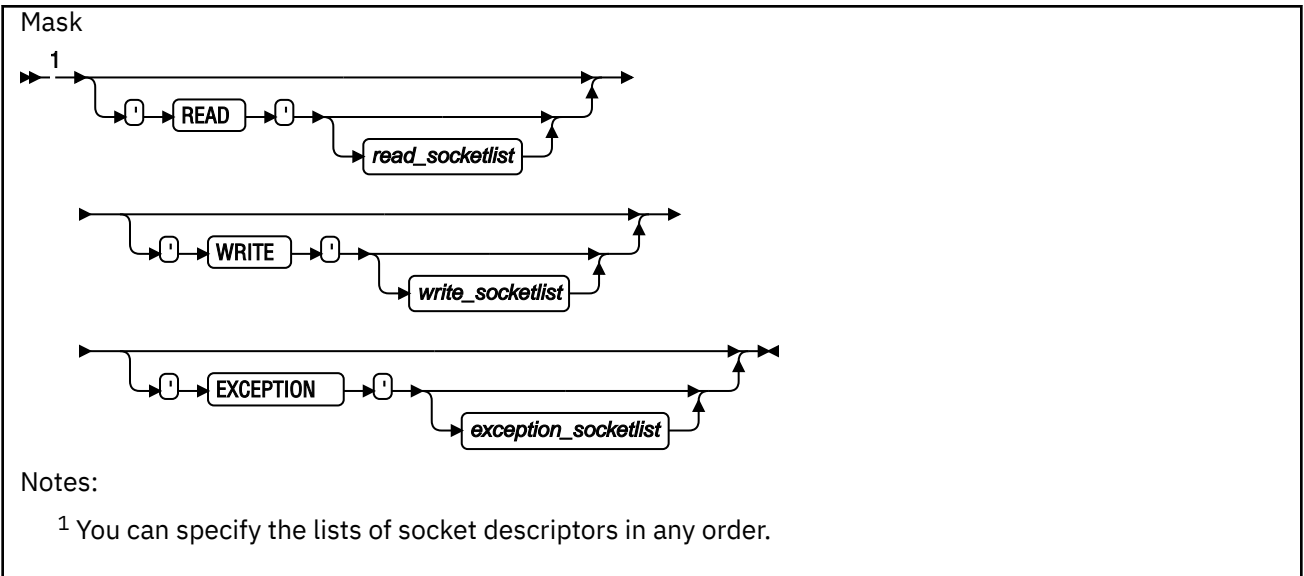```
Socket('Cancel',messageid)
```
        `'0'`

**Note:** A C language equivalent does not exist.

### Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

## Close

```
►►─ SOCKET ─►─ ( ─►─ ' ─►─ CLOSE ─►─ ' ─►─ , ─►─ socketid ─►─ ) ─►◄
```
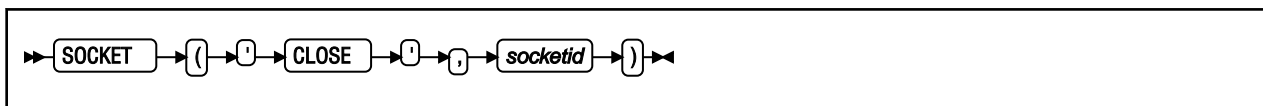
Purpose

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.

Parameters

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 252.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Connect

```
▶▶─ SOCKET ─▶─(─▶─'─▶─ CONNECT ─▶─'─▶─,─▶─ socketid ─▶─,─▶─ socketname ─▶─)─▶◀
```
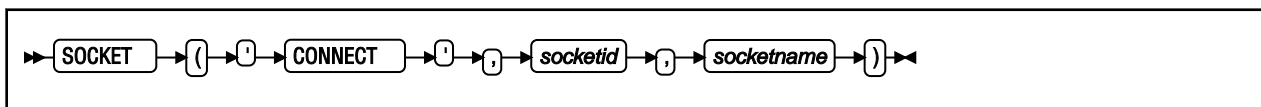
Purpose

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. (The domain, port ID, and IP address are set to AF_INET, INPORT_ANY, and INADDR_ANY.)
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 36 (EINPROGRESS) or another return code indicating an error.

Parameters

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.

Return Values

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'
```

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Fcntl
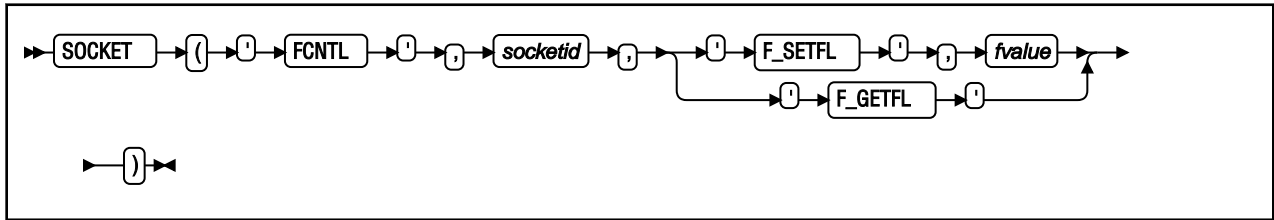
Purpose

Use the Fcntl function to set blocking or nonblocking mode for a socket, or to get the setting for the socket.

Parameters

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 35 (EWOULDBLOCK).

> **BLOCKING or 0**
>> Turns the FNDELAY flag off, which marks the socket as being in blocking mode.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetClientId
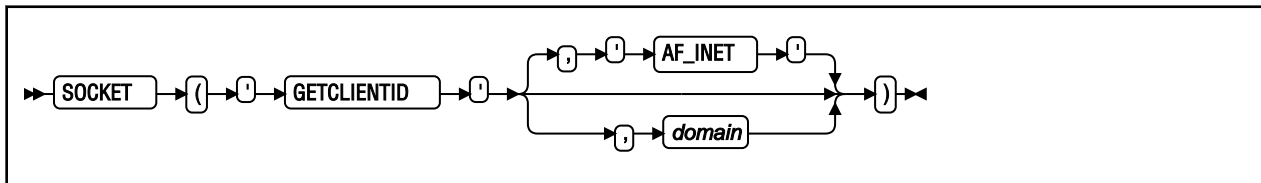
Purpose

Use the GetClientId function to get the identifier by which the calling program is known to the TCP/IP virtual machine.

Parameters

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.
- AF_UNSPEC (or the equivalent decimal value 0).

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetDomainName
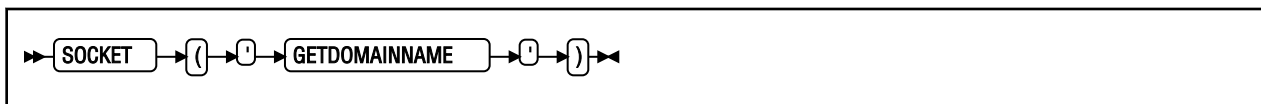
Purpose

Use the GetDomainName function to get the domain name for the processor running the program.

Return Values

If successful, this function returns a string containing return code 0 and the domain name for the 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('GetDomainName')
```
   '0 ZURICH.IBM.COM'
```

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetHostByAddr
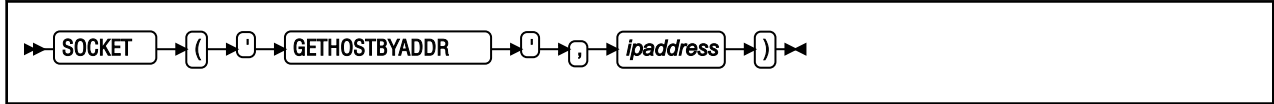
Purpose

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.

Parameters

ipaddress
> is the IP address of the host, in dotted-decimal notation.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetHostByName
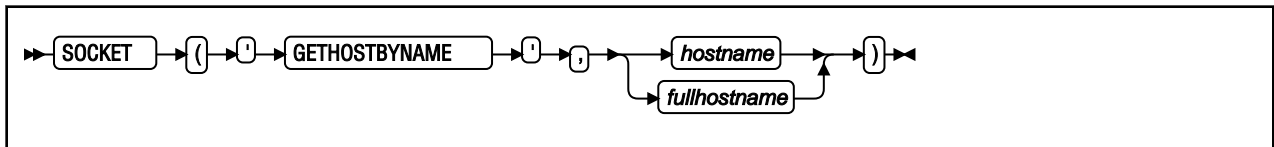
Purpose

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.

Parameters

hostname
    is the host processor name as a character string.

fullhostname
    is the fully qualified host name in the form *hostname.domainname*.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetHostId
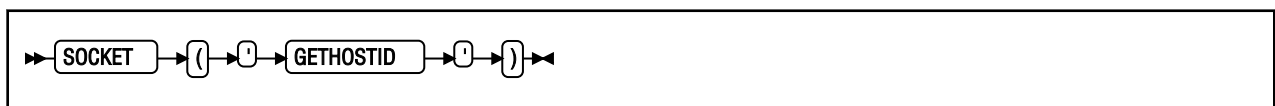
Purpose

Use the GetHostId function to get the IP address for the current host. This address is the default home IP address.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetHostName

```
►►─ SOCKET ─( ─'─ GETHOSTNAME ─'─ )─►◄
```

Purpose

Use the GetHostName function to get the name of the host processor on which the program is running.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetPeerName

```
►►─ SOCKET ─( ─'─ GETPEERNAME ─'─ , ─ socketid ─ )─►◄
```
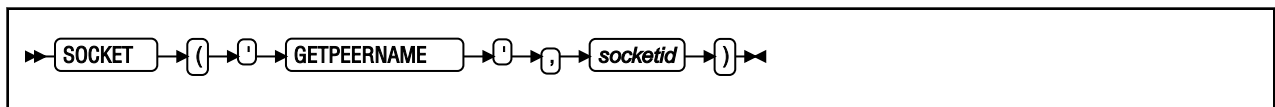
Purpose

Use the GetPeerName function to get the name of the peer connected to a socket.

Parameters

socketid
> is the identifier of the socket.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetProtoByName
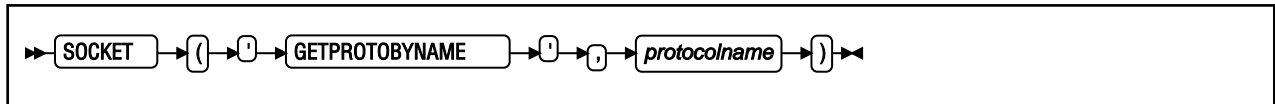
Purpose

Use the GetProtoByName function to get the network protocol number for a specified protocol name.

Parameters

protocolname
> is the name of a network protocol. The names TCP, UDP, and IP are valid.

Return Values

If successful, this function returns a string containing return code 0 and the protocol number. If the protocol name specified on the call is incorrect, the function returns a 0 return code and a null protocol name string. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

Call
> **Return Values**

```
Socket('GetProtoByName','TCP')
```
> `'0 6'`

The C socket call is: `getprotobyname(name)`

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetProtoByNumber
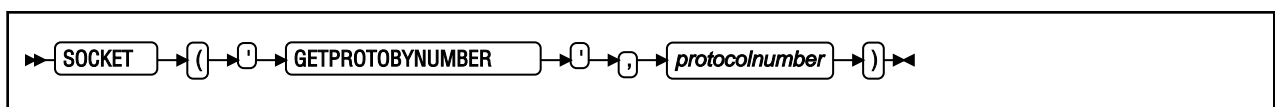
Purpose

Use the GetProtoByNumber function to get the network protocol name for a specified protocol number.

Parameters

protocolnumber
>is the number of a network protocol. This must be a positive integer.

Return Values

If successful, this function returns a string containing return code 0 and the protocol name. If the protocol number specified on the call is incorrect, the function returns a 0 return code and a null protocol number string. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Examples

Call
>**Return Values**

```
Socket('GetProtoByNumber',6)
   '0 TCP'
```

The C socket call is: `getprotobynumber(name)`

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetServByName
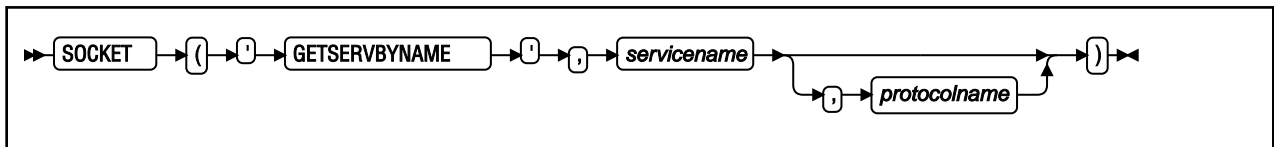
Purpose

Use the GetServByName function to get the port and network protocol name for a specified service name.

Parameters

servicename
>is the service name, such as FTP.

protocolname
>is a network protocol name, such as TCP, UDP, or IP.

Return Values

If successful, this function returns a string containing return code 0, the service name, the port ID, and the protocol name. If the service name specified on the call is incorrect, the function returns a 0 return code with no additional data. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message. If the protocol name specified on the call is incorrect, the return code is 2001.

Examples

Call
>**Return Values**

```
Socket('GetServByName','ftp','tcp')
   '0 FTP 21 TCP'
```

The C socket call is: getservbyname(name, proto)

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetServByPort
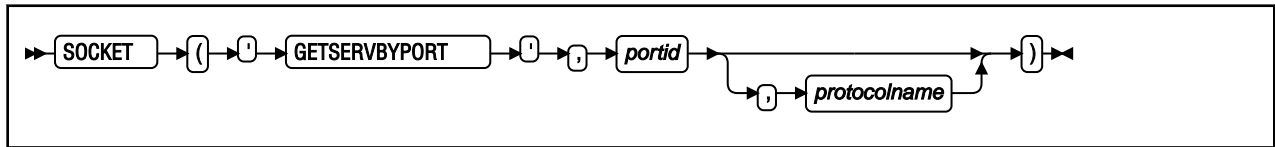
Purpose

Use the GetServByPort function to get the service name and network protocol name for a specified port number.

Parameters

portid
　　is a port number. This must be an integer between 0 and 65535. Instead of a number, you can specify INPORT_ANY or ANY to have TCP/IP get any available port ID.

protocolname
　　is a network protocol name, such as TCP, UDP, or IP.

Return Values

If successful, this function returns a string containing return code 0, the service name, the port ID, and the protocol 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('GetServByPort',21,'tcp')
   '0 FTP 21 TCP'
```

The C socket call is: getservbyport(name, proto)

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetSockName
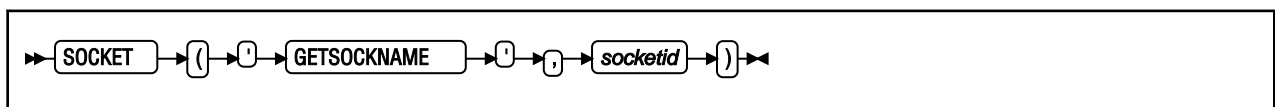
Purpose

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.

Parameters

socketid
> is the identifier of the socket.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GetSockOpt


```
►►─ SOCKET ─( ─'─ GETSOCKOPT ─'─,─ socketid ─,─ level ─,─ optname ─) ─►◄
```
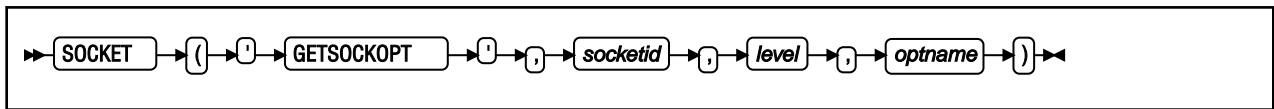
Purpose

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.

Parameters

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 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 a value that indicates the type of information requested:

> **Value**
> > **Description**

> **SO_ASCII**
> > Gets the status of the SO_ASCII option, which controls the translation of data to ASCII. The setting can be On or Off. When SO_ASCII is On, data is translated to ASCII. When SO_ASCII is Off, data is not translated to or from ASCII. This option is ignored by ASCII hosts. In the return string, the option setting is followed by the name of the translation table used, if translation is applied to the data.

SO_BROADCAST
Gets the status of the SO_BROADCAST option, which controls the ability to send broadcast messages over the socket. The setting can be On or Off. This option does not apply to stream sockets.

SO_DEBUG
Gets the status of the SO_DEBUG option, which controls the recording of debug information. The setting can be On or Off.

SO_EBCDIC
Gets the status of the SO_EBCDIC option, which controls the translation of data to EBCDIC. The setting can be On or Off. When SO_EBCDIC is On, data is translated to EBCDIC. When SO_EBCDIC is Off, data is not translated to or from EBCDIC. In the return string, the option setting is followed by the name of the translation table used, if translation is applied to the data.

SO_ERROR
Gets the pending errors on the socket and clears the error status. It can be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors (errors that are not explicitly returned by one of the socket functions).

SO_KEEPALIVE
Gets the status of the SO_KEEPALIVE option, which controls whether a datagram is periodically sent on an idle connection. The setting can be On or Off.

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.

SO_OOBINLINE
Gets the status of the SO_OOBINLINE option, which controls how out-of-band data is to be received. This option applies only to AF_INET stream sockets. The setting can be On or Off:

- When SO_OOBINLINE is On, out-of-band data is placed in the normal data input queue as it is received. The Recv and RecvFrom functions can then receive the data without enabling the the MSG_OOB flag.

- When SO_OOBINLINE is Off, out-of-band data is placed in the priority data input queue as it is received. The Recv and Recvfrom functions must enable the MSG_OOB flag to receive the data.

SO_SNDBUF
Gets the size of the TCP/IP send buffer in OPTVAL.

SO_REUSEADDR
Gets the status of the SO_REUSEADDR option, which controls whether local addresses can be reused. The setting can be On or Off. When SO_REUSEADDR is On, local addresses that are already in use can be bound. Instead of the usual algorithm of checking at Bind time, TCP/IP checks at Connect time to ensure that the local address and port are not the same as the remote address and port. If the association already exists at Connect time, Connect returns Error 48 (EADDRINUSE) and the connect request fails.

SO_TYPE
Gets the socket type, which can be SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW.

TCP_NODELAY
Gets the status of the do-not-delay-the-send flag, which indicates whether TCP can send small packets as soon as possible. The setting can be On or Off.

Return Values

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_ASCII')
   '0 On STANDARD'
```

```
Socket('GetSockOpt',5,'Sol_Socket','So_Broadcast')
   '0 On'
```

```
Socket('GetSockOpt',5,'Sol_Socket','So_Error')
   '0 0'
```

```
Socket('GetSockOpt',5,'Sol_Socket','So_Linger')
   '0 On 60'
```

```
Socket('GetSockOpt',5,'Sol_Socket','So_Sndbuf')
   '0 8192'
```

```
Socket('GetSockOpt',5,'Sol_Socket','So_Type')
   '0 SOCK_STREAM'
```

```
Socket('GetSockOpt',5,'IPproto_TCP','TCP_NoDelay')
   '0 Off'
```

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

GiveSocket
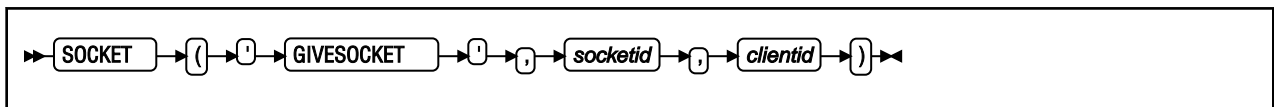
Purpose

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.

Parameters

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 z/VM 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.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Initialize
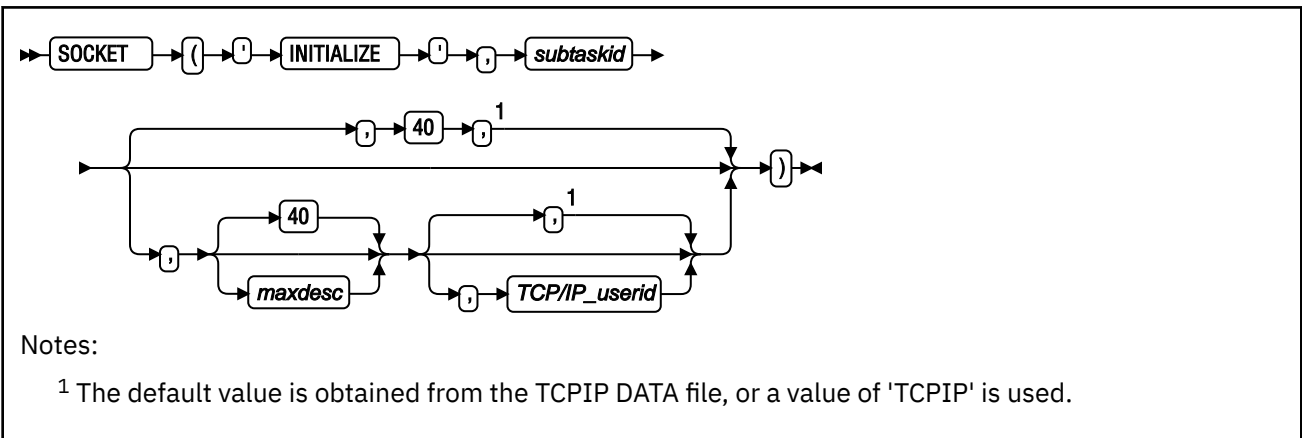
Notes:

[1] The default value is obtained from the TCPIP DATA file, or a value of 'TCPIP' is used.

Purpose

Use the Initialize function to define a socket set. If the function is successful, this socket set becomes the active socket set.

Parameters

subtaskid
> 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, this value is obtained from the TCPIPUserid field of the TCPIP DATA file. If the field is undefined or the file is not available, a value of 'TCPIP' is used.

Return Values

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'
```

Note: A C language equivalent does not exist.

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Ioctl
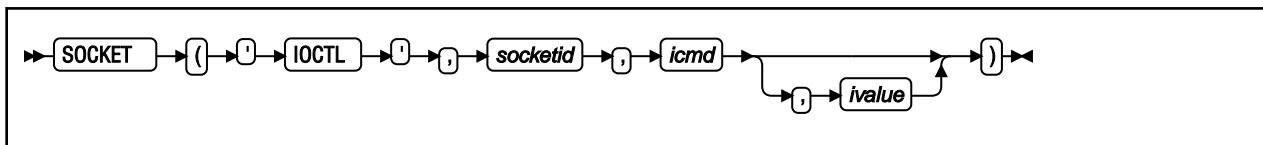
Purpose

Use the Ioctl function to control the operating characteristics of a socket.

Parameters

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*.

> **FIONREAD**
> > Gets the number of immediately readable bytes of data for the socket and returns it in the return string.

> **SIOCATMARK**
> > Determines if the current location in the input data is pointing to out-of-band data. Yes or No is returned in the return string.

> **SIOCDINTERFACE**
> > Removes an interface. You specify the network interface in *ivalue*. If the interface is inactive, the interface configuration information is removed from the TCP/IP server.

> **SIOCGIFADDR**
> > Gets the network interface address. You specify the network interface in *ivalue*. The address is returned in the return string in the format `interface domain port IP_address`.

SIOCGIFBRDADDR
Gets the network interface broadcast address. You specify the network interface in *ivalue*. The address is returned in the return string in the format `interface domain port IP_address`.

SIOCGIFCONF
Gets the network interface configuration. You specify the maximum number of interfaces to be returned in *ivalue*. The list of interfaces is returned in the return string, each in the format `interface domain port IP_address`.

SIOCGIFDSTADDR
Gets the network interface destination address. You specify the network interface in *ivalue*. The address is returned in the return string in the format `interface domain port IP_address`.

SIOCGIFFLAGS
Gets the network interface flags. You specify the network interface in *ivalue*. The network interface is returned in the return string, followed by the flag settings as four hexadecimal digits, followed by the symbolic names of the enabled flags.

SIOCGIFMETRIC
Gets the network interface routing metric. You specify the network interface in *ivalue*. The network interface and the metric integer are returned in the return string.

SIOCGIFNETMASK
Gets the network interface network mask. You specify the network interface in *ivalue*. The network interface is returned in the return string, followed by the network mask in the format `interface domain port IP_address`.

SIOCSIFMETRIC
Sets the network interface routing metric. You specify a string containing the interface name followed by the new metric 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.

Return Values

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'
```
Socket('Ioctl',5,'FionRead')
```
   '0 8192'
```
Socket('Ioctl',5,'SiocAtMark')
```
   '0 No'
```
Socket('Ioctl',5,'Siocdinterface','eth1')
```
   '0'
```
Socket('Ioctl',5,'SiocGifAddr','TR1')
```
   '0 TR1 AF_INET 0 9.4.3.2'
```
Socket('Ioctl',5,'SiocGifConf',2)
```
   '0 TR1 AF_INET 0 9.4.3.2 TR2 AF_INET 0 9.4.3.3'
```
Socket('Ioctl',5,'SiocGifFlags','TR1')
```
   '0 TR1 0063 IFF_UP IFF_BROADCAST IFF_NOTRAILERS IFF_RUNNING'

```
Socket('Ioctl',5,'SiocGifMetric','TR1')
    '0 TR1 0'
```

```
Socket('Ioctl',5,'SiocGifNetMask','TR1')
    '0 TR1 AF_INET 0 255.255.255.0'
```

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Listen
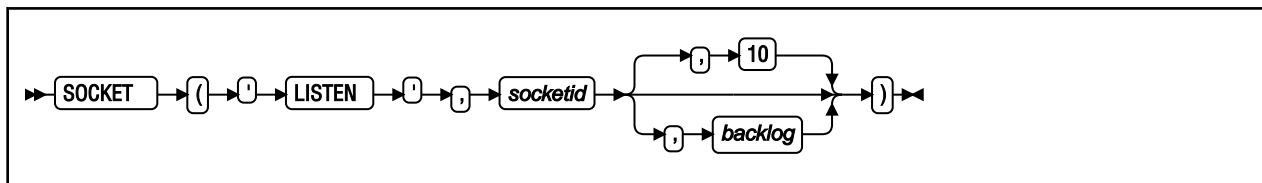
Purpose

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.

Parameters

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.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Read
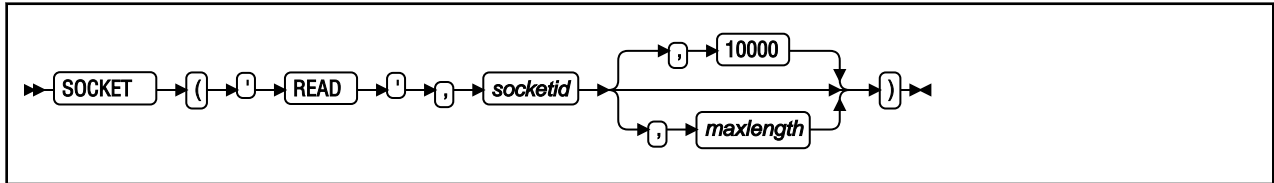
Purpose

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.

Parameters

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. If a number larger than the upper limit of 100000 is given, it will be replaced with 100000. If not specified, the default of 10000 is used.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.
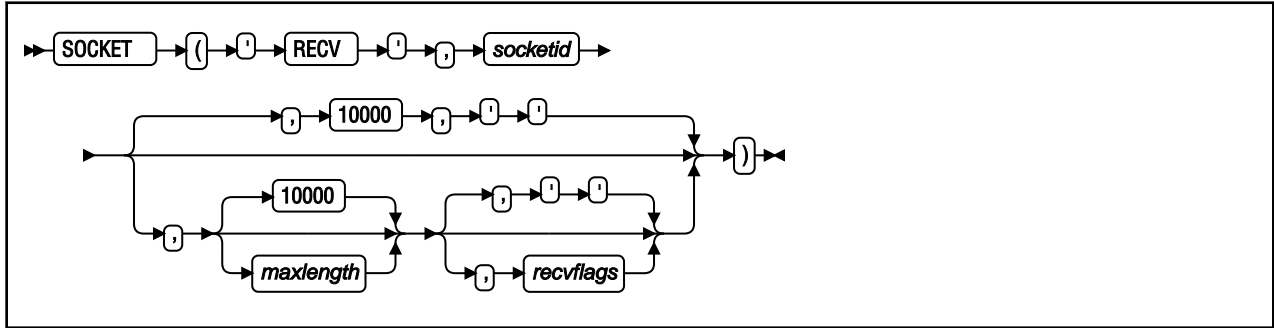
Recv

Purpose

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:

- Read out-of-band data
- Peek at the incoming data without removing it from the buffer

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.

Parameters

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. If a number larger than the upper limit of 100000 is given, it will be replaced with 100000. If not specified, the default of 10000 is used.

recvflags
   are flags that control the Recv operation:

   MSG_OOB or OOB or OUT_OF_BAND
      Read out-of-band data on the socket. Only AF_INET stream sockets support out-of-band data.

   MSG_PEEK or PEEK
      Look at the data on the socket but do not change or destroy it. The next call to Recv can read the same data.

   "
      Receive the data. No flag is set. This is the default.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

RecvFrom
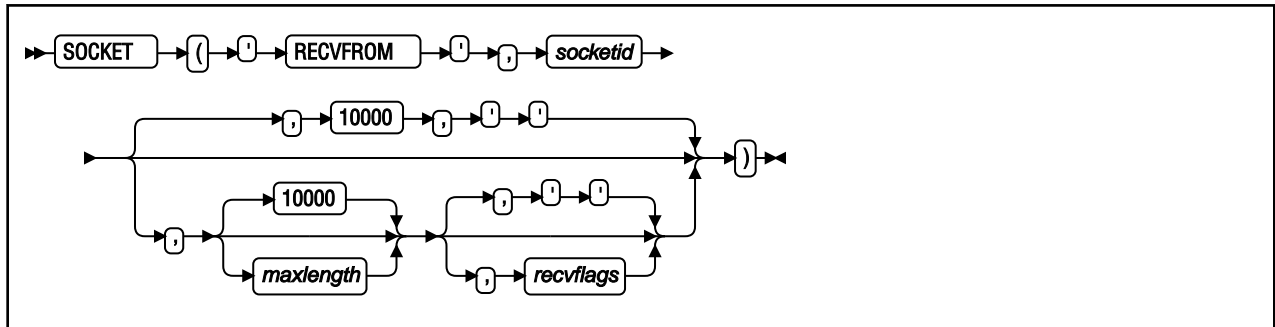
Purpose

Use the RecvFrom function to receive data on a socket, up to a specified maximum number of bytes, and get the sender's address. By specifying option flags, you can also:

- Receive out-of-band data
- Peek at the incoming data without removing it from the buffer

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.

Parameters

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. If a number larger than the upper limit of 100000 is given, it will be replaced with 100000. If not specified, the default of 10000 is used.

recvflags
are flags that control the RecvFrom operation:

MSG_OOB or OOB or OUT_OF_BAND
Read out-of-band data on the socket. Only AF_INET stream sockets support out-of-band data.

MSG_PEEK or PEEK
>   Look at the data on the socket but do not change or destroy it.

"
>   Receive the data. No flag is set. This is the default.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Resolve

Notes:

   [1] Default value obtained from the ResolverTimeout field in the TCPIP DATA file.
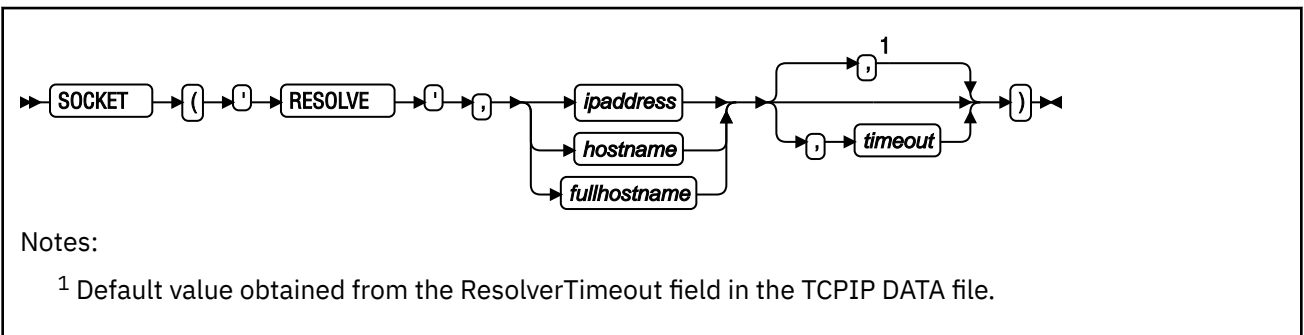
Purpose

Use the Resolve function to resolve the host name through a name server, if one is present.

Parameters

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*.

timeout
>   is a positive integer indicating the maximum wait time in seconds. If a value is not specified, the default wait time is obtained from the ResolverTimeout field in the TCPIP DATA file.

Return Values

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'`

Note: A C language equivalent does not exist.

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Select

Signal Options

Mask

Notes:

[1] You can specify the lists of socket descriptors in any order.
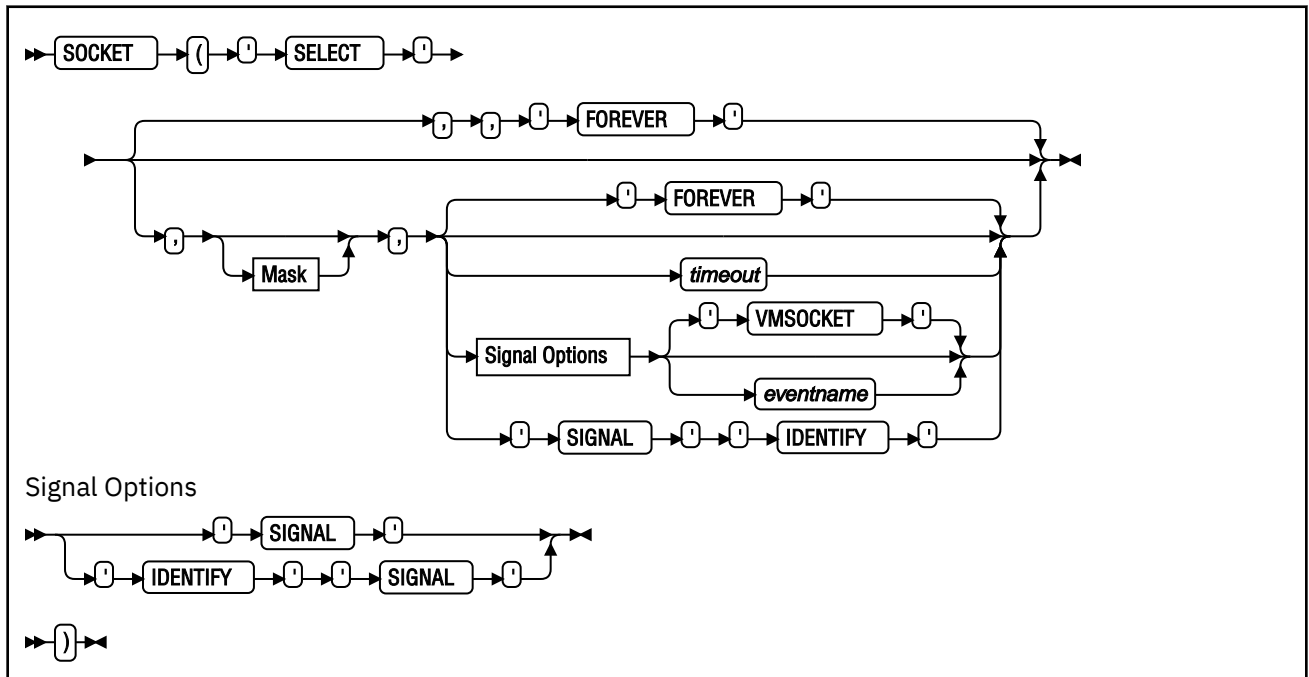
Purpose

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 36 (EINPROGRESS). 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 35 (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.

Parameters

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.

SIGNAL *eventname*
: specifies the name of a CMS multitasking event that is signaled when the Select request completes. The default event is VMSOCKET. The Socket function will end immediately with return code 0 indicating that the Select request has been accepted. Event *eventname* will be signaled (with an event

key length of 0) later on when the Select request actually completes. The data that Select normally returns can be obtained by using the EventRetrieve CSL routine after *eventname* has been signaled.

IDENTIFY

is used to have Select return the *messageid* corresponding to the outstanding Select request. This *messageid* can be used with the Cancel function to cancel the Select request. IDENTIFY should only be used in conjunction with the SIGNAL parameter, since there is no *messageid* available if the Select function call is synchronous.

Since SELECT normally has an event associated with it, when the IDENTIFY keyword immediately follows the SELECT, the VMSOCKET event is implied.

Return Values

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.

Usage Notes

1. To specify a CMS multitasking event name, your program must:

   a. Call the EventCreate CSL routine to create a named event.

   b. Call the EventMonitorCreate routine to associate this event, and possibly others, with a monitor. For example, if you also want to wait on console input, include the VMCONINPUT event on this monitor.

   c. Call the Socket function to create the socket.

   d. Call the Select function, specifying the event name.

   e. Call the EventWait routine to wait for the monitor to wake up.

   f. Call the EventRetrieve routine to get the results.

   For information about event management, including descriptions of the event-related CSL routines, see *z/VM: CMS Application Multitasking*.

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.
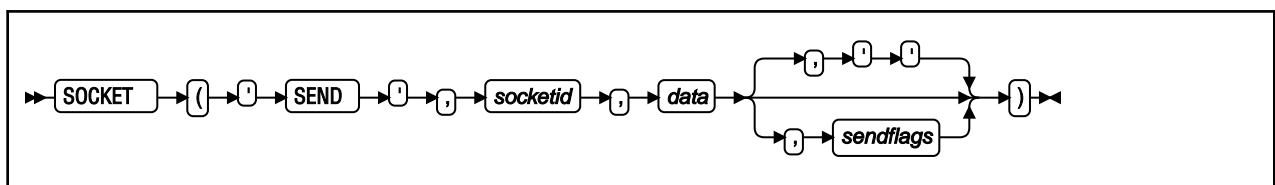
Send

Purpose

Use the Send function to send data on a connected socket. By specifying option flags, you can also:

- Send out-of-band data
- Suppress the use of local routing tables

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.

Parameters

socketid
    is the identifier of the socket.

data
    is the message string to be sent.

sendflags
    are flags that control the Send operation:

    MSG_OOB or OOB or OUT_OF_BAND
        Sends out-of-band data on the socket. Only AF_INET stream sockets support out-of-band data.

    MSG_DONTROUTE or DONTROUTE
        Do not route the data. Routing is handled by the calling program. This flag is valid only for datagram sockets.

    "
        Send the data. No flag is set. This is the default.

Return Values

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'`

```
Socket('Send',6,'Out-of-band data','OOB')
```
    `'0 16'`

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

SendTo
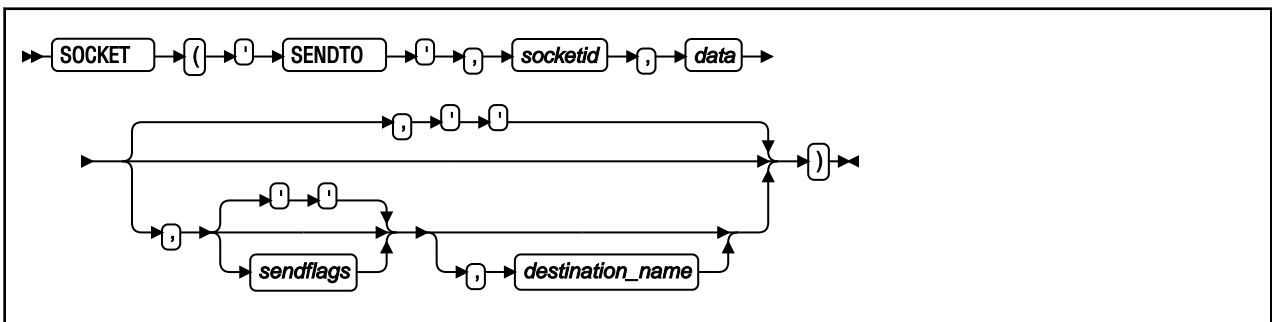
Purpose

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. By specifying option flags, you can also:

- Send out-of-band data
- Suppress the use of local routing tables

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.

Parameters

socketid
    is the identifier of the socket.

data
    is the message string to be sent.

sendflags
    are flags that control the SendTo operation:

    MSG_OOB or OOB or OUT_OF_BAND
        Sends out-of-band data on the socket. Only AF_INET stream sockets support out-of-band data.

    MSG_DONTROUTE or DONTROUTE
        Do not route the data. Routing is handled by the calling program.

    "
        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.

Return Values

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'
Socket('SendTo',6,'some text',,'AF_INET 5678 ZURLVM1')
   '0 9'
```

```
Socket('SendTo',6,'some text',,'AF_INET 5678

ZURLVM1.ZURICH.IBM.COM')
   '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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

SetSockOpt
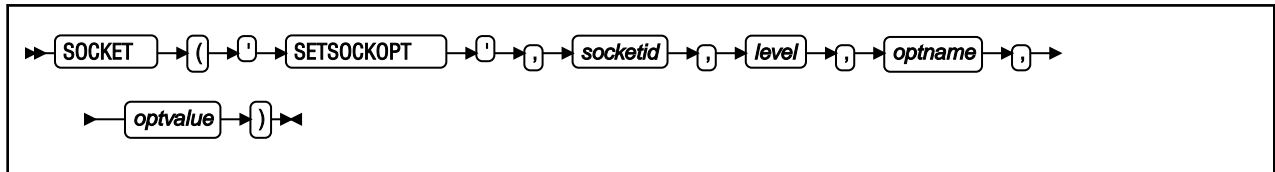
Purpose

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.

Parameters

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_ASCII
        Controls the translation of data to ASCII. When SO_ASCII is On, data is translated to ASCII. When SO_ASCII is Off, data is not translated to or from ASCII. Following the setting in the *optvalue* parameter, you can specify the name of the translation table used, if translation is applied to the data. This option is ignored by ASCII hosts.

    SO_BROADCAST
        Controls the ability to send broadcast messages over the socket. This option does not apply to stream sockets. The initial setting is Off.

    SO_DEBUG
        Controls the recording of debug information. The initial setting is Off.

    SO_EBCDIC
        Controls the translation data to EBCDIC. When SO_EBCDIC is On, data is translated to EBCDIC. When SO_EBCDIC is Off, data is not translated to or from EBCDIC. Following the setting in the *optvalue* parameter, you can specify the name of the translation table used, if translation is applied to the data. This option is ignored by EBCDIC hosts.

SO_KEEPALIVE

Controls whether a datagram is periodically sent on an idle connection. If the remote TCP/IP does not respond to this datagram or to retransmissions of this datagram, the connection ends with error code 60 (ETIMEDOUT). The initial setting is Off.

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.

This option applies only to stream sockets. The initial setting is Off.

SO_OOBINLINE

Controls how out-of-band data is to be received:

- When SO_OOBINLINE is On, out-of-band data is placed in the normal data input queue as it is received. The Recv and RecvFrom functions can then receive the data without enabling the the MSG_OOB flag.

- When SO_OOBINLINE is Off, out-of-band data is placed in the priority data input queue as it is received. The Recv and RecvFrom functions must enable the MSG_OOB flag to receive the data.

This option applies only to stream sockets. The initial setting is Off.

SO_REUSEADDR

Controls whether local addresses can be reused. When SO_REUSEADDR is On, local addresses that are already in use can be bound. Instead of the usual algorithm of checking at Bind time, TCP/IP checks at Connect time to ensure that the local address and port are not the same as the remote address and port. If the address and port are duplicated at Connect time, Connect returns error code 48 (EADDRINUSE) and the connect request fails.

The initial setting is Off.

TCP_NODELAY

Controls the do-not-delay-the-send flag, which indicates whether TCP/IP can send small packets as soon as possible.

optvalue

is the option setting.

For all options except SO_LINGER, you can specify On, Off, or a number (0 = Off, nonzero = On).

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.

Return Values

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_ASCII','On')
```
   '0'
```
Socket('SetSockOpt',5,'Sol_Socket','So_Broadcast','On')
```
   '0'

```
Socket('SetSockOpt',5,'Sol_Socket','So_Linger',60)
    '0'
Socket('SetSockOpt',5,'IPproto_TCP','TCP_NoDelay','On')
    '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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

ShutDown
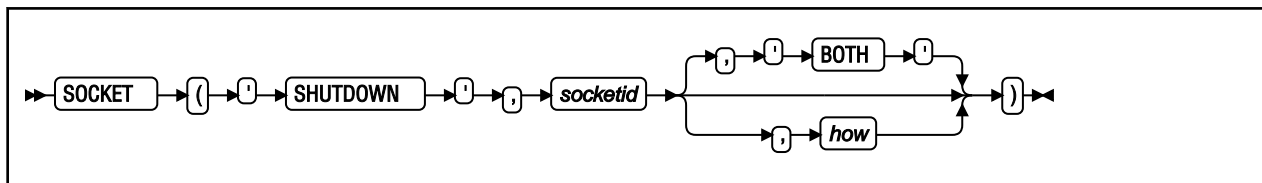
Purpose

Use the ShutDown function to shut down all or part of a duplex connection.

Parameters

socketid
> is the identifier of the socket.

how
> sets the communication direction to be shut down:
>
> **FROM or RECEIVE or RECEIVING or READ or READING**
>> Disables further receive-type operations on the socket, ending communication from the socket.
>
> **TO or SEND or SENDING or WRITE or WRITING**
>> Disables further send-type operations on the socket, ending communication to the socket.
>
> **BOTH**
>> Disables further receive-type and send-type operations on the socket, ending communication from and to the socket. This is the default.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Socket

Type and Protocol

Notes:

[1] The default protocol depends on the domain and socket type. There is no default protocol for a raw socket.

Purpose

Use the Socket function to create a socket in the active socket set. Different types of sockets provide different communication services.

Parameters

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. 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. 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.

SOCK_RAW
The abbreviated form RAW is also permitted. This type of socket provides the interface to internal protocols, such as IP and ICMP. You can specify this type only if you have been authorized to create raw sockets by the TCP/IP server virtual machine.

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

There is no default protocol for a raw socket. For raw sockets, the following protocols are valid:

- IPPROTO_ICMP
- IPPROTO_RAW or RAW (equivalent values)

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.
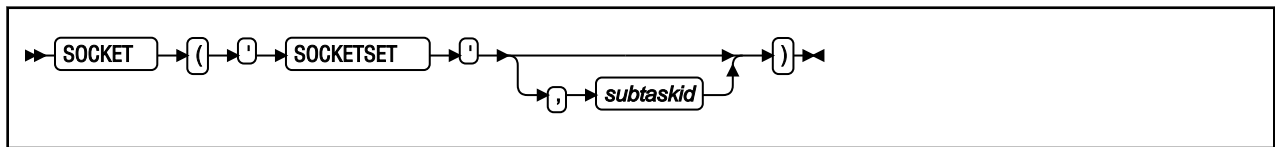
SocketSet

Purpose

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.

Parameters

subtaskid
is the name of a socket set. The name can be up to eight printable characters; it cannot contain blanks.

Return Values

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'
```

Note: A C language equivalent does not exist.

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

SocketSetList

```
►►─ SOCKET ─( ─'─ SOCKETSETLIST ─'─) ─►◄
```

Purpose

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.

Return Values

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'
```

Note: A C language equivalent does not exist.

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

SocketSetStatus

```
►►─ SOCKET ─( ─'─ SOCKETSETSTATUS ─'─┬──────────────┬─) ─►◄
                                     └─,─ subtaskid ─┘
```
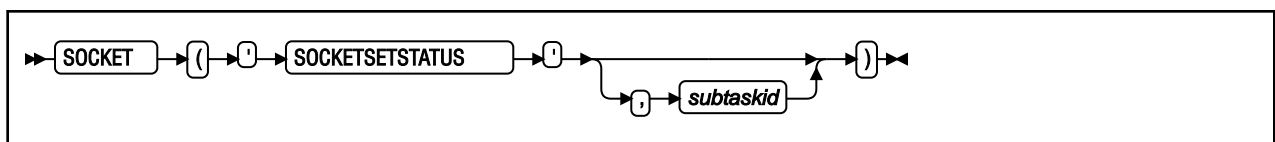
Purpose

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.

Parameters

subtaskid
> is the name of a socket set. The name can be up to eight printable characters; it cannot contain blanks.

Return Values

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'`

Note: A C language equivalent does not exist.

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

TakeSocket
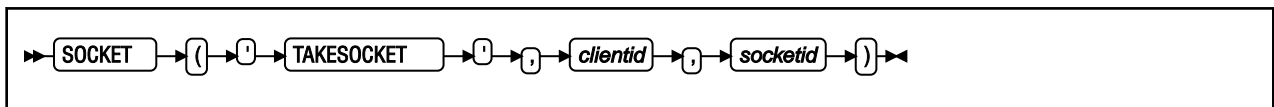
Purpose

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.

Parameters

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 z/VM 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.

Return Values

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'
```

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Terminate
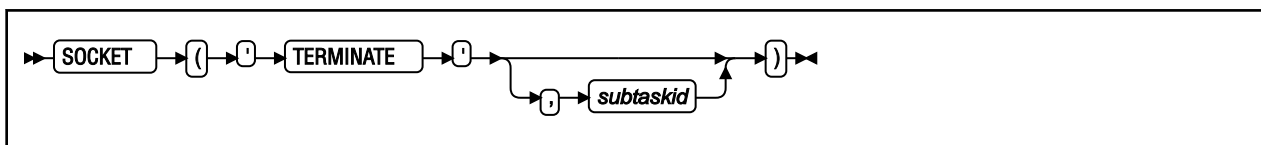
Purpose

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.

Parameters

subtaskid
> is the name of the socket set. The name can be up to eight printable characters; it cannot contain blanks.

Return Values

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'
```

Note: A C language equivalent does not exist.

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Trace
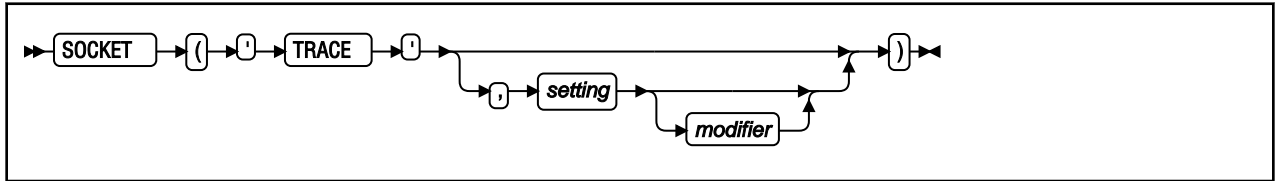
Purpose

Use the Trace function to enable or disable tracing facilities.

Parameters

setting
indicates the type of tracing. The supported settings are:

ON
Enables both IUCV and Resolver tracing

IUCV
Enables IUCV tracing only

RESOLVER
Enables Resolver tracing only (tracing of domain name server queries)

OFF
Disables all tracing

modifier
allows you to enable or disable a specific trace. This parameter is valid only with the IUCV and RESOLVER settings. You can specify either ON or OFF. The default is ON.

Return Values

If successful, this function returns a string containing return code 0 and the previous Trace setting. If unsuccessful, this function returns a string containing a nonzero return code, an error name, and an error message.

Usage Notes

1. Because tracing can generate large amounts of console traffic, this function should be used with caution.

2. If Trace is specified without a setting, it returns the current setting.

Examples

Call
     Return Values

```
Socket('Trace')
   '0 Off'
```

```
Socket('Trace','IUCV')
   '0 Off'
Socket('Trace')
   '0 IUCV'
Socket('Trace','Resolver')
   '0 IUCV'
Socket('Trace')
   '0 On'
Socket('Trace','IUCV Off')
   '0 On'
Socket('Trace')
   '0 Resolver'
Socket('Trace','On')
   '0 Resolver'
Socket('Trace')
   '0 On'
Socket('Trace','Off')
   '0 On'
Socket('Trace')
   '0 Off'
```

Note: A C language equivalent does not exist.

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Translate
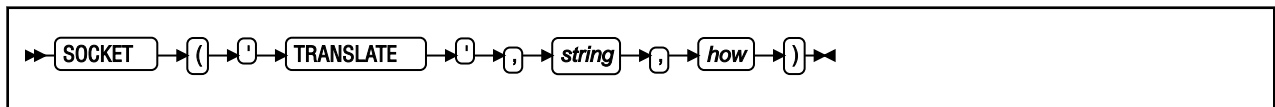
Purpose

Use the Translate function to translate data from one type of notation to another.

Parameters

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

Return Values

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.

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','0002007B80E4010200000000000000000'X,'To_SockAddr_In')` **Return Values:** `'0 23 AF_INET 123 128.228.1.2'`

Note: A C language equivalent does not exist.

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Version

```
►►─ SOCKET ─►─( ─►─ ' ─►─ VERSION ─►─ ' ─►─ ) ─►◄
```

Purpose

Use the Version function to get the version number and date for the REXX Sockets function package.

Return Values

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 *n.nn dd month yyyy*'

Note: A C language equivalent does not exist.

Messages and Return Codes

For a list of REXX Sockets system messages, see "REXX Sockets System Messages" on page 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

Write

```
►►─ SOCKET ─( ─' ─ WRITE ─' ─, ─ socketid ─, ─ data ─} ─►◄
```
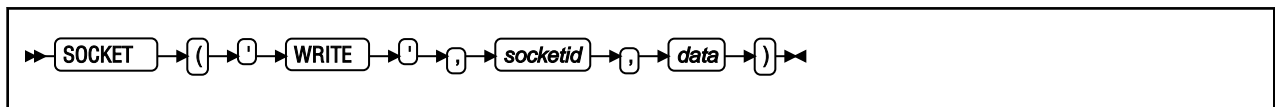
Purpose

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.

Parameters

socketid
>   is the identifier of the socket.

data
>   is the data to be written.

Return Values

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 264. For a list of REXX Sockets return codes, see "REXX Sockets Return Codes" on page 265.

REXX Sockets System Messages

Depending on how your z/VM system is set up, the REXX Sockets module may be invoked from a logical saved segment or from disk. The following error and warning system messages may be displayed during the process of initializing REXX Sockets, or when doing tracing or other functions.

For explanations of these messages, see *z/VM: CP Messages and Codes* or enter:

```
help msg msgno
```

```
help msg msgno
```

DMS1400E	**Unable to acquire Dynamic Save Area storage**	**DMS1419E**	**{EventCreate|EventSignal| EventDelete} failed for event** *event_name*; **RC=**nn **Reason=**nn
DMS1401E	**System-dependent initialization module not found**	**DMS1420E**	**Abend** *nnn* **detected in REXX/ Sockets at (**where**)+**nnnn**=**xxxxxxxx
DMS1402E	**Unable to initialize REXX/Sockets Global Work Area**	**DMS1421E**	**RXSOCKET loaded at** *xxxxxxxx*; **Global Work Area at** *xxxxxxxx*
DMS1403I	**>** *name*+*nnnn* **calls** *name* **R0-3:** *registers*	**DMS1422E**	**-------------- Registers at time of failure: --------------**
DMS1404I	**> to** *name*+*nnnn* **CC=**cc **R0-3:** *registers*	**DMS1423E**	**PSW:** *xxxxxxxx xxxxxxxx*
DMS1404I	**> R4-7:** *registers*	**DMS1424E**	**R**xx**-R**xx**:** *xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx*
DMS1405E	**Return code** *nn* **from NUCEXT SET for** *xxxxxxxx*	**DMS1425E**	**-------------------- Module traceback: --------------------**
DMS1406I	**Unable to establish ABEND exit; processing continues**	**DMS1426E**	*nnnnnnnnnn* **called from** *nnnnnnnnnn*+*xxxx* **with DSA at** *xxxxxxxx*
DMS1407I	**REXX/Sockets anchor located by NUCEXT {CALL|QUERY|SET}**	**DMS1427E**	**----------------- First 80 bytes of DSA: -----------------**
DMS1408W	**File** *fn ft* ***** **not found**	**DMS1428E**	*xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx* ***** *nnnnnnnnnnnnnnnn* *****
DMS1409I	**Opening file** *fn ft*		
DMS1410E	**FSSTATE failed; rc=**nn	**DMS1430I**	**REXX/SOCKETS 3.01 12 April 1996**
DMS1412I	**File** *fn ft fm* **opened successfully**	**DMS1431I**	**IUCV SEND DATA={PRMMSG| BUFFER|BUFFER, BUFFLIST=YES}**
DMS1413I	**REXX source:** *source_string*		
DMS1414I	**REXX clause:** *statement*	**DMS1432I**	**PRMMSG1:** *xxxxxxxx* **PRMMSG2:** *xxxxxxxx*
DMS1415I	**RXSOCKET - REXX/Sockets (for VM)REXX support for the TCP/IP Socket Interface Type: 'HELP RXSOCKET' for more information**	**DMS1433I**	**The variations of this message are explained below. MESSAGES: o Buffer: (**nnnnnn **bytes) o Reply: (**nnnnnn **bytes with IUCV message ID:** nnnn**) o Buffer** n**: (**nnnnnn **bytes)**
DMS1416I	**Multitasking environment detected; switching to Block/ Unblock mode**		
DMS1417I	**Blocking thread** *nn* **in process** *nn*	**DMS1434I**	*xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx*
DMS1418I	**Unblocked thread** *nn* **in process** *nn*		

DMS1435I	Connecting to NameServer: *nnn.nnn.nnn.nnn*, Time: *hh:mm:ss*
DMS1436I	Question to NameServer: *nnn.nnn.nnn.nnn*, ResolverTimeout: *nnn* seconds
DMS1437I	Answer from NameServer: *nnn.nnn.nnn.nnn*, Time: *hh:mm:ss*
DMS1438I	*xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx*
DMS1440E	RXSOCKET requires TCPIP Version 2 or higher

DMS1441E	IUCV error; IPAUDIT: *xxxxxxxx*
DMS1442I	The variations of this message are explained below. MESSAGES: o IUCV *nnnnnnnn* interrupt for socket call *nnnnnnnn* on socket {*socket*|*PATH*} {*socket number*|*path id*} o IUCV *nnnnnnnn* interrupt on PATH *path id* (*socket set name*) o IUCV *nnnnnnnn* interrupt on PATH *path id* (*socket set name*), Reason: *reason*

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
0	-	-
1	EPERM	Not owner
2	ENOENT	No such file or directory
3	ESRCH	No such process
4	EINTR	Interrupted system call
5	EIO	I/O error
6	ENXIO	No such device or address
7	E2BIG	Arg list too long
8	ENOEXEC	Exec format error
9	EBADF	Bad file number
10	ECHILD	No children
11	EAGAIN	No more processes
12	ENOMEM	Not enough memory
13	EACCES	Permission denied
14	EFAULT	Bad address
15	ENOTBLK	Block device required
16	EBUSY	Device busy
17	EEXIST	File exists
18	EXDEV	Cross-device link
19	ENODEV	No such device

Code	Error Name	Error Message
20	ENOTDIR	Not a directory
21	EISDIR	Is a directory
22	EINVAL	Argument not valid
23	ENFILE	File table overflow
24	EMFILE	Too many open files
25	ENOTTY	Inappropriate ioctl for device
26	ETXTBSY	Text file busy
27	EFBIG	File too large
28	ENOSPC	No space left on device
29	ESPIPE	Illegal seek
30	EROFS	Read-only file system
31	EMLINK	Too many links
32	EPIPE	Broken pipe
33	EDOM	Argument too large
34	ERANGE	Result too large
35	EWOULDBLOCK	Operation would block
36	EINPROGRESS	Operation now in progress
37	EALREADY	Operation already in progress
38	ENOTSOCK	Socket operation on non-socket
39	EDESTADDRREQ	Destination address required
40	EMSGSIZE	Message too long
41	EPROTOTYPE	Protocol wrong type for socket
42	ENOPROTOOPT	Option not supported by protocol
43	EPROTONOSUPPORT	Protocol not supported
44	ESOCKTNOSUPPORT	Socket type not supported
45	EOPNOTSUPP	Operation not supported on socket
46	EPFNOSUPPORT	Protocol family not supported
47	EAFNOSUPPORT	Address family not supported by protocol family
48	EADDRINUSE	Address already in use
49	EADDRNOTAVAIL	Cannot assign requested address
50	ENETDOWN	Network is down
51	ENETUNREACH	Network is unreachable
52	ENETRESET	Network dropped connection on reset
53	ECONNABORTED	Software caused connection abort
54	ECONNRESET	Connection reset by peer

Code	Error Name	Error Message
55	ENOBUFS	No buffer space available
56	EISCONN	Socket is already connected
57	ENOTCONN	Socket is not connected
58	ESHUTDOWN	Cannot send after socket shutdown
59	ETOOMANYREFS	Too many references: cannot splice
60	ETIMEDOUT	Connection timed out
61	ECONNREFUSED	Connection refused
62	ELOOP	Too many levels of symbolic links
63	ENAMETOOLONG	File name too long
64	EHOSTDOWN	Host is down
65	EHOSTUNREACH	Host is unreachable
66	ENOTEMPTY	Directory not empty
67	EPROCLIM	Too many processes
68	EUSERS	Too many users
69	EDQUOT	Disc quota exceeded
70	ESTALE	Stale NFS file handle
71	EREMOTE	Too many levels of remote in path
72	ENOSTR	Not a stream device
73	ETIME	Timer expired
74	ENOSR	Out of stream resources
75	ENOMSG	No message of desired type
76	EBADMSG	Not a data message
77	EIDRM	Identifier removed
78	EDEADLK	Deadlock situation detected/avoided
79	ENOLCK	No record locks available
80	ENONET	Machine is not on the network
81	ERREMOTE	Object is remote
82	ENOLINK	The link has been severed
83	EADV	Advertise error
84	ESRMNT	SRMOUNT error
85	ECOMM	Communication error on send
86	EPROTO	Protocol error
87	EMULTIHOP	Multihop attempted
88	EDOTDOT	Cross mount point
89	EREMCHG	Remote address changed

Code	Error Name	Error Message
90	ECONNCLOSED	Connection closed by peer
1000	EIBMBADCALL	Bad socket-call constant
1001	EIBMBADPARM	Bad parm
1002	EIBMSOCKETOUTOFRANGE	Socket out of range
1003	EIBMSOCKINUSE	Socket in use
1004	EIBMIUCVERR	IUCV error
2001	EINVALIDRXSOCKETCALL	Syntax error in RXSOCKET parameter list
2002	ECONSOLEINTERRUPT	Console interrupt
2003	ESUBTASKINVALID	Subtask ID not valid
2004	ESUBTASKALREADYACTIVE	Subtask already active
2005	ESUBTASKNOTACTIVE	Subtask not active
2006	ESOCKETNOTALLOCATED	Socket could not be allocated
2007	EMAXSOCKETSREACHED	Maximum number of sockets reached
2008	ESOCKETALREADYDEFINED	Socket already defined
2009	ESOCKETNOTDEFINED	Socket not defined
2010	ETCPIPSEVEREDPATH	TCPIP severed IUCV path
2011	EDOMAINSERVERFAILURE	Domain name server failure
2012	EINVALIDNAME	"name" received from TCPIP server not valid
2013	EINVALIDCLIENTID	"clientid" received from TCPIP server not valid
2014	EINVALIDFILENAME	File name specified not valid
2015	ENUCEXTFAILURE	Error during NUCEXT function
2016	EHOSTNOTFOUND	Host not found in SITEINFO file
2017	EIPADDRNOTFOUND	IP address not found in ADDRINFO file
2018	EREQUESTNOTACTIVE	Specified socket request not active
2051	EFORMATERROR	Format error
2052	ESERVERFAILURE	Server failure
2053	EUNKNOWNHOST	Unknown host
2054	EQUERYTYPENOTIMPLEMENTED	Query type not implemented
2055	EQUERYREFUSED	Query refused
2056	EIPADDRNOTFOUND	IP address not found in ETC HOSTS file
2057	EHOSTNOTFOUND	Host not found in ETC HOSTS file

3*nnn* error codes identify internal errors. The following list is provided for IBM use only.

Code	Error Name	Error Message
3001	EIUCVINVALIDPATH	IUCV path ID not valid
3002	EIUCVPATHQUIESCED	IUCV path quiesced

Code	Error Name	Error Message
3003	EIUCVMSGLIMITEXCEEDED	IUCV message limit exceeded
3004	EIUCVNOPRIORITY	IUCV priority message not allowed on this path
3005	EIUCVSMALLBUFFER	IUCV buffer too small
3006	EIUCVBADFETCH	IUCV Fetch Protection Exception
3007	EIUCVBADADDRESS	IUCV Addressing Exception on answer buffer
3008	EIUCVBADMSGCLASS	IUCV conflicting message class/path/msgid
3009	EIUCVPURGEDMSG	IUCV message was purged
3010	EIUCVBADMSGLENGTH	IUCV negative message length
3011	EIUCVPARTNERNOTLOGGEDON	IUCV target userid not logged on
3012	EIUCVPARTNERNOTINITIALIZED	IUCV target userid not enabled for IUCV
3013	EIUCVPATHLIMITEXCEEDED	IUCV path limit exceeded
3014	EIUCVPARTNERPATHLIMIT	IUCV partner path limit exceeded
3015	EIUCVNOTAUTHORIZED	IUCV not authorized
3016	EIUCVINVALIDCPSYSTEMSERVICE	IUCV CP System Service not valid
3018	EIUCVINVALIDMSGLIMIT	IUCV message limit not valid
3019	EIUCVBUFFERALREADYDECLARED	IUCV buffer already declared
3020	EIUCVPARTNERSEVERED	IUCV partner severed path
3021	EIUCVPARTNERNOPRMDATA	IUCV cannot accept data in parmlist
3022	EIUCVSENDLISTINVALID	IUCV SEND buffer list not valid
3023	EIUCVINVALIDBUFFERLENGTH	IUCV negative length in answerlist
3024	EIUCVINVALIDLISTLENGTH	IUCV total list length not valid
3025	EIUCVPRMMSGANSLISTCONFLICT	IUCV PRMMSG/answer-list conflict
3026	EIUCVBUFFERLISTNOTALIGNED	IUCV buffer list not aligned
3027	EIUCVANSWERLISTNOTALIGNED	IUCV answer list not aligned
3028	EIUCVNOCONTROLBUFFER	IUCV no control buffer
3048	EIUCVFUNCTIONNOTSUPPORTED	IUCV function not supported
3052	EIBMNOMSG	IUCV API: No message found
3053	EIBMDIRERR	IUCV API: Errors encountered reading directory
3054	EIBMPROTERR	IUCV API: Protection exception
3055	EIBMADDRERR	IUCV API: Addressing exception
3056	EIBMSPECERR	IUCV API: Specification exception
3057	EIBMOPERR	IUCV API: Operation exception

Chapter 16. Sample Programs

This section describes two sample REXX socket programs that are provided with the REXX Sockets package:

- REXX-EXEC RSCLIENT — a client sample program
- REXX-EXEC RSSERVER — a server sample program

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 virtual machine'
  say 'and returns a number of data lines as requested to the client.'
  say 'It is started with the command:                                '
  say '     RSSERVER                                                  '
  say 'and terminated with the command:                               '
  say '     HX                                                        '
  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 '     RSCLIENT 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     */
address command 'IDENTIFY ( LIFO'
parse upper pull userid . locnode .

/* 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
    address command 'EXECIO 1 EMSG (CASE M STRING' 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 on a VM Userid.                      '
  say 'It returns a number of data lines as requested to the client. '
  say 'It is started with the command:  RSSERVER                      '
  say 'and terminated by issuing HX.                                  '
  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:  RSCLIENT 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

/* 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
```

Appendix A. Error Numbers and Messages

The error numbers produced by syntax errors during processing of REXX programs are all in the range 3-49 (and this is the value placed in the variable RC when SIGNAL ON SYNTAX event is trapped). The language processor adds 20000 to these error return codes before leaving an exec in order to provide a different range of codes than those used by CMS EXEC and EXEC 2. When the language processor displays an error message, it first sets the CMSTYPE indicator to RT (Resume Typing), ensuring that the message will be seen by the user, even if HT (Halt Typing) was in effect when the error occurred.

External interfaces to the language processor can generate three of the error messages either before the language processor gains control or after control has left the language processor. Therefore, SIGNAL ON SYNTAX cannot trap these errors. The error numbers involved are: 3 and 5 (if the initial requirements for storage could not be met) and 26 (if on exit the returned string could not be converted to form a valid return code). Error 4 can be trapped only by SIGNAL ON HALT or CALL ON HALT. Five errors the language processor detects cannot be trapped by SIGNAL ON SYNTAX unless the label SYNTAX appears earlier in the program than the clause with the error. These errors include: 6, 12, 13, 22, and 30.

The CP command SET EMSG ON causes error messages to be prefixed with a CMS error code. The full form of the message, including this error code, is given in the message list.

The message list is ordered by the CMS message number. For your convenience, a table cross-referencing the REXX error code with the CMS message number precedes the message list.

In the message list, each message is followed by an explanation giving possible causes for the error. The same explanation can be obtained from CMS using the following command:

```
HELP MSG DMSnnnE  (where nnn is the CMS error number
and error type is either 'E' or'T')
```

The following is a list of the error codes and their associated CMS messages:

Table 9: List of Error Codes and CMS Messages

Error code	CMS message	Error code	CMS message
No number	DMSREX255T	Error 26	DMSREX466E
Error 3	DMSREX451E	Error 27	DMSREX467E
Error 4	DMSREX452E	Error 28	DMSREX486E
Error 5	DMSREX450E	Error 29	DMSREX487E
Error 6	DMSREX453E	Error 30	DMSREX468E
Error 7	DMSREX454E	Error 31	DMSREX469E
Error 8	DMSREX455E	Error 32	DMSREX492E
Error 9	DMSREX456E	Error 33	DMSREX488E
Error 10	DMSREX457E	Error 34	DMSREX470E
Error 11	DMSREX458E	Error 35	DMSREX471E
Error 12	DMSREX459E	Error 36	DMSREX472E
Error 13	DMSREX460E	Error 37	DMSREX473E
Error 14	DMSREX461E	Error 38	DMSREX489E
Error 15	DMSREX462E	Error 39	DMSREX474E
Error 16	DMSREX463E	Error 40	DMSREX475E

Table 9: List of Error Codes and CMS Messages (continued)

Error code	CMS message	Error code	CMS message
Error 17	DMSREX465E	Error 41	DMSREX476E
Error 18	DMSREX491E	Error 42	DMSREX477E
Error 19	DMSREX482E	Error 43	DMSREX478E
Error 20	DMSREX483E	Error 44	DMSREX479E
Error 21	DMSREX464E	Error 45	DMSREX480E
Error 22	DMSREX449E	Error 46	DMSREX218E
Error 23	DMSREX1106E	Error 47	DMSREX219E
Error 24	DMSREX484E	Error 48	DMSREX490E
Error 25	DMSREX485E	Error 49	DMSREX481E

In these messages, the term "language processor" refers to the z/VM REXX/VM interpreter.

In addition to the following error messages, the language processor issues the terminal (unrecoverable) message DMSREX255T Insufficient storage for Exec interpreter.

For information and a complete listing of the REXX/VM error messages, see *z/VM: CMS and REXX/VM Messages and Codes*.

Appendix B. 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:

- Symbol and string handling capabilities with DBCS characters
- An option that allows DBCS characters in symbols, comments, and literal strings.
- An option that allows data strings to contain DBCS characters.
- 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 3, "Functions," on page 67. 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.
- 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 10: DBCS Ranges

Byte	EBCDIC
1st	X'41' to X'FE'
2nd	X'41' to X'FE'
DBCS blank	X'4040'

- DBCS alphanumeric and special symbols

  A DBCS contains double-byte representation of alphanumeric and special symbols corresponding to those of the Single-Byte Character Set (SBCS). In EBCDIC, the first byte of a double-byte alphanumeric or special symbol is X'42' and the second is the same hex code as the corresponding EBCDIC code.

  Here are some examples:

  X'42C1' is an EBCDIC double-byte A
  X'4281' is an EBCDIC double-byte a
  X'427D' is an EBCDIC double-byte quote

- No case translation

  In general, there is no concept of lowercase and uppercase in DBCS.

- Notational conventions

  This appendix uses the following notational conventions:

```
DBCS character            ->     .A .B .C .D
SBCS character            ->     a b c d e
DBCS blank                ->     '. '
EBCDIC shift-out (X'0E')  ->     <
EBCDIC shift-in  (X'0F')  ->     >
```

Note: In EBCDIC, the shift-out (SO) and shift-in (SI) characters distinguish DBCS characters from SBCS characters.

Enabling DBCS Data Operations and Symbol Use

The OPTIONS instruction controls how REXX regards DBCS data. To enable DBCS operations, use the EXMODE option. To enable DBCS symbols, use the ETMODE option on the OPTIONS instruction; this must be the first instruction in the program. (See "OPTIONS" on page 46 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 nonblank DBCS codes as indicated in Table 10 on page 279.

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.

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.

Instruction Examples

Here are some examples that illustrate how instructions work with DBCS.

Note: <.A> is used only as a shortcut notational convention. In EBCDIC, actual instructions using the DBCS support must be written in hexadecimal. For example,

```
DATATYPE('<.A.B.C>')
```

would actually be coded as

```
DATATYPE('0E42C142C242C30F'X)
```

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 program stack. Since a stack entry is limited to 255 bytes, the *expression* must be truncated less than 256 bytes. If the truncation splits a DBCS string, REXX will insure that the integrity of the SO-SI pairing will be kept under OPTIONS EXMODE.

SAY and TRACE

The SAY and TRACE instructions write data to the output stream. As was true for the PUSH and QUEUE instructions, REXX will guarantee the SO-SI pairs are kept for any data that is separated to meet the requirements of the output stream. The SAY and TRACE instructions display data on the user's terminal. As was true for the PUSH and QUEUE instructions, REXX will guarantee the SO-SI pairs are kept for any data that is separated to meet the requirements of the terminal line size. This is generally 130 bytes or fewer if the DIAG-24 value returns a smaller value.

When the data is split up in shorter lengths, again the DBCS data integrity is kept under OPTIONS EXMODE. In EBCDIC, if the terminal line size is less than 4, the string is treated as SBCS data, because 4 is the minimum for mixed string data.

UPPER

Under OPTIONS EXMODE, the UPPER instruction translates only SBCS characters in contents of one or more variables to uppercase, but it never translates DBCS characters. If the content of a variable is not valid mixed string data, no uppercasing occurs.

DBCS Function Handling

Some built-in functions can handle DBCS. The functions that deal with word delimiting and length determining conform with the following rules under OPTIONS EXMODE:

1. **Counting characters**—Logical character lengths are used when counting the length of a string (that is, 1 byte for one SBCS logical character, 2 bytes for one DBCS logical character). In EBCDIC, SO and SI are considered to be transparent, and are not counted, for every string operation.

2. **Character extraction from a string**—Characters are extracted from a string on a logical character basis. In EBCDIC, leading SO and trailing SI are not considered as part of one DBCS character. For instance, .A and .B are extracted from <.A.B>, and SO and SI are added to each DBCS character when they are finally preserved as completed DBCS characters. When multiple characters are consecutively extracted from a string, SO and SI that are between characters are also extracted. For example, .A><.B is extracted from <.A><.B>, and when the string is finally used as a completed string, the SO prefixes it and the SI suffixes it to give <.A><.B>.

   Here are some EBCDIC examples:

   ```
   S1 = 'abc<>def'

   SUBSTR(S1,3,1)        ->      'c'
   SUBSTR(S1,4,1)        ->      'd'
   SUBSTR(S1,3,2)        ->      'c<>d'

   S2 = '<><.A.B><>'

   SUBSTR(S2,1,1)        ->      '<.A>'
   SUBSTR(S2,2,1)        ->      '<.B>'
   SUBSTR(S2,1,2)        ->      '<.A.B>'
   SUBSTR(S2,1,3,'x')    ->      '<.A.B><>x'

   S3 = 'abc<><.A.B>'

   SUBSTR(S3,3,1)        ->      'c'
   SUBSTR(S3,4,1)        ->      '<.A>'
   SUBSTR(S3,3,2)        ->      'c<><.A>'
   DELSTR(S3,3,1)        ->      'ab<><.A.B>'
   DELSTR(S3,4,1)        ->      'abc<><.B>'
   DELSTR(S3,3,2)        ->      'ab<.B>'
   ```

3. **Character concatenation**—String concatenation can only be done with valid mixed strings. In EBCDIC, adjacent SI and SO (or SO and SI) that are a result of string concatenation are removed. Even during implicit concatenation as in the DELSTR function, unnecessary SO and SI are removed.

4. **Character comparison**—Valid mixed strings are used when comparing strings on a character basis. A DBCS character is always considered greater than an SBCS one if they are compared. In all but the strict comparisons, SBCS blanks, DBCS blanks, and leading and trailing contiguous SO and SI (or SI and SO) in EBCDIC are removed. SBCS blanks may be added if the lengths are not identical.

   In EBCDIC, contiguous SO and SI (or SI and SO) between nonblank characters are also removed for comparison.

   Note: The strict comparison operators do not cause syntax errors even if you specify mixed strings that are not valid.

In EBCDIC:

```
     '<.A>' = '<.A. >'        ->    1      /* true  */
  '<><><.A>' = '<.A><><>'     ->    1      /* true  */
  '<>  <.A>' = '<.A>'         ->    1      /* true  */
'<.A><><.B>' = '<.A.B>'       ->    1      /* true  */
       'abc' < 'ab<. >'       ->    0      /* false */
```

5. **Word extraction from a string**—"Word" means that characters in a string are delimited by an SBCS or a DBCS blank.

   In EBCDIC, leading and trailing contiguous SO and SI (or SI and SO) are also removed when *word*s are separated in a string, but contiguous SO and SI (or SI and SO) in a word are not removed or separated for word operations. Leading and trailing contiguous SO and SI (or SI and SO) of a word are not removed if they are among words that are extracted at the same time.

   In EBCDIC:

   ```
   W1 = '<><. .A. . .B><.C. .D><>'

   SUBWORD(W1,1,1)       ->      '<.A>'
   SUBWORD(W1,1,2)       ->      '<.A. . .B><.C>'
   SUBWORD(W1,3,1)       ->      '<.D>'
   SUBWORD(W1,3)         ->      '<.D>'

   W2 = '<.A. .B><.C><> <.D>'

   SUBWORD(W2,2,1)       ->      '<.B><.C>'
   SUBWORD(W2,2,2)       ->      '<.B><.C><> <.D>'
   ```

Built-in Function Examples

Examples for built-in functions, those that support DBCS and follow the rules defined, are given in this section. For full function descriptions and the syntax diagrams, refer to Chapter 3, "Functions," on page 67.

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
DATATYPE('<.A.B>','S')    ->   1      /* if ETMODE is on */
```

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.

SYMBOL

In EBCDIC:

```
 Drop A.3 ;  <.A.B>=3      /* if ETMODE is on */

 SYMBOL('<.A.B>')    ->    'VAR'
```

```
SYMBOL(<.A.B>)        ->      'LIT'  /* has tested '3' */
SYMBOL('a.<.A.B>')    ->      'LIT'  /* has tested A.3 */
```

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.

VALUE

In EBCDIC:

```
Drop A3  ;  <.A.B>=3  ;  fred='<.A.B>'

VALUE('fred')      ->    '<.A.B>'  /* looks up FRED   */
VALUE(fred)        ->    '3'       /* looks up <.A.B> */
VALUE('a'<.A.B>)   ->    'A3'      /* if ETMODE is on */
```

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
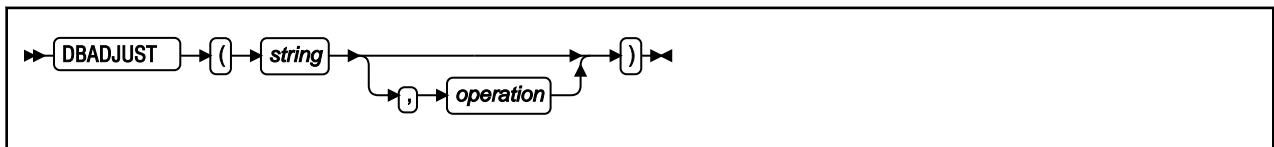
In EBCDIC, adjusts all contiguous SI and SO (or SO and SI) characters in *string* based on the *operation* specified. The following are valid *operation*s. Only the capitalized and highlighted letter is needed; all characters following it are ignored.

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
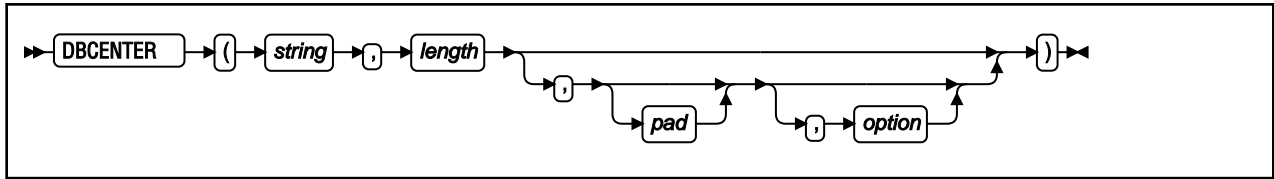
returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up length. The default *pad* character is a blank. If *string* is longer than *length*, it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand 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
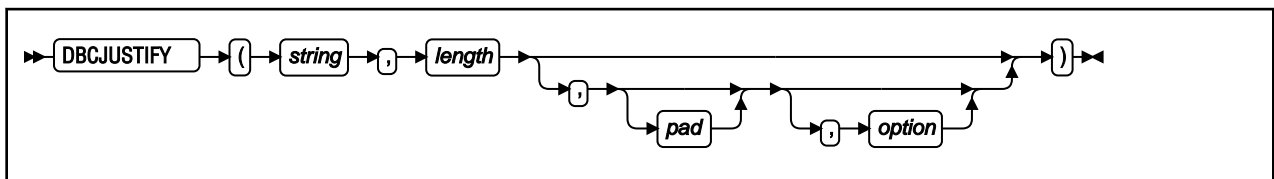
formats *string* by adding *pad* characters between nonblank characters to justify to both margins and length of bytes *length* (*length* must be nonnegative). Rules for adjustments are the same as for the JUSTIFY function. The default *pad* character is a blank.

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
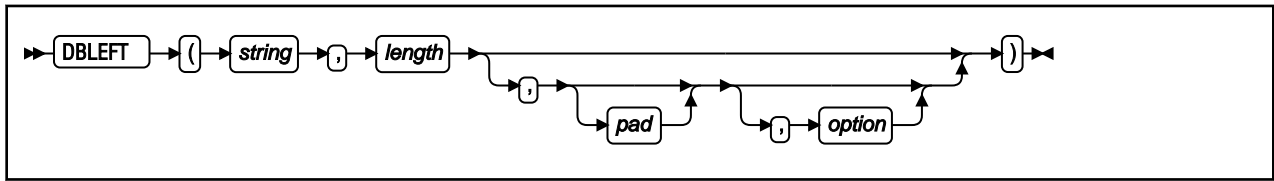
```
►►─ DBLEFT ─►─(─►─ string ─►─,─►─ length ─►──────────────────────────────────►─)─►◄
                                        └─►─,─┬─────────┬─┬────────────────┬─┘
                                              └─ pad ─┘ └─►─,─►─ option ─┘
```

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
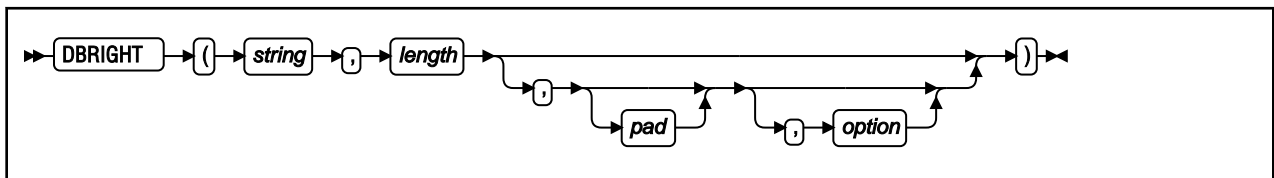
```
►►─ DBRIGHT ─►─(─►─ string ─►─,─►─ length ─►─────────────────────────────────►─)─►◄
                                         └─►─,─┬─────────┬─┬────────────────┬─┘
                                               └─ pad ─┘ └─►─,─►─ option ─┘
```

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

```
►►─ DBRLEFT ─►─(─►─ string ─►─,─►─ length ─►──────────────────►─)─►◄
                                         └─►─,─►─ option ─┘
```
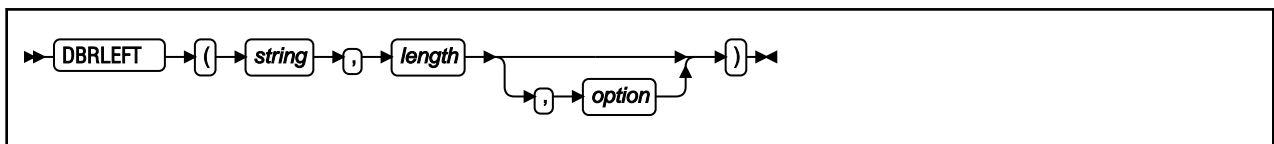
returns the remainder from the DBLEFT function of *string*. If *length* is greater than the length of *string*, returns a null string.

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
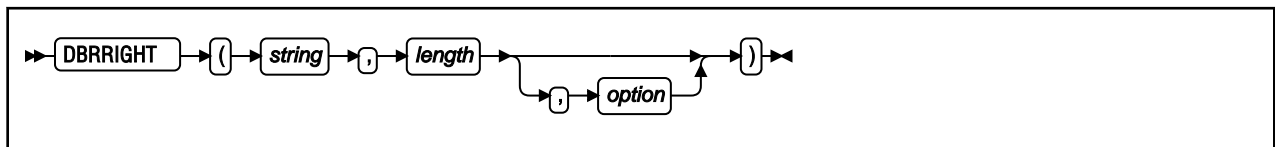
returns the remainder from the DBRIGHT function of *string*. If *length* is greater than the length of *string*, returns a null string.

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
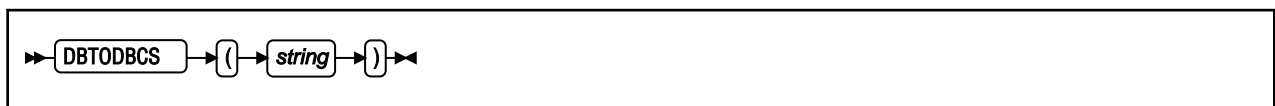
converts all passed, valid SBCS characters (including the SBCS blank) within *string* to the corresponding DBCS equivalents. Other single-byte codes and all DBCS characters are not changed. In EBCDIC, SO and SI brackets are added and removed where appropriate.

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
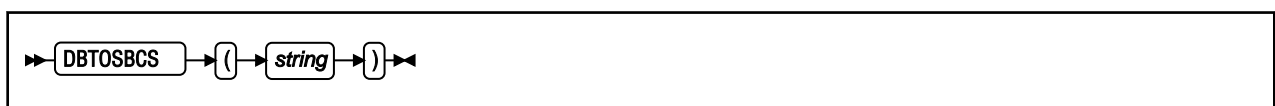
converts all passed, valid DBCS characters (including the DBCS blank) within *string* to the corresponding SBCS equivalents. Other DBCS characters and all SBCS characters are not changed. In EBCDIC, SO and SI brackets are removed where appropriate.

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

```
►►──DBUNBRACKET──(──string──)──►◄
```
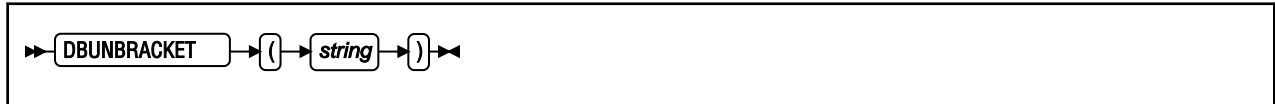
In EBCDIC, removes the SO and SI brackets from a DBCS-only *string* enclosed by SO and SI brackets. If the *string* is not bracketed, a SYNTAX error results.

Here are some EBCDIC examples:

```
DBUNBRACKET('<.A.B>')    ->    '.A.B'
DBUNBRACKET('ab<.A>')    ->    SYNTAX error
```

DBVALIDATE

```
►►──DBVALIDATE──(──string──┬──────────┬──)──►◄
                           └──,──'C'──┘
```
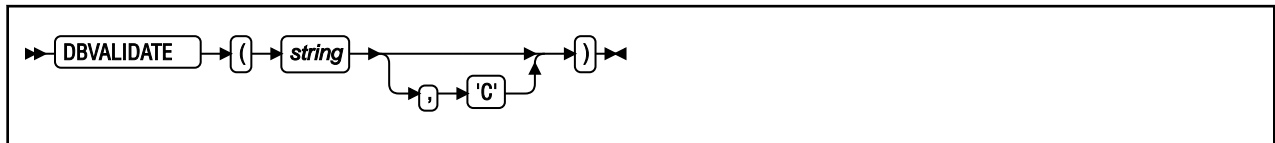
returns 1 if the *string* is a valid mixed string or SBCS string. Otherwise, returns 0. Mixed string validation rules are:

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

```
▶▶──DBWIDTH──(──string──────────────────────▶──)──▶◀
                       └──,──option──┘
```
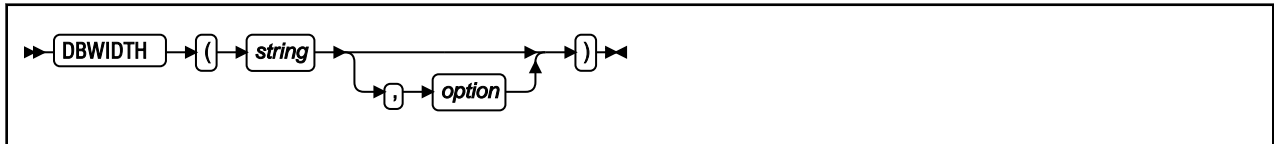
returns the length of *string* in bytes.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBWIDTH('ab<.A.B>','Y')    ->    8
DBWIDTH('ab<.A.B>','N')    ->    6
```

Appendix C. Performance Considerations

Here are some tips to help you code your programs for better performance.

The overhead of including comments on a line with an instruction is negligible except for the storage they take up and the initial read-in time. Comments on a separate line may affect performance, but these may be removed in the executable form by EXECUPDT.

Special information is kept for DO-loops to minimize loop overhead.

Parsing is optimized for mixed case data. PARSE ARG and PARSE PULL are therefore slightly faster than ARG and PULL.

Where possible, the executable form of REXX programs should be in V-format. This minimizes execution time, main storage use (paging), and file space or minidisk space required. (**Note:** If EXECUPDT is used, the library files are F-format but the executable file is V-format.)

Wherever possible, REXX programs should be written in mixed case (especially comments). This maximizes reading speed and minimizes human errors because of misreading data, and so improves the performance of the human side of the REXX programming operation.

There is no particular area in the language processor that can be described as a bottleneck. However, any external call may incur significant system overhead. High precision numbers should be avoided unless truly needed.

Appendix D. Example of a Function Package

```
            TITLE 'USERFN: Sample model for user function package'
*
*
*  The first part of this example deals with obtaining free
*  storage and moving the rest of the program into that storage
*  as a nucleus extension.  The code just loaded (from FREEGO
*  label to the table before FUNC1) then responds to the
*  original call and successive calls to RXUSERFN.  Calls to
*  load a user function are handled by setting up their entry
*  points as nucleus extensions.
*  In order to set up new user functions, the user must add an
*  entry in the FUNLIST table and add the code following the
*  other functions.
*
USERFN    CSECT *
          USING *,R12
          USING NUCON,0
          USING USERSAVE,R13
          LR    R10,R14            Save return address
          SLR   R2,R2             Assume it is NUCEXT
*                                 "RXUSERFN" only.
          CLI   ARG1(R1),X'FF'    Any arguments?
          BE    GOLOAD            Br if not - go install
          CLC   ARG1(8,R1),=CL8'LOAD'   Is this explicit load?
          BNE   BADPL             Br if not - go complain
*         Note:  You  do not have to handle RESET because the
*                package has not yet been loaded
          SPACE 1
*-> LOAD request, so check function name against FUNLIST
          SPACE 1
          LA    R4,LENTRY         Length of FUNLIST entry
          LA    R2,FUNLIST        Start of function table
          LA    R5,EFUNLIST       End of function table
CHECK     EQU   *
          CLC   ARG2(,R1),FUNLNAME(R2) Names match?
          BE    GOLOAD            Br if yes - go do
*                                 appropriate NUCEXTing.
          BXLE  R2,R4,CHECK       Continue testing if more
          LA    R15,1             Indicate function not found
          BR    R10               Not in list - return
          SPACE 1
*=> NUCEXT "RXUSERFN" as well as specific function (for example, if
*   LOAD specified on invocation).
          SPACE 1
GOLOAD    EQU   *
          LA    R0,FREELEND       Length of code in DWs
*                                 Get the storage
          CMSSTOR OBTAIN,DWORDS=(R0),SUBPOOL='NUCLEUS',          X
                ERROR=NOSTORE
          LA    R8,FREEGO         Start of free storage code
          L     R9,=A(FREELEN)    Get length in bytes
          LR    R7,R9             Copy length for MVCL
          LR    R4,R9             Save for later use
          LR    R3,R1                  ""
          LR    R6,R1             Free storage area start
          SPKA  0                 Set nucleus key
          MVCL  R6,R8             Move code to free storage
          NUCEXT SET,NAME='RXUSERFN',ENTRY=(R3),                X
                ORIGIN=((R3),(R4)),KEY=NUCLEUS,SYSTEM=YES,      X
                SERVICE=YES,ERROR=(R10)
*-> See if there is a function...
          LTR   R2,R2             Install "RXUSERFN" only?
          BZR   R10               Br if yes - return to caller
* R2 points to FUNLIST entry to be installed.
* R3 points to start of NUCXLOADed area.
          A     R3,FUNOFFS(,R2)   Calculate true start address
          LA    R2,FUNLNAME(R2)   Address of startup name
          NUCEXT SET,NAME=(R2),ENTRY=(R3),KEY=NUCLEUS,          X
                ORIGIN=(0,0),SYSTEM=YES,SERVICE=NO,ERROR=*
          BR    R10               Return to caller
          DROP  R12
          SPACE 3
          LTORG ,
          TITLE 'USERFN:  Code residing in free storage'
```

```
*-----------------------------------------------------------------*
* The following code resides in free storage, and is capable  *
* of replying to LOAD or RESET.                               *
* A LOAD call results in the identifying of the functions     *
* passed as parameters following LOAD as entry points in      *
* RXUSERFN.                                                   *
* A RESET service call from NUCXDROP will turn the functions  *
* OFF.  A PURGE service call is ignored.                      *
*-----------------------------------------------------------------*
          SPACE 2
FREEGO    DS    0D                  Force doubleword alignment
*                                   of free-loaded code.
          USING *,R12
          B     STARTCOD
          DC    CL8'>USERFN<'       Eye-catcher for storage dump
STARTCOD  EQU   *
          LR    R10,R14             Save return address
          CLC   ARG1(8,R1),=CL8'LOAD'   Is this a load?
          BE    CHK4ARGS            Yes, check for any args
          CLC   ARG1(8,R1),=CL8'RESET'  Reset ?
          BE    DOOFF               Yes, turn off functions
          SLR   R15,R15             In case of service call
          CLI   USECTYP,EPLFABEN    Is it an abend call ?
          BER   R14                 Br if yes - quick quit
          LA    R15,4               No, set error RC
          BR    R14                  .. and return
          SPACE 1
CHK4ARGS  EQU   *
          LA    R15,1               Set possible return code
          CLI   ARG2(R1),X'FF'      Any arguments passed?
          BER   R14                 No, error (already loaded)
*-----------------------------------------------------------------*
* AUTOLOAD: switch on selected function                       *
*-----------------------------------------------------------------*
*                                                             *
* 'LOAD' request.  Check function name against FUNLIST.       *
*                                                             *
*  Only turn on the requested (autoload) function.            *
*-----------------------------------------------------------------*
          SPACE 1
          PUSH  USING               Save USING status
AUTOLOAD  EQU   *
          LR    R3,R1               Save old PLIST pointer
          LR    R1,R13              Get new PLIST address
          LA    R4,LENTRY           Length of FUNLIST entry
          LA    R5,EFUNLIST         End of function table
          LA    R2,FUNLIST          Start of function table
          LA    R15,1               Set error return code
CHECK1    EQU   *
          CLC   ARG2(,R3),FUNLNAME(R2) Check against name
          BE    TURNON              Found - turn function on
          BXLE  R2,R4,CHECK1        Loop for another check
          BR    R10                 Return with RC = 1
          SPACE 1
TURNON    EQU   *
* See if function is already a nucleus extension
          LA    R3,FUNLNAME(R2)     Get startup name
          NUCEXT QUERY,NAME=(R3),ERROR=*
          LTR   R15,R15             Check return code
          BZR   R10                 Exit if already loaded
          L     R6,FUNOFFS(,R2)     Load address offset
          ALR   R6,R12              True start address
          NUCEXT SET,NAME=(R3),ENTRY=(R6),KEY=NUCLEUS,            X
                ORIGIN=(0,0),SYSTEM=YES,SERVICE=NO,ERROR=*
          BR    R10                 Return
          POP   USING               Restore USING status
          SPACE 1
*****************************************************************
* RESET call: switch off functions                            *
*****************************************************************
DOOFF     EQU   *
          LA    R5,FUNLIST          -> to list
FUNLOOP   EQU   *
          DMSLT R15,FUNOFFS(R5)     Any more to cancel?
          BZR   R10                 0 = all done ... Get out
          LA    R3,FUNLNAME(R5)
          NUCEXT CLR,NAME=(R3),ERROR=*
*         (ignore errors, for example: function already canceled)
          LA    R5,LENTRY(,R5)      -> next item in FUNLIST
          B     FUNLOOP
          SPACE 3
*-----------------------------------------------------------------*
```

```
* List of functions included in this pack, with their offsets
FUNLNAME EQU   4,8                   Offset & length of name
FUNOFFS  EQU   0,4                   Offset to the routine
FUNLIST  DC    A(FUNC1-FREEGO),CL8'RXUSER1'
LENTRY   EQU   *-FUNLIST             Length of a single entry
         DC    A(FUNC2-FREEGO),CL8'RXUSER2'
         DC    A(FUNC3-FREEGO),CL8'RXUSER3'
EFUNLIST EQU   *                     End of the funlist proper
         DC    A(*-*)                End fence
*-------------------------------------------------------------*
         EJECT
*+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
*    A sample user written function is shown below.  As many
*    other functions can be added as the user desires.
*
*    The usual order is to obtain an EVALBLOK (here done by
*    the GETBLOK routine), do the function and put the result
*    in the EVALBLOK, and finally to complete the EVALBLOK and
*    return (here done by the EBLOCK routine).
*+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
         SPACE 2
*         'USERFN:  USER1 - User function 1'
*         This function simply returns the first passed parameter!
FUNC1    EQU   *
         USING *,R12                 Tell assembler of base
         LR    R10,R14               Save return address
         LR    R13,R0                Get copy of R0
         USING EFPLIST,R13           Addressing for the PLIST
         L     R11,EARGLIST          Get pointer to arg list
         MVC   SAVEFRET,EFUNRET      Save function return addr
         DROP  R13                   Done with this for now
         USING PARMBLOK,R11          Tell assembler
         L     R1,PARM1LEN           Returned data length
         LR    R3,R1                 Save it for later
         BAL   R14,GETBLOK           Go get EVALBLOK
         USING EVALBLOK,R5           Tell assembler
```

```
****************************************************************
*                                                              *
* other processing for function 1 would be here               *
*                                                              *
****************************************************************
         L     R15,PARM1ADR
         EX    R3,MOVEIT             Move the data
         LA    R15,0                 Set good return code
         B     EBLOCK                Complete EVALBLOK & return
MOVEIT   MVC   EVDATA(0),0(R15)      Move user parm to eval block
         SPACE 2
*         'USERFN:  USER2 - User function 2'
FUNC2    EQU   *
****************************************************************
*                                                              *
* code for user function 2 goes here!                          *
*                                                              *
****************************************************************
         LA    R15,0                 Set RC of 2
         BR    R14                   Return
         SPACE 2
*         'USERFN:  USER3 - User function 3'
FUNC3    EQU   *
****************************************************************
*                                                              *
* code for user function 3 goes here!                          *
*                                                              *
****************************************************************
         LA    R15,0                 Set RC of 3
         BR    R14                   Return
         EJECT
         TITLE 'USERFN: Common get EVALBLOK subroutine'
*-------------------------------------------------------------*
* This subroutine obtains an EVALBLOK.                         *
* The assumed input is:                                        *
*      - R1: length of EVDATA (return data length)             *
*      - R14: return address                                   *
*                                                              *
* The output is:                                               *
*      - R0, R1, & R2 undefined                                *
*      - R4: number of doublewords in entire EVALBLOK          *
*      - R5: address of the EVALBLOK                           *
*      - R15: undefined                                        *
*      - other registers are unchanged.                        *
```

```
*                                                              *
* If storage is not available, an error message is displayed  *
* and return is taken to the caller with a nonzero return     *
* code.                                                        *
*--------------------------------------------------------------*
         SPACE 2
GETBLOK  EQU   *
         BALR  R2,0                  Establish base register
         USING *,R2                  Tell assembler
         LA    R0,EVCTLEN+7(,R1)     Add in overhead + rounding
         C     R1,STORLIM            Compare with storage limit
         BH    NOSTORE               Branch if higher than limit
         SRL   R0,3                  Make it doublewords
         LR    R4,R0                 Return number of doublewords
*                                    in entire EVALBLOK.
         CMSSTOR OBTAIN,DWORDS=(R0),ERROR=NOSTORE  Get the storage
         LR    R5,R1                 Save A(EVALBLOK)
*                                    Now clear the storage block
         LR    R15,R3                Save R3
         LR    R0,R5                 Addr of storage block in R0
         LR    R1,R4                 Length of storage in R1
         SLL   R1,3                  Make it bytes!
         LA    R3,0                  length to 0, pad of '00'x
         MVCL  R0,R2                 Clear the block
         LR    R3,R15                Restore R3
         BR    R14                   Return to caller
         DROP  R2                    Done with this guy
         TITLE 'USERFN:  Common complete EVALBLOK routine'
*--------------------------------------------------------------*
* At this point the EVALBLOK is filled in. The registers       *
* are assumed to be as follows:                                *
* R3 - the number of bytes of data to be returned             *
* R4 - the size (in doublewords) of the entire EVALBLOK        *
* R5 - the address of the EVALBLOK                             *
*--------------------------------------------------------------*
         SPACE 1
EBLOCK   EQU   *
         BALR  R12,0                 Set base register
         USING *,R12                 Tell assembler
         USING EVALBLOK,R5           Addressing for EVALBLOK
         ST    R4,EVSIZE             Total block size (DW's)
         L     R4,SAVEFRET           Get back return address
         ST    R5,0(R4)              Pass address back to caller
         ST    R3,EVLEN              Set it in EVALBLOK
         BR    R10                   Abandon ship
         DROP  R5
         TITLE 'Common Error Processing Routines'
*--------------------------------------------------------------*
* Error handling routines.                                     *
*--------------------------------------------------------------*
         SPACE 3
BADPL    EQU   *                     Something's wrong with PLIST
         BALR  R12,0                 Load base for this code
         USING *,R12                 Tell assembler of this
         LA    R2,MSG1               Get message address
         B     DISPMSG               Go display the message
         SPACE 1
NOSTORE  EQU   *                     CMSSTORE not successful
         BALR  R12,0                 Load base for this code
         USING *,R12                 Tell assembler of this
         LA    R2,MSG2               Get message address
DISPMSG  EQU   *
         BALR  R12,0                 Load base for this code
         USING *,R12                 Tell assembler of this
         LR    R1,R13                Use USERSAVE for PLIST
         APPLMSG APPLID=USR,TEXTA=(R2)
NODISPL1 EQU   *
         LA    R15,4                 Set nonzero return code
         BR    R10                   Return
         SPACE 1
MSG1     DC    AL1(L'MSG1TXT)
MSG1TXT  DC    C'DMSRUF070E Invalid parameter'
         SPACE
MSG2     DC    AL1(L'MSG2TXT)
MSG2TXT  DC    C'DMSRUF450E Machine storage exhausted or request exceedX
               s limit'
         SPACE 2
STORLIM  DC    F'16777216'           Constant of 16MB
SAVEFRET DS    F                     Function return address
         ORG   ,
         SPACE 2
         LTORG                       Literal pool
```

```
         TITLE 'USERFN:  Common symbolic assignments'
         SPACE 1
ARG1     EQU   8,8                   First argument
ARG2     EQU   16,8                  Second argument
         REGEQU
         DS    0D                    Get to doubleword boundary
FREELEN  EQU   *-FREEGO              Bytes of free store code.
FREELEND EQU   (*-FREEGO+7)/8        Doublewords of free store
*                                    code.
         SPACE 1
* NUCEXT PLIST Flags:
SERVICE  EQU   X'40'
SYSTEM   EQU   X'80'
         SPACE 2
*-- DSECT for the function PLIST ------------------------------
EFPLIST  DSECT
ECOMVERB DS    F                     COMVERB pointer
EBEGARGS DS    F                     pointer to argument string
EENDARGS DS    F                     pointer to arg string end
EFBLOCK  DS    F                     fileblock pointer (0)
EARGLIST DS    F                     pointer to function args
EFUNRET  DS    F                     location of return data
*-- DSECT for the returned data block ------------------------
EVALBLOK DSECT
EVBPAD1  DS    F                     Reserved
EVSIZE   DS    F                     Total block size in DW's
EVLEN    DS    F                     Length of Data (in bytes)
EVBPAD2  DS    F                     Reserved
EVCTLEN  EQU   *-EVALBLOK            Length of preceding section
EVDATA   DS    0D                    First byte of data
EVDATAW1 DS    F                     First word of data
EVDATAW2 DS    F                     Second word of data
EVDATAW3 DS    F                     Third word of data
EVDATAW4 DS    F                     Fourth word of data
EVDATAW5 DS    F                     Fifth word of data
         SPACE 3
*-- DSECT for input parameters  -----------------------------*
PARMBLOK DSECT
PARM1ADR DS    F                     Address of parameter 1
PARM1LEN DS    F                     Length of parameter 1
PARMNTRY EQU   *-PARMBLOK            Length of table entry
PARM2ADR DS    F                     Address of parameter 2
PARM2LEN DS    F                     Length of parameter 2
PARM3ADR DS    F                     Address of parameter 3
PARM3LEN DS    F                     Length of parameter 3
PARM4ADR DS    F                     Address of parameter 4
PARM4LEN DS    F                     Length of parameter 4
PARM5ADR DS    F                     Address of parameter 5
PARM5LEN DS    F                     Length of parameter 5
PADR     EQU   0,4                   Offset in each pair to
*                                    parameter's address.
PLEN     EQU   4,4                   Offset in each pair to
*                                    parameter's length.
         SPACE 3
         USERSAVE
         EPLIST
         PRINT NOGEN                 No need to see NUCON....
         NUCON
         END
```

Appendix E. z/VM REXX/VM Interpreter in the GCS Environment

Most REXX capabilities available in the CMS environment are also available in the Group Control System (GCS) environment. You can use the REXX instructions, functions, expressions, operators, and so forth. There are, however, some differences between writing REXX programs for the GCS environment and writing REXX programs for the CMS environment.

The differences in the GCS environment are as follows:

1. Execs usually reside in CMS formatted disk files and have a file type of GCS. The GCS file type can be overridden by using the file block.

2. GCS does not support the following immediate commands: TS, TE, and HI.

3. An exec written for the GCS environment should not have the same name as an immediate command. (Note that an immediate command lets you interrupt a program and halt its execution either temporarily or permanently.) Immediate commands are higher in the search order; therefore, an immediate command would be run before an exec. An exec written for the GCS environment with the same name as an immediate command would never be run.

4. GCS does not support the external function libraries: RXSYSFN, RXLOCFN, and RXUSERFN. However, GCS does support external function calls. These functions and subroutines must be written in the REXX language.

5. The GCS CMDSI macro can be used to call REXX programs from Assembler language programs. The FILEBLK parameter on the CMDSI macro contains the address of the file block. FILEBLK is useful for executing in-storage execs, executing execs with file types other than GCS, and establishing an initial subcommand environment.

6. The default ADDRESS environment of REXX is GCS.

   ADDRESS GCS specifies that full command resolution is in effect. With full command resolution, first search for an exec with the given name. If such an exec does not exist, then call the given name using SVC 202. If the above fails, search for a CP command with the given name.

   ADDRESS COMMAND searches for host commands (GCS commands).

7. GCS does not have a terminal input buffer. If you enter a PULL instruction and the program stack is empty, the WTOR macro generates a read to the console.

8. Each task has its own program stack. Therefore, data in a program stack can be shared among execs running in the same task.

9. To specify other subcommand environments in GCS you must use LOADCMD. LOADCMD defines a command name to the requested module of a CMS load library and loads this command module into storage. Therefore, GCS can call the requested command module when a command is entered at the console or submitted by a program with the CMDSI macro.

10. The SIGNAL ON HALT instruction has no effect in GCS.

11. GCS does not support the third parameter (*selector*) on a call to the VALUE function.

12. GCS does not support the REXX Level 2 Stream I/O functions. If the user has an exec with the same name as one of the REXX Stream I/O functions, the exec will be run.

For information about exits in the GCS environment, see "REXX Exits" on page 195.

The Extended PLIST (EPLIST)

The EPLIST macro generates a DSECT for the GCS extended parameter list.

The format of the EPLIST macro is:
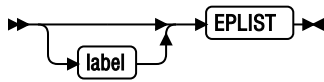
where

label
    is an optional assembler label for the statement.

The first statement in the EPLIST macro expansion is labeled EPLIST. For the format of the EPLIST macro, refer to the data areas and control blocks information in the IBM z/VM Internet Library at IBM: z/VM Internet Library (www.ibm.com/vm/library).

The Standard Tokenized PLIST (PLIST)

The standard tokenized PLIST has the following format:

```
        DC    CL8'EXEC'
        DC    CL8'execname'
        DC    XL8'FF'
```

The File Block (FBLOCK)

For the format of the FBLOCK macro, refer to the data areas and control blocks information in the IBM z/VM Internet Library at IBM: z/VM Internet Library (www.ibm.com/vm/library).

EXECCOMM Processing (Sharing Variables)

The EXECCOMM macro allows programs to access and manipulate the current generation of REXX variables. For a description of EXECCOMM, see *z/VM: Group Control System*.

Shared Variable Request Block (SHVBLOCK)

If the address of the shared variable request block passed in register 0 is incorrect, the task is terminated with abend code FCB and reason code 0D01. Each request block in the chain must be structured as indicated in the SHVBLOCK macro refer to the data areas and control blocks information in the IBM z/VM Internet Library at IBM: z/VM Internet Library (www.ibm.com/vm/library).

A typical calling sequence using the EXECCOMM macro is:

```
EXECCOMM REQLIST=(5)
```

where register 5 points to the first of a chain of one or more request blocks.

Function Codes (SHVCODE)

For the format of the SHVCODE macro, refer to the data areas and control blocks information in the IBM z/VM Internet Library at IBM: z/VM Internet Library (www.ibm.com/vm/library).

RXITDEF Processing (Assigning Values for Exits)

The RXITDEF macro assigns the correct values to the symbols used for the exit routine function and subfunction codes. You can use this macro for CMS and GCS programs.

The format of the RXITDEF macro is:

```
RXITDEF
```

For more information on using this macro with the REXX exits, see "REXX Exits" on page 195.

This macro assigns the following symbols:

```
RXFNC    EQU   X'0002'          Process a function request
RXFNCCAL EQU            X'0001'  FNC Call a function/subroutine
RXCMD    EQU   X'0003'          Process a command request
RXCMDHST EQU            X'0001'  CMD Process a host command request
RXMSQ    EQU   X'0004'          Manipulate the session queue
RXMSQPLL EQU            X'0001'  MSQ Pull an entry from queue
RXMSQPSH EQU            X'0002'  MSQ Push an entry onto queue
RXMSQSIZ EQU            X'0003'  MSQ Determine the queue size
RXSIO    EQU   X'0005'          Perform session Input/Output
RXSIOSAY EQU            X'0001'  SIO Output a SAY string
RXSIOTRC EQU            X'0002'  SIO Output a TRACE string
RXSIOTRD EQU            X'0003'  SIO Terminal read
RXSIODTR EQU            X'0004'  SIO Debug terminal read
RXSIOTLL EQU            X'0005'  SIO determine line length
RXMEM    EQU   X'0006'          Memory management services
RXMEMGET EQU            X'0001'  MEM Get memory
RXMEMRET EQU            X'0002'  MEM Return memory
RXHLT    EQU   X'0007'          Halt services
RXHLTCLR EQU            X'0001'  HLT Clear the halt status
RXHLTTST EQU            X'0002'  HLT Test the halt status
RXTRC    EQU   X'0008'          Test the TRACE status
RXTRCTST EQU            X'0001'  TRC Test the TRACE status
RXINI    EQU   X'0009'          Initialization service
RXINIEXT EQU            X'0001'  INI Initialization exit
RXTER    EQU   X'000A'          Termination service
RXTEREXT EQU            X'0001'  TER Termination exit
```

RXITPARM Processing (Mapping Parameter List for Exits)

The RXITPARM macro maps the parameter list used to pass information between the language processor and an exit routine. You can use this macro for CMS and GCS programs.

The format of the RXITPARM macro is:

```
RXITPARM
```

For more information on using this macro with the REXX exits, see "REXX Exits" on page 195. For the format of the RXITPARM macro and which parameters are used for the exit routines, refer to the data areas and control blocks information in the IBM z/VM Internet Library at IBM: z/VM Internet Library (www.ibm.com/vm/library).

Appendix F. Input and Output Return and Reason Codes

The z/VM implementation of input and output uses CSL routines to perform I/O. The return codes and reason codes from the higher level internal routines are documented in this table.

Table 11: Variables and Their Possible Values	
Variable Name	**Integer and Pseudonym Values**
reascode	**0** Task successfully completed. **90101** Data truncated. **96100** Insufficient storage. **99504** Incorrect open intent for the specified object. (For example, WRITE was specified, but the stream cannot be written to.) **99505** Incorrect length specified. **99508** Object not defined. **99509** End of object encountered. **99510** System program error. **99511** Error releasing storage. **99512** System program error. **99513** System program error. **99514** System program error. **99515** Incorrect object name. **99518** System program error. **99520** No unique IDs available. **99522** No information available. **99524** Object already exists. **99525** Object already open. **99526** Object not open. **99527** Object is read only. **99528** Object was not found.

Table 11: Variables and Their Possible Values (continued)	
Variable Name	**Integer and Pseudonym Values**

reascode (continued)	**99529**
	    Object is not readable.
	99530
	    Object specific error. This is usually followed by a lower level routine name and the return and reason codes from this lower level routine.
	99532
	    Pointer is not movable.
	99533
	    Pointer is out of range.

retcode	**0**
	    Task successfully completed.
	4
	    Warning
	8
	    Error

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain services of z/VM.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at IBM copyright and trademark information - United States (www.ibm.com/legal/us/en/copytrade.shtml).

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Terms and Conditions for Product Documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal Use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM Online Privacy Statement Highlights at http://www.ibm.com/privacy and the IBM Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies", and the IBM Software Products and Software-as-a-Service Privacy Statement at http://www.ibm.com/software/info/product-privacy.

Bibliography

This topic lists the publications in the z/VM library. For abstracts of the z/VM publications, see *z/VM: General Information*.

Where to Get z/VM Information

The current z/VM product documentation is available in IBM Knowledge Center - z/VM (www.ibm.com/support/knowledgecenter/SSB27U).

z/VM Base Library

Overview

- *z/VM: License Information*, GI13-4377
- *z/VM: General Information,* GC24-6286

Installation, Migration, and Service

- *z/VM: Installation Guide*, GC24-6292
- *z/VM: Migration Guide*, GC24-6294
- *z/VM: Service Guide*, GC24-6325
- *z/VM: VMSES/E Introduction and Reference*, GC24-6336

Planning and Administration

- *z/VM: CMS File Pool Planning, Administration, and Operation*, SC24-6261
- *z/VM: CMS Planning and Administration*, SC24-6264
- *z/VM: Connectivity*, SC24-6267
- *z/VM: CP Planning and Administration*, SC24-6271
- *z/VM: Getting Started with Linux on IBM Z*, SC24-6287
- *z/VM: Group Control System*, SC24-6289
- *z/VM: I/O Configuration*, SC24-6291
- *z/VM: Running Guest Operating Systems*, SC24-6321
- *z/VM: Saved Segments Planning and Administration*, SC24-6322
- *z/VM: Secure Configuration Guide,* SC24-6323
- *z/VM: TCP/IP LDAP Administration Guide*, SC24-6329
- *z/VM: TCP/IP Planning and Customization*, SC24-6331
- z/OS and z/VM: Hardware Configuration Manager User's Guide (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sc342670/$file/eequ100_v2r3.pdf), SC34-2670

Customization and Tuning

- *z/VM: CP Exit Customization*, SC24-6269
- *z/VM: Performance*, SC24-6301

Operation and Use

- *z/VM: CMS Commands and Utilities Reference*, SC24-6260
- *z/VM: CMS Primer*, SC24-6265
- *z/VM: CMS User's Guide*, SC24-6266
- *z/VM: CP Commands and Utilities Reference*, SC24-6268
- *z/VM: System Operation*, SC24-6326
- *z/VM: TCP/IP User's Guide*, SC24-6333
- *z/VM: Virtual Machine Operation*, SC24-6334
- *z/VM: XEDIT Commands and Macros Reference*, SC24-6337
- *z/VM: XEDIT User's Guide*, SC24-6338

Application Programming

- *z/VM: CMS Application Development Guide*, SC24-6256
- *z/VM: CMS Application Development Guide for Assembler*, SC24-6257
- *z/VM: CMS Application Multitasking*, SC24-6258
- *z/VM: CMS Callable Services Reference*, SC24-6259
- *z/VM: CMS Macros and Functions Reference*, SC24-6262
- *z/VM: CMS Pipelines User's Guide and Reference*, SC24-6252
- *z/VM: CP Programming Services*, SC24-6272
- *z/VM: CPI Communications User's Guide*, SC24-6273
- *z/VM: ESA/XC Principles of Operation*, SC24-6285
- *z/VM: Language Environment User's Guide*, SC24-6293
- *z/VM: OpenExtensions Advanced Application Programming Tools*, SC24-6295
- *z/VM: OpenExtensions Callable Services Reference*, SC24-6296
- *z/VM: OpenExtensions Commands Reference*, SC24-6297
- *z/VM: OpenExtensions POSIX Conformance Document*, GC24-6298
- *z/VM: OpenExtensions User's Guide*, SC24-6299
- *z/VM: Program Management Binder for CMS*, SC24-6304
- *z/VM: Reusable Server Kernel Programmer's Guide and Reference*, SC24-6313
- *z/VM: REXX/VM Reference*, SC24-6314
- *z/VM: REXX/VM User's Guide*, SC24-6315
- *z/VM: Systems Management Application Programming*, SC24-6327
- *z/VM: TCP/IP Programmer's Reference*, SC24-6332
- *CPI Communications Reference*, SC26-4399
- *Common Programming Interface Resource Recovery Reference*, SC31-6821
- z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS (www.ibm.com/servers/resourcelink/ svc00100.nsf/pages/zosv2r3sa760169/$file/glpa300_v2r3.pdf), SA76-0169
- z/OS: Language Environment Concepts Guide (www.ibm.com/servers/resourcelink/svc00100.nsf/ pages/zosv2r3sa380687/$file/ceea800_v2r3.pdf), SA38-0687
- z/OS: Language Environment Debugging Guide (www.ibm.com/servers/resourcelink/svc00100.nsf/ pages/zosv2r3ga320908/$file/ceea100_v2r3.pdf), GA32-0908
- z/OS: Language Environment Programming Guide (www.ibm.com/servers/resourcelink/svc00100.nsf/ pages/zosv2r3sa380682/$file/ceea200_v2r3.pdf), SA38-0682
- z/OS: Language Environment Programming Reference (www.ibm.com/servers/resourcelink/ svc00100.nsf/pages/zosv2r3sa380683/$file/ceea300_v2r3.pdf), SA38-0683

- z/OS: Language Environment Runtime Messages (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sa380686/$file/ceea900_v2r3.pdf), SA38-0686
- z/OS: Language Environment Writing Interlanguage Communication Applications (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sa380684/$file/ceea400_v2r3.pdf), SA38-0684
- z/OS: MVS Program Management Advanced Facilities (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sa231392/$file/ieab200_v2r3.pdf), SA23-1392
- z/OS: MVS Program Management User's Guide and Reference (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sa231393/$file/ieab100_v2r3.pdf), SA23-1393

Diagnosis

- *z/VM: CMS and REXX/VM Messages and Codes*, GC24-6255
- *z/VM: CP Messages and Codes*, GC24-6270
- *z/VM: Diagnosis Guide*, GC24-6280
- *z/VM: Dump Viewing Facility*, GC24-6284
- *z/VM: Other Components Messages and Codes*, GC24-6300
- *z/VM: TCP/IP Diagnosis Guide*, GC24-6328
- *z/VM: TCP/IP Messages and Codes*, GC24-6330
- *z/VM: VM Dump Tool*, GC24-6335
- z/OS and z/VM: Hardware Configuration Definition Messages (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sc342668/$file/cbdm100_v2r3.pdf), SC34-2668

z/VM Facilities and Features

Data Facility Storage Management Subsystem for VM

- *z/VM: DFSMS/VM Customization*, SC24-6274
- *z/VM: DFSMS/VM Diagnosis Guide*, GC24-6275
- *z/VM: DFSMS/VM Messages and Codes*, GC24-6276
- *z/VM: DFSMS/VM Planning Guide*, SC24-6277
- *z/VM: DFSMS/VM Removable Media Services*, SC24-6278
- *z/VM: DFSMS/VM Storage Administration*, SC24-6279

Directory Maintenance Facility for z/VM

- *z/VM: Directory Maintenance Facility Commands Reference*, SC24-6281
- *z/VM: Directory Maintenance Facility Messages*, GC24-6282
- *z/VM: Directory Maintenance Facility Tailoring and Administration Guide*, SC24-6283

Open Systems Adapter

- Open Systems Adapter-Express Customer's Guide and Reference (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sa227935/$file/ioaz100_v2r3.pdf), SA22-7935
- Open Systems Adapter-Express Integrated Console Controller User's Guide (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sc279003/$file/ioaq100_v2r3.pdf), SC27-9003
- Open Systems Adapter-Express Integrated Console Controller 3215 Support (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sa232247/$file/ioan100_v2r3.pdf), SA23-2247
- Open Systems Adapter/Support Facility on the Hardware Management Console (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sc147580/$file/ioas100_v2r3.pdf), SC14-7580

Performance Toolkit for VM

- *z/VM: Performance Toolkit Guide*, SC24-6302
- *z/VM: Performance Toolkit Reference*, SC24-6303

RACF® Security Server for z/VM

- *z/VM: RACF Security Server Auditor's Guide*, SC24-6305
- *z/VM: RACF Security Server Command Language Reference,* SC24-6306
- *z/VM: RACF Security Server Diagnosis Guide,* GC24-6307
- *z/VM: RACF Security Server General User's Guide,* SC24-6308
- *z/VM: RACF Security Server Macros and Interfaces*, SC24-6309
- *z/VM: RACF Security Server Messages and Codes*, GC24-6310
- *z/VM: RACF Security Server Security Administrator's Guide*, SC24-6311
- *z/VM: RACF Security Server System Programmer's Guide*, SC24-6312
- *z/VM: Security Server RACROUTE Macro Reference,* SC24-6324

Remote Spooling Communications Subsystem Networking for z/VM

- *z/VM: RSCS Networking Diagnosis*, GC24-6316
- *z/VM: RSCS Networking Exit Customization*, SC24-6317
- *z/VM: RSCS Networking Messages and Codes*, GC24-6318
- *z/VM: RSCS Networking Operation and Use*, SC24-6319
- *z/VM: RSCS Networking Planning and Configuration*, SC24-6320
- z/OS: Network Job Entry (NJE) Formats and Protocols (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3sa320988/$file/hasa600_v2r3.pdf), SA32-0988

Prerequisite Products

Device Support Facilities

- Device Support Facilities (ICKDSF): User's Guide and Reference (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3gc350033/$file/ickug00_v2r3.pdf), GC35-0033

Environmental Record Editing and Printing Program

- Environmental Record Editing and Printing Program (EREP): Reference (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3gc350152/$file/ifc2000_v2r3.pdf), GC35-0152
- Environmental Record Editing and Printing Program (EREP): User's Guide (www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r3gc350151/$file/ifc1000_v2r3.pdf), GC35-0151

Index

M

macro
    editor 24
    EPLIST 303
    REXEXIT 204
    RXITDEF in GCS 304
    RXITPARM in GCS 305
    XEDIT interface 20
MAKEBUF
    creating additional buffers 54, 55
    description 211
mapping parameter lists for exits, RXITPARM macro 305
MAX function
    description 92
    example 92
    implementation maximum 92
memory
    accessing 135
    finding upper limit of 135
message examples, notation used in xviii
messages, error 277
MIN function
    description 92
    example 92
    implementation maximum 92
minutes calculated from midnight 108
mixed DBCS string 80
model of input and output 167
Month option of DATE function 82
multi-way call 30, 60
multiple
    argument passing 184
    assignments in parsing 142
    string parsing 142, 146
multiplication
    description 155
    operator 8

N

names
    CMSMIXED, environment 25
    of execs 49
    of functions 67
    of programs 49
    of subroutines 29
    of variables 4
negation
    of logical values 10
    of numbers 8
nesting of control structures 31
nibbles 4
NOETMODE 46
NOEXMODE 46
NOP instruction
    description 43
    example 43
Normal option of DATE function 82
not equal operator 9
not greater than operator 9
not less than operator 9
NOT operator 5, 10
notation

notation *(continued)*
    engineering 159
    exponential, example 159
    scientific 159
notation used in message and response examples xviii
note
    condition traps 163
    to CMS EXEC and EXEC 2 users 28
NOTREADY condition
    condition trapping 173
    raised by stream errors 173
    SIGNAL and CALL instructions 164
NOTYPING flag cleared before error messages 277
NOVALUE condition
    not raised by VALUE function 112
    of SIGNAL and CALL instructions 164
    on SIGNAL instruction 161
    use of 209
null
    clauses 12
    strings 3, 7
number from stream 77
numbers
    arithmetic on 8, 153, 154
    checking with DATATYPE 80
    comparison of 9, 157
    description 5, 153, 154
    formatting for display 87
    implementation maximum 5
    in DO instruction 32
    truncating 110
    use in the language 160
    whole 160
numeric
    comparisons, example 158
    options in TRACE 63
NUMERIC instruction
    description 44
    DIGITS option 44
    FORM option 44, 159
    FUZZ option 44
    option of PARSE instruction 49, 160
    settings saved during subroutine calls 30

O

online HELP Facility, using 21
Open and Query Spool File Characteristics, DIAGRC(BC) 132
OPENVM routines 217
operations
    arithmetic 155
    tracing results 61
operator
    arithmetic
        description 7, 153, 154
        list 8
    as special characters 5
    characters
        description 5
        implementation maximum 15
    comparison 9, 157
    concatenation 8
    examples 156, 157
    logical 10

X

X2B function
    description 114
    example 114
X2C function
    description 114
    example 115
X2D function
    description 115
    example 115
    implementation maximum 115
XA flag of CMSFLAG function 120
XC flag of CMSFLAG function 120
XEDIT macro interface 20
XOR, logical 10
XORing character strings together 74
XRANGE function
    description 114
    example 114

Y

YEAR2000 flag of CMSFLAG function 120

Z

Z flag of CMSFLAG function 120
z/VM
    HELP Facility, using 21
    unique functions 118
    unique routines 118
zeros
    added on the left 94
    removal with STRIP function 106

IBM