Object REXX for Windows

Reference

Version 2.1

SH12-6725-00

Object REXX for Windows

Reference

Version 2.1

SH12-6725-00

Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix D. Notices" on page 547.

First Edition, March 2001

This edition applies to Version 2.1 of IBM Object REXX for Windows Interpreter Edition (5639-M69) and Development Edition (5639-M68), and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

This edition replaces SH12-6222-03.

Contents

About This Book

This book describes the Object REXX Interpreter, called **interpreter** or **language processor** in the following, and the Object-Oriented REstructured eXtended eXecutor (**REXX**) language.

This book is intended for people who plan to develop applications using REXX. Its users range from the novice, who might have experience in some programming language but no REXX experience, to the experienced application developer, who might have had some experience with Object REXX.

This book is a reference rather than a tutorial. It assumes you are already familiar with object-oriented programming concepts.

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

Related Information

Object REXX for Windows: Programming Guide, SH12-6726

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other REXX documentation:

- Visit our home page at `http://www.ibm.com/software/ad/obj-rexx/support.html#Buy or get support`. There you can access the Internet Online Form where you can enter comments and send them.
- Send your comments by e-mail to `swsdid@de.ibm.com`. Be sure to include the name of the book, the part number of the book, the version of REXX, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).
- Fill out one of the forms at the back of this book and return it by mail, by fax, or by giving it to an IBM representative. The mailing address is on the back of the Readers' Comments form. The fax number is +49-(0)7031-16-4892.

How to Read the Syntax Diagrams

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

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

  The ►►── symbol indicates the beginning of a statement.

  The ──► symbol indicates that the statement syntax is continued on the next line.

  The ►── symbol indicates that a statement is continued from the previous line.

  The ──►◄ symbol indicates the end of a statement.

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

```
►►──STATEMENT─required_item────────────────────────────────────────►◄
```

- Optional items appear below the main path.

```
►►──STATEMENT──┬───────────────┬───────────────────────────────────►◄
               └─optional_item─┘
```

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

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

```
►►──STATEMENT──┬─required_choice1─┬────────────────────────────────►◄
               └─required_choice2─┘
```

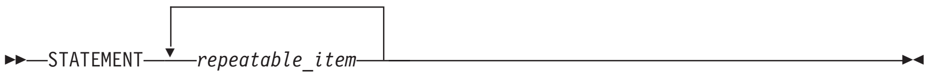
- If choosing one of the items is optional, the entire stack appears below the main path.

```
►►──STATEMENT──┬──────────────────┬────────────────────────────────►◄
               ├─optional_choice1─┤
               └─optional_choice2─┘
```

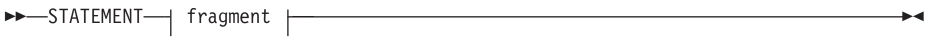
- If one of the items is the default, it appears above the main path and the remaining choices are shown below.

```
               ┌─default_choice──┐
►►──STATEMENT──┼─────────────────┼─────────────────────────────────►◄
               ├─optional_choice─┤
               └─optional_choice─┘
```

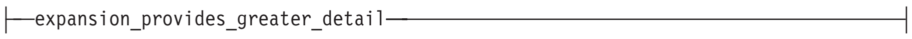
- An arrow returning to the left above the main line indicates an item that can be repeated.



  A repeat arrow above a stack indicates that you can repeat the items in the stack.
- A set of vertical bars around an item indicates that the item is a *fragment*, a part of the syntax diagram that appears in greater detail below the main diagram.

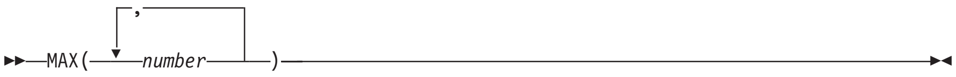


fragment:



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

The following example shows how the syntax is described:



Chapter 1. REXX General Concepts

The REXX language is particularly suitable for:
- Application scripting
- Command procedures
- Application front ends
- User-defined macros (such as editor subcommands)
- Prototyping
- Personal computing

As an object-oriented language, REXX provides, for example, data encapsulation, polymorphism, an object class hierarchy, class-based inheritance of methods, and concurrency. Object REXX is compatible with earlier REXX versions. It has the usual structured-programming instructions, for example IF, SELECT, DO WHILE, and LEAVE, and a number of useful built-in functions.

The language imposes few restrictions on the 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. There are no restrictions on the types of data that variables can contain.

The limit on the length of symbols (variable names) is 250 characters. You can use compound symbols, such as

```
NAME.Y.Z
```

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

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

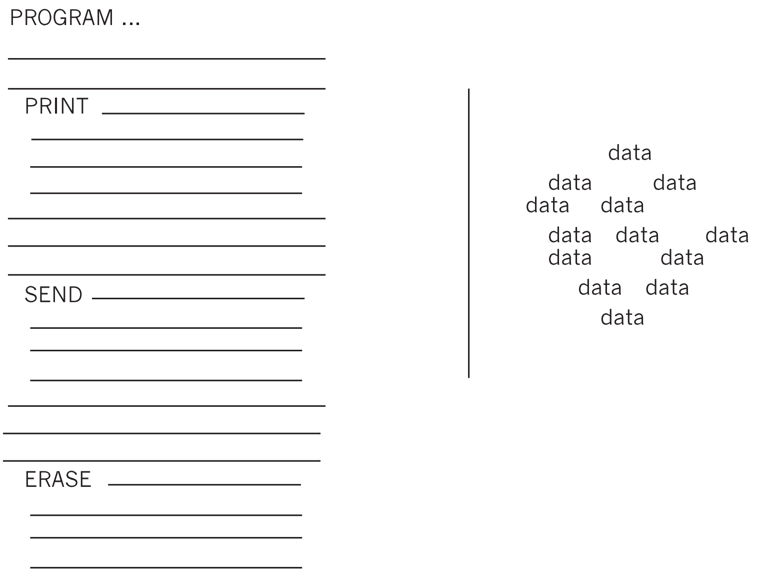
What Is Object-Oriented Programming?

Object-oriented programming is a way to write computer programs by focusing not on the instructions and operations a program uses to manipulate data, but on the data itself. First, the program simulates, or models, objects in the physical world as closely as possible. Then the objects interact with each other to produce the desired result.

Real-world objects, such as a company's employees, money in a bank account, or a report, are stored as data so the computer can *act* upon it. For example, when you print a report, print is the action and report is the object acted upon. Often several actions apply; you could also send or erase the report.

Modularizing Data

In conventional, structured programming, actions like print are often isolated from the data by placing them in subroutines or modules. A module typically contains an operation for implementing one simple action. You might have a PRINT module, a SEND module, an ERASE module. These actions are independent of the data they operate on.



But with object-oriented programming, it is the data that is modularized. And each data module includes its own operations for performing actions directly related to its data.

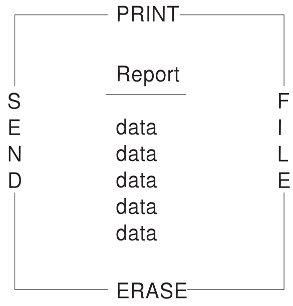


Figure 1. Modular Data—a Report Object

In the case of report, the report object would contain its own built-in PRINT, SEND, ERASE, and FILE operations.

Object-oriented programming lets you model real-world objects—even very complex ones—precisely and elegantly. As a result, object manipulation becomes easier and computer instructions become simpler and can be modified later with minimal effort.

Object-oriented programming *hides* any information that is not important for acting on an object, thereby concealing the object's complexities. Complex tasks can then be initiated simply, at a very high level.

Modeling Objects

In object-oriented programming, objects are modeled to real-world objects. A real-world object has actions related to it and characteristics of its own.

Take a ball, for example. A ball can be acted on—rolled, tossed, thrown, bounced, caught. But it also has its own physical characteristics—size, shape, composition, weight, color, speed, position. An accurate data model of a real ball would define not only the physical characteristics but *all* related actions and characteristics in one package:

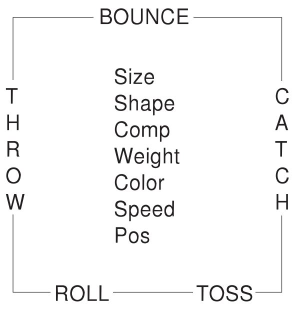


Figure 2. A Ball Object

In object-oriented programming, objects are the basic building blocks—the fundamental units of data.

There are many kinds of objects; for example, character strings, collections, and input and output streams. An object—such as a character string—always consists of two parts: the possible actions or operations related to it, and its characteristics or variables. A variable has a variable *name,* and an associated data value that can change over time. These actions and characteristics are so closely associated that they cannot be separated:

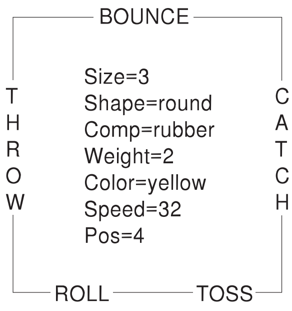


Figure 3. Ball Object with Variable Names and Values

To access an object's data, you must always specify an action. For example, suppose the object is the number 5. Its actions might include addition, subtraction, multiplication, and division. Each of these actions is an interface to the object's data. The data is said to be *encapsulated* because the only way to access it is through one of these surrounding actions. The encapsulated internal characteristics of an object are its *variables*. Variables are associated with an object and exist for the lifetime of that object:

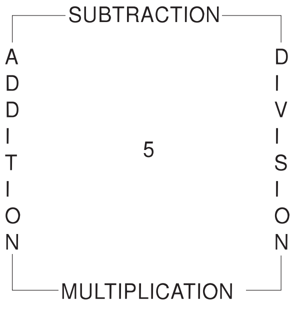


Figure 4. Encapsulated 5 Object

REXX comes with a basic set of classes for creating objects (see “Chapter 4. Objects and Classes” on page 95). Therefore, you can create objects that exactly match the needs of a particular application.

How Objects Interact

The actions within an object are its only interface to other objects. Actions form a kind of “wall” that encapsulates the object, and shields its internal information from outside objects. This shielding is called *information hiding.* Information hiding protects an object’s data from corruption by outside objects, and also protects outside objects from relying on another object’s private data, which can change without warning.

One object can act upon another (or cause it to act) only by calling that object’s actions, namely by sending *messages.* Objects respond to these messages by performing an action, returning data, or both. A message to an object must specify:

- A receiving object
- The “message send” symbol, ~, which is called the *twiddle*
- The action and, optionally in parentheses, any parameters required

So the message format looks like this:

```
object~action(parameters)
```

Assume that the object is the string `!iH`. Sending it a message to use its REVERSE action:

```
'!iH'~reverse
```

returns the string object `Hi!`.

Methods

Sending a message to an object results in performing some action; that is, it results in running some underlying code. The action-generating code is called a *method*. When you send a message to an object, you specify its method name in the message. Method names are character strings like REVERSE. In the preceding example, sending the `reverse` message to the `!iH` object causes it to run the REVERSE method. Most objects are capable of more than one action, and so have a number of available methods.

The classes REXX provides include their own predefined methods. The Message class, for example, has the COMPLETED, INIT, NOTIFY, RESULT, SEND, and START methods. When you create your own classes, you can write new methods for them in REXX code. Much of the object programming in REXX is writing the code for the methods you create.

Polymorphism

REXX lets you send the same message to objects that are different:

```
'!iH'~reverse   /* Reverses the characters "!iH" to form "Hi!"  */
pen~reverse     /* Reverses the direction of a plotter pen      */
ball~reverse    /* Reverses the direction of a moving ball      */
```

As long as each object has its own REVERSE method, REVERSE runs even if the programming implementation is different for each object. This ability to hide different functions behind a common interface is called *polymorphism*. As a result of information hiding, each object in the previous example knows only its own version of REVERSE. And even though the objects are different, each reverses itself as dictated by its own code.

Although the `!iH` object's REVERSE code is different from the plotter pen's, the method name can be the same because REXX keeps track of the methods each object owns. The ability to reuse the same method name so that one message can initiate more than one function is another feature of polymorphism. You do not need to have several message names like REVERSE_STRING, REVERSE_PEN, REVERSE_BALL. This keeps method-naming schemes simple and makes complex programs easy to follow and modify.

The ability to hide the various implementations of a method while leaving the interface the same illustrates polymorphism at its lowest level. On a higher level, polymorphism permits extensive code reuse.

Classes and Instances

In REXX, objects are organized into *classes*. Classes are like templates; they define the methods and variables that a group of similar objects have in common and store them in one place.

If you write a program to manipulate some screen icons, for example, you might create an Icon class. In that Icon class you can include all the icon objects with similar actions and characteristics:

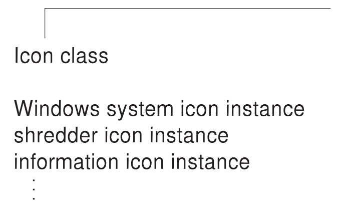


Figure 5. A Simple Class

All the icon objects might use common methods like DRAW or ERASE. They might contain common variables like position, color, or size. What makes each icon object different from one another is the data assigned to its variables. For the Windows system icon, it might be position='20,20', while for the shredder it is '20,30' and for information it is '20,40':

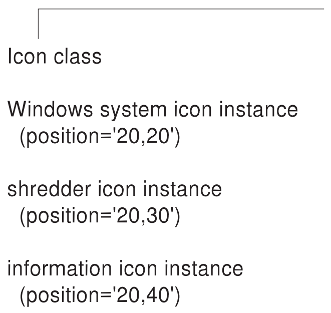

Icon class
Windows system icon instance (position='20,20')
shredder icon instance (position='20,30')
information icon instance (position='20,40')

Figure 6. Icon Class

Objects that belong to a class are called *instances* of that class. As instances of the Icon class, the Windows system icon, shredder icon, and information icon *acquire* the methods and variables of that class. Instances behave as if they each had their own methods and variables of the same name. All instances, however, have their own unique properties—the *data* associated with the variables. Everything else can be stored at the class level.

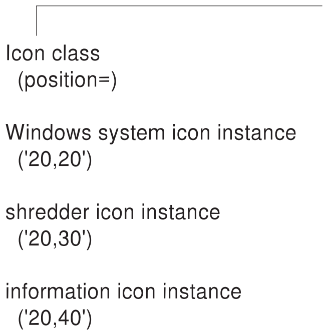


Figure 7. Instances of the Icon Class

If you must update or change a particular method, you only have to change it at one place, at the class level. This single update is then acquired by every new instance that uses the method.

A class that can create instances of an object is called an *object class*. The Icon class is an object class you can use to create other objects with similar properties, such as an application icon or a drives icon.

An object class is like a factory for producing instances of the objects.

Data Abstraction

The ability to create new, high-level data types and organize them into a meaningful class structure is called *data abstraction*. Data abstraction is at the core of object-oriented programming. Once you model objects with real-world properties from the basic data types, you can continue creating, assembling, and combining them into increasingly complex objects. Then you can use these objects as if they were part of the original programming language.

Subclasses, Superclasses, and Inheritance

When you write your first object-oriented program, you do not have to begin your real-world modeling from scratch. REXX provides predefined classes and methods. From there you can create additional classes and methods of your own, according to your needs.

REXX classes are hierarchical. Any *subclass* (a class below another class in the hierarchy) *inherits* the methods and variables of one or more *superclasses* (classes above a class in the hierarchy):

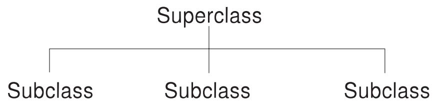


Figure 8. Superclass and Subclasses

You can add a class to an existing superclass. For example, you might add the Icon class to the Screen-Object superclass:

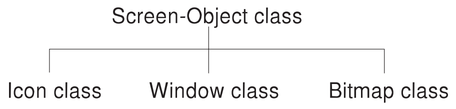


Figure 9. The Screen-Object Superclass

In this way, the subclass inherits additional methods from the superclass. A class can have more than one superclass, for example, subclass Bitmap might have the superclasses Screen-Object and Art-Object. Acquiring methods and variables from more than one superclass is known as *multiple inheritance*:

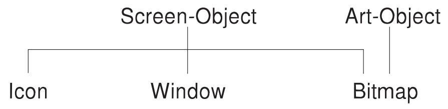


Figure 10. Multiple Inheritance

Structure and General Syntax

On Windows, REXX programs are not required to start with a standard comment. However, for portability reasons, start each REXX program with a standard comment that begins in the first column of the first line. For more information on comments, refer to “Comments” on page 10.

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 12)
- Zero or more blanks (again ignored)
- A semicolon (;) delimiter that the line end, certain keywords, or the colon (:) implies.

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 several blanks (except within literal strings) are converted to single blanks. Blanks adjacent to operator characters and special characters 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 depend on the character set used.

A code page specifies the encodings for each character in a set. 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 mark (!).

Comments

A comment is a sequence of characters delimited by specific characters. It is ignored by the program but acts as a separator. For example, a token containing one comment is treated as two tokens.

The interpreter recognizes the following types of comments:

- A line comment, where the comment is limited to one line
- The standard REXX comment, where the comment can cover several lines

A *line comment* is started by two subsequent minus signs (––) and ends at the end of a line. Example:

```
'Fred'
"Don't Panic!"
'You shouldn''t'        -- Same as "You shouldn't"
''
```

In this example, the language processor processes the statements from `'Fred'` to `'You shouldn''t'`, ignores the words following the line comment, and continues to process the statement `''`.

A *standard comment* is a sequence of characters (on one or more lines) delimited by /* and */. Within these delimiters any characters are allowed. Standard comments can contain other standard comments, as long as each begins and ends with the necessary delimiters. They are called *nested comments*. Standard comments can be anywhere and of any length.

```
/* 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 standard comment. It would not process the rest of the program because it would be looking for a matching standard 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)
```

Both types of comments can be mixed and nested. However, when you nest the two types, the type of comment that comes first takes precedence over the one nested. Here is an example:

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

In this example, the language processor ignores everything after `'You shouldn''t'` up to the end of the last line. In this case, the standard comment has precedence over the line comment.

When nesting the two comment types, make sure that the start delimiter of the standard comment /* is not in the line commented out with the line comment signs.

Example:

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

This example produces an error because the language processor ignores the start delimiter of the standard comment, which is commented out using the line comment.

Tokens

A *token* is the unit of low-level syntax from which clauses are built. Programs written in REXX are composed of tokens. Tokens can be of any length, up to an implementation-restricted maximum. They are separated by blanks or comments, or by the nature of the tokens themselves. The classes of tokens are:

- Literal strings
- Hexadecimal strings
- Binary strings
- Symbols
- Numbers
- Operator characters
- Special characters

Literal Strings

A literal string is a sequence including *any* characters except line feed (X'10') and delimited by a single quotation mark (') or a double quotation mark ("). You use two consecutive double quotation marks ("") to represent one double quotation mark (") within a string delimited by double quotation marks. Similarly, you use two consecutive single quotation marks ('') to represent one single quotation mark (') within a string delimited by single quotation marks. A literal string is a constant and its contents are never modified when it is processed. Literal strings must be complete on a single line. This means that unmatched quotation marks can be detected on the line where they occur.

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         */
```

Implementation maximum: A literal string can contain up to 250 characters. The length of the evaluated result of an expression, however, is limited only by the available virtual storage of your computer, with an additional limit of 512MB maximum per process.

Note that a string immediately followed by a right bracket is considered to be the name of a function. If immediately followed 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.

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 beginning of the string to make an even number of hexadecimal digits. The groups of digits are optionally separated by one or more blanks, and the whole sequence is delimited by single or double quotation marks and immediately followed by the symbol `X` or `x`. Neither `x` nor `X` can be part of a longer symbol. The blanks, which can only be byte boundaries (and not at the beginning or end of the string), are to improve 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, '41'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. It is an escape mechanism that lets a user describe a character in terms of its encoding (and, therefore, is machine-dependent). In ASCII, '20'X is the encoding for a blank. In every case, a string of the form '.....'x is an alternative to a straightforward string. In ASCII '41'x and 'A' are identical, as are '20'x and a blank, and must be treated identically.

Implementation maximum: The packed length of a hexadecimal string (the string with blanks removed) can be up to 250 bytes.

Binary Strings

A binary string is a literal string, expressed using a binary representation of its encoding. It is any sequence of zero or more binary digits (0 or 1) in groups of 8 (bytes) or 4 (nibbles). The first group can have less than four digits; in this case, up to three 0 digits are assumed to the left of the first digit, making a total of four digits. The groups of digits are optionally separated by one or more blanks, and the whole sequence is delimited by matching single or double quotation marks and immediately followed by the symbol b or B. Neither b nor B can be part of a longer symbol. The blanks, which can only be byte or nibble boundaries (and not at the beginning or end of the string), are to improve 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 8, leading zeros are added on the left to make a multiple of 8 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 binary-literal string can be up to 250 bytes.

Symbols

Symbols are groups of characters, selected from the:
- English alphabetic characters (A–Z and a–z). Note that some code pages do not include lowercase English characters a–z.
- Numeric characters (0–9)
- Characters . ! ? and underscore (_). Note that the encoding of the exclamation mark depends on the code page used.

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

These are valid symbols:

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

If a symbol does not begin with a digit or a period, you can use it as a variable and can assign it a value. If you have not assigned a value to it, 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 directly be assigned a value. (See "Environment Symbols" on page 35.)

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

These are valid numbers in exponential notation:

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

Numbers

Numbers 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 can occur to a precision specified by the NUMERIC DIGITS instruction (the default is nine digits). See "Chapter 11. Numbers and Arithmetic" on page 421 for a full definition of numbers.

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

These are valid numbers:

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

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 can 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 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 nine).

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 19). (The `||` can also be used as the concatenation symbol.) 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 (and some special characters—see the next section) might not be available in all character sets. In this case, appropriate translations can be used. In particular, the vertical bar (|) 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.

Note: The REXX interpreter uses ASCII character 124 in the concatenation operator and as the logical OR operator. Depending on the code page or keyboard for your particular country, ASCII 124 can be shown as a solid vertical bar (|) or a split vertical bar (¦). The character on the screen might not match the character engraved on the key. If you receive error 13, `Invalid character in program`, on an instruction including a vertical bar character, make sure this character is ASCII 124.

The REXX interpreter uses ASCII character 170 for the logical NOT operator. Depending on your country, the ¬ might not appear on your keyboard. If the character is not available, you can use the backslash (\) in place of ¬.

Special Characters

The following characters, together with the operator characters, 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 or bracket is deleted only if it is also adjacent to another special character (unless the character is a parenthesis or bracket 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 removes the blanks in `(A) + (Z)` because this is equivalent to `(A)+(Z)`.

Example
The following example shows how a clause is composed of tokens:

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

This example is composed of six tokens—a literal string (`'REPEAT'`), a blank operator, a symbol (`A`, which can have an assigned 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 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 on 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 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.
```

Terms, 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. All expressions evaluate to objects.

Everything in REXX is an object. REXX provides some objects, which are described in later sections. You can also define and create objects that are useful in particular applications—for example, a menu object for user interaction. See “Modeling Objects” on page 3 for more information.

Terms and Expressions

Terms are literal strings, symbols, message terms, function calls, or subexpressions interspersed with zero or more operators that denote operations to be carried out on terms.

Literal strings, which are delimited by quotation marks, are constants.

Symbols (no quotation marks) are translated to uppercase. A symbol that does not begin with a digit or a period can be the name of a variable; in this case the value of that variable is used. A symbol that begins with a period can identify an object that the current environment provides; in this case, that object is used. Otherwise a symbol is treated as a constant string. A symbol can also be *compound*.

Message terms are described in “Message Terms” on page 25.

Function calls (see "Chapter 8. Functions" on page 297), which are of the following form:

```
                       ┌─,──────────────┐
                       ▼                │
►►──symbolorstring(─────┬──────────────┬─┴──)─────────────────────────────►◄
                        └─expression───┘
```

The *symbolorstring* is a symbol or literal string.

An *expression* consists of one or more terms. A *subexpression* is a term in an expression surrounded with a left and a right parenthesis.

Evaluation of an expression is left to right, modified by parentheses and operator precedence in the usual algebraic manner (see "Parentheses and Operator Precedence" on page 23). Expressions are wholly evaluated, unless an error occurs during evaluation.

As each term is used in an expression, it is evaluated as appropriate. The result is an object. Consequently, the result of evaluating any expression is itself an object (such as a character string).

Operators

An *operator* is a representation of an operation, such as an addition, to be carried out on one or two terms. Each operator, except for the prefix operators, acts on two terms, which can be symbols, strings, function calls, message terms, 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, if they occur in expressions but are not adjacent to another operator, also act as an operator. The language processor functionally translates operators into message terms. For dyadic operators, which operate on two terms, the language processor sends the operator as a message to the term on the left, passing the term on the right as an argument. For example, the sequence

```
say 1+2
```

is functionally equivalent to:

```
say 1~'+'(2)
```

The blank concatenation operator sends the message " " (a single blank), and the abuttal concatenation operator sends the "" message (a null string). When the ¬ character is used in an operator, it is changed to a \. That is, the operators ¬= and \= both send the message \= to the target object.

For an operator that works on a single term (for example, the prefix − and prefix + operators), REXX sends a message to the operand, with no arguments. This means `-z` has the same effect as `z~'-'`.

See “Operator Methods” on page 179 for operator methods of the Object class and “Arithmetic Methods” on page 208 for operator methods of the String class.

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 only separated 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`.

The two adjoining strings, one hexadecimal and one literal, `'4a 4b'x'LMN'` evaluate to `JKLMN`.

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 "Numbers" on page 15) using the following 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 can 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 11. Numbers and Arithmetic" on page 421 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 the other 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 other comparison operators, if *both* terms involved are numeric, a numeric comparison (see "Numeric Comparisons" on page 426) is effected. Otherwise, both terms are treated as character strings, leading and trailing blanks are ignored, and the shorter string is padded with blanks on the right.

Character comparison and strict comparison operations are both case-sensitive, and the exact collating order might depend on the character set used for the implementation. In an ASCII environment, such as Windows, the ASCII character value of digits is lower than that of the alphabetic characters, and that of lowercase alphabetic characters is higher than that of uppercase alphabetic characters.

The comparison operators and operations are:

=	True if the terms are equal (numerically or when padded)
\=, ¬=	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 has the value false if it is `0`, and true if it is `1`. A logical operator can take at least two values and return `0` or `1` as appropriate:

&	AND — returns 1 if both terms are true.

|	Inclusive OR — returns 1 if either term or both terms are true.
&&	Exclusive OR — returns 1 if either term, but not both terms, 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 the arguments on messages (see “Message Terms” on page 25) and 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 precedence over `operator1`, the subexpression (`term2 operator2 term3`) is evaluated first.

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

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

+ - ¬ \\	(prefix operators)
**	(power)
* / % //	(multiply and divide)
+ -	(add and subtract)
(blank) || (abuttal)	(concatenation with or without blank)
= > <	(comparison operators)
== >> <<	
\\= ¬=	
>< <>	

\> ¬>

\< ¬<

\== ¬==

\>> ¬>>

\<< ¬<<

>= >>=

<= <<=

&	(and)
| &&	(or, exclusive or)

Examples:

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

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

Note: 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 from left to right.

For example:

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

Message Terms

You can include *messages* to objects in an expression wherever a term, such as a literal string, is valid. A message can be sent to an object to perform an action, obtain a result, or both.

A *message term* can have one of the following forms:

```
>>──receiver──┬─ ~ ─┬──messagename──┬─────────┬──┬──────────────────────────┬──────><
              └─ ~~ ┘               └─:symbol─┘  └─(──┬──────────────────┬──)─┘
                                                      │ ┌─,──────────┐   │
                                                      │ ▼            │   │
                                                      └───expression─┴───┘
```

```
>>──receiver[──┬──────────────────┬──]──────────────────────────────────────────><
               │ ┌─,──────────┐   │
               │ ▼            │   │
               └───expression─┴───┘
```

The *receiver* is a term (see "Terms and Expressions" on page 18 for a definition of term). It receives the message. The ~ or ~~ indicates sending a message. The *messagename* is a literal string or a symbol that is taken as a constant. The *expression*s (separated by commas) between the parentheses or brackets are the *arguments* for the message. The *receiver* and the argument *expression*s can themselves include message terms. If the message has no arguments, you can omit the parentheses.

The left parenthesis, if present, must immediately follow a token (*messagename* or *symbol*) with **no** blank in between them. Otherwise, only the first part of the construct is recognized as a message term. (A blank operator would be assumed at that point.) Only a comment (which has no effect) can appear between a token and the left parenthesis.

You can use any number of *expression*s, separated by commas. The *expression*s are evaluated from left to right and form the argument during the execution of the routine. Any ARG, PARSE ARG, or USE ARG instruction or ARG built-in function in the called routine accesses these objects while the called routine is running. You can omit *expression*s, if appropriate, by including extra commas.

The *receiver* is evaluated, followed by one or more *expression* arguments. The message name (in uppercase) and the resulting argument objects are then sent to the receiver object. The receiver object selects a method to be run based on the message name (see Table 1 on page 108), and runs the selected method with the specified argument objects. The receiver eventually returns, allowing processing to continue.

If the message term uses ~, the receiver must return a result object. This object is included in the original expression as if the entire message term had been replaced by the name of a variable whose value is the returned object.

For example, the message POS is valid for strings, and you could code:

```
c='escape'
a="Position of 'e' is:" c~pos('e',3)
/* would set A to "Position of 'e' is: 6" */
```

If the message term uses ~~, the receiver needs not return a result object. Any result object is discarded, and the receiver object is included in the original expression in place of the message term.

For example, the messages INHERIT and SUBCLASS are valid for classes (see “The Class Class” on page 161) and, assuming the existence of the Persistent class, you could code:

```
account = .object~subclass('Account')~~inherit(.persistent)
/* would set ACCOUNT to the object returned by SUBCLASS, */
/* after sending that object the message INHERIT */
```

If the message term uses brackets, the message [] is sent to the receiver object. (The *expression*s within the brackets are available to the receiver object as arguments.) The effect is the same as for the corresponding ~ form of the message term. Thus, `a[b]` is the same as `a~'[]'(b)`.

For example, the message [] is valid for arrays (see “The Array Class” on page 118) and you could code:

```
a = .array~of(10,20)
say "Second item is" a[2]   /* Same as: a~at(2)             */
/* or a~'[]'(2)                  */
/* Produces: "Second item is 20" */
```

A message can have a variable number of arguments. You need to specify only those required. For example, `'ESCAPE'~POS('E')` returns 1.

A colon (:) and symbol can follow the message name. In this case, the symbol must be the name of a variable (usually the special variable SUPER—see page 481) or an environment symbol (see “Environment Symbols” on page 35). The resulting value changes the usual method selection. For more information, see “Changing the Search Order for Methods” on page 102.

Message Sequences

The ~ and ~~ forms of message terms differ only in their treatment of the result object. Using ~ returns the result of the method. Using ~~ returns the object that received the message. Here is an example:

```
/* Two ways to use the INSERT method to add items to a list */
/* Using only ~ */
team = .list~of('Bob','Mary')
team~insert('Jane')
team~insert('Joe')
team~insert('Steve')
say 'First on the team is:' team~firstitem     /* Bob   */
say 'Last on the team is:' team~lastitem       /* Steve */
/* Do the same thing using ~~ */
team=.list~of('Bob','Mary')
/* Because ~~ returns the receiver of the message       */
/* each INSERT message following returns the list       */
/* object (after inserting the argument value).         */
team~~insert('Jane')~~insert('Joe')~~insert('Steve')
say 'First on the team is:' team~firstitem     /* Bob   */
say 'Last on the team is:' team~lastitem       /* Steve */
```

Thus, you would use ~ when you want the returned result to incorporate the methods included in each stage of the message.

Clauses and Instructions

Clauses can be subdivided into the following types:

- Null clauses
- Directives
- Labels
- Instructions
- Assignments
- Message instructions
- Keyword instructions
- Commands

Null Clauses

A clause consisting only of blanks, comments, or both is a *null clause*. It is completely ignored.

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 the C language). The NOP instruction is provided for this purpose.

Directives

A clause that begins with two colons is a *directive*. Directives are nonexecutable code and can start in any column. They divide a program into separate executable units (methods and routines) and supply information about the program or its executable units. Directives perform various functions, such as associating methods with a particular class (::CLASS directive) or defining a method (::METHOD directive). See "Chapter 3. Directives" on page 87 for more information about directives.

Labels

A clause that consists of a single symbol or string followed by a colon is a *label*. The colon in this context implies a semicolon (clause separator), so no semicolon is required.

The label's name is taken from the string or symbol part of the label. If the label uses a symbol for the name, the label's name is in uppercase. If a label uses a string, the name can contain mixed-case characters.

Labels identify the targets of CALL instructions, SIGNAL instructions, and internal function calls. Label searches for CALL, SIGNAL, and internal function calls are case-sensitive. Label-search targets specified as symbols cannot match labels with lowercase characters. Literal-string or computed-label searches can locate labels with lowercase characters. More than one label can precede an instruction. Labels are treated as null clauses and can be traced selectively to aid debugging.

Labels can be any number of successive clauses. Several labels can precede other clauses. Duplicate labels are permitted, but control is only passed to the first of any duplicates in a program. The duplicate labels occurring later can be traced but cannot be used as a target of a CALL, SIGNAL, or function invocation.

Instructions

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

Assignments

A single clause of the form *symbol=expression* is an instruction known as an *assignment*. An assignment gives a (new) value to a variable. See "Assignments and Symbols" on page 29.

Message Instructions

A *message instruction* is a single clause in the form of a message term (see "Message Terms" on page 25) or in the form *messageterm=expression*. A message is sent to an object, which responds by performing some action. See "Message Instructions" on page 36.

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, for example, the external interfaces and the flow of control. 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 a particular instruction, for example, the symbols TO and WHILE in the DO instruction.

Commands

A *command* is a clause consisting of an expression only. The expression is evaluated and the result is passed as a command string to an 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 object. Note that an object can be composed of other objects, such as an array or directory object.

You can assign a new value to a variable with the ARG, PARSE, PULL, or USE 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 in 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.

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.

You can use a symbol in an expression even if you have not assigned a value to it, because a symbol has a defined value at all times. A variable to which 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 33), 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, a symbol belongs to one of the following groups: constant symbols, simple symbols, compound symbols, environment 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 although the collection classes might be preferable in many cases. See "Chapter 5. The Collection Classes" on page 117.

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
```

Symbols where the first character is a period and the second character is alphabetic are environment symbols.

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
```

Stems

A *stem* is a symbol that contains a period as the last character. It cannot start with a digit or a period.

These are stems:

```
FRED.
A.
```

By default, the value of a stem is a Stem object. (See “The Stem Class” on page 186.) The stem variable's Stem object is automatically created the first time you use the stem variable or a compound variable (see “Compound Symbols” on page 33) containing the stem variable name. The Stem object's assigned name is the name of the stem variable (with the characters translated to uppercase). If the stem variable has been assigned a value, or the Stem object has been given a default value, a reference to the stem variable returns the assigned default value.

Further, when a stem is the target of an assignment, a new Stem object is created and assigned to the stem variable. The new value assigned to the stem variable is given to the new Stem object as a default value. Following the assignment, a reference to any compound symbol with that stem variable returns the new value until another value is assigned to the stem, the Stem object, or the individual compound variable.

Example:

```
hole.  = "empty"
hole.19 = "full"
say  hole.1  hole.mouse  hole.19
/* says "empty empty full" */
```

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

If the object assigned to a stem variable is already a Stem object, then a new Stem object is not created. The assignment updates the stem variable to refer to the existing Stem object.

Example:

```
hole.   = "empty"
hole.19 = "full"
say  hole.1  hole.mouse  hole.19
/* Says "empty empty full" */

hole2. = hole.       /* copies reference to hole. stem to hole2. */

say  hole2.1  hole2.mouse  hole2.19

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

You can pass stem collections as function, subroutine, or method arguments.

Example:

```
/* CALL RANDOMIZE count, stem. calls routine */
Randomize: Use Arg count, stem.
do i = 1 to count
  stem.i = random(1,100)
end
return
```

Note: USE ARG must be used to access the stem variable as a collection. PARSE and PARSE ARG force the stem to be a string value.

Stems can also be returned as function, subroutine, or method results.

Example:

```
/* RANDOMIZE(count) calls routine */
Randomize: Use Arg count
do i = 1 to count
  stem.i = random(1,100)
end
return stem.
```

Note: The value that has been assigned to the whole collection of variables can always be obtained by using the stem. However, this is not the same as using a compound variable whose derived name is the null string.

Example:

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

You can use the DROP, EXPOSE, and PROCEDURE instructions to manipulate collections of variables, referred to by their stems. `DROP FRED.` assigns a new Stem object to the specified stem. (See "DROP" on page 50.) `EXPOSE FRED.` and `PROCEDURE EXPOSE FRED.` expose all possible variables with that stem (see "EXPOSE" on page 52 and "PROCEDURE" on page 66).

The DO instruction can also iterate over all of the values assigned to a stem variable. See "DO" on page 49 for more details.

Notes:

1. When the ARG, PARSE, PULL, or USE instruction, the VALUE built-in function, or the variable pool interface changes a variable, the effect is identical with an assignment. Wherever a value can be assigned, using a stem sets an entire collection of variables.
2. 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 that

the clause is processed as a command, such as by putting a null string before the first name, or by enclosing the first part of the expression in parentheses.

If you unintentionally use a REXX keyword as the variable name in an assignment, this should not cause confusion. For example, the following clause is an assignment, not an ADDRESS instruction:

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

3. You can use the VAR function (see "VAR" on page 353) to test whether a symbol has been assigned a value. In addition, you can set SIGNAL ON NOVALUE to trap the use of any uninitialized variables (except when they are tails in compound variables—see page 431–or stems).

Compound Symbols

A *compound symbol* contains at least one period and two other characters. It cannot start with a digit or a period, and if there is only one period it cannot be the last character.

The name begins with a stem (that part of the symbol up to and including the first period) and is followed by a tail, which are parts of the name (delimited by periods) that are constant symbols, simple symbols, or null. Note that 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 valid.

These are compound symbols:

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

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

The substitution in the symbol 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 once.

More formally, the derived name of a compound variable that is referenced by the symbol

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

is given by

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

where `d0` is the name of the Stem object associated with the stem variable `s0` and `v1` to `vn` are the values of the constant or simple symbols `s1` through `sn`. Any of the symbols `s1` to `sn` can be null. The values `v1` to `vn` can also be null and can contain *any* characters. Lowercase characters are not translated to uppercase, blanks are not removed, and periods have no special significance. There is no limit on the length of the evaluated name.

Some examples of simple and compound symbols follow in the form of a small extract from a REXX program:

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

You can use compound symbols to set up arrays and lists of variables in which the subscript is not necessarily numeric, thus offering a 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, producing a form of associative memory (content-addressable).

Evaluated Compound Variables

The value of a stem variable is always a Stem object (see "The Stem Class" on page 186 for details). A Stem object is a type of collection that supports the [] and []= methods used by other collection classes. The [] provides an alternate means of accessing compound variables that also allows embedded subexpressions.

Examples:

```
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.[z+1]='Rick'      /*   'Rick'   to A.5      */
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.[z+1] a.[c]  c.[a]  a.[fred] y.[a,z]
/* displays the string:                     */
/*    "3 4 Fred A.3 Fred Rick Bill C.3 5 Annie"  */
```

Environment Symbols

An environment symbol starts with a period and has at least one other character. This character must not be a digit. By default the value of an environment symbol is the string consisting of the characters of the symbol (translated to uppercase). If the symbol identifies an object in the current environment, its value is that object.

These are environment symbols:

```
.method    /* Same as .METHOD */

.true
```

When you use an environment symbol, the language processor performs a series of searches to see if the environment symbol has an assigned value. The search locations and their ordering are:

1. The directory of classes declared on ::CLASS directives (see "::CLASS" on page 87) within the current program file.
2. The directory of PUBLIC classes declared on ::CLASS directives of other files included with a ::REQUIRES directive.
3. The local environment directory. The local environment includes process-specific objects such as the .INPUT and .OUTPUT objects. You can directly access the local environment directory by using the .LOCAL environment symbol. (See "The Local Environment Object (.LOCAL)" on page 294 .)
4. The global environment directory. The global environment includes all permanent REXX objects such as the REXX supplied classes (.ARRAY and so on) and constants such as .TRUE and .FALSE. You can directly access the global environment by using the .ENVIRONMENT environment symbol (see "The Environment Object" on page 293) or the VALUE built-in function (see "VALUE" on page 350) with a null string for the *selector* argument.
5. REXX defined symbols. Other simple environment symbols are reserved for use by REXX built-in objects. The currently defined built-in objects are .RS and .METHODS.

If an entry is not found for an environment symbol, then the default character string value is used.

Note: You can place entries in both the .LOCAL and the .ENVIRONMENT directories for programs to use. To avoid conflicts with future REXX defined entries, it is recommended that the entries that you place in either directory include at least one period in the entry name.

Example:

```
/* establish settings directory */
.local~setentry('MyProgram.settings', .directory~new)
```

Message Instructions

You can send a message to an object to perform an action, obtain a result, or both. You use a message instruction if the main purpose of the message is to perform an action. You use a message term (see “Message Terms” on page 25) if the main purpose of the message is to obtain a result.

A *message instruction* is a clause of the form:

```
>>--messageterm--+-------------+--;---------------------------------><
                 +-=expression-+
```

If there is only a *messageterm*, the message is sent in exactly the same way as for a message term (see “Message Terms” on page 25). If the message yields a result object, it is assigned to the sender’s special variable RESULT. If you use the ~~ form of message term, the receiver object is used as the result. If there is no result object, the variable RESULT is dropped (becomes uninitialized).

Example:

```
mytable~add('John',123)
```

This sends the message ADD to the object MYTABLE. The ADD method need not return a result. If ADD returns a result, the result is assigned to the variable RESULT.

The equal sign (=) sets a value. If `=expression` follows the message term, a message is sent to the receiver object with an = concatenated to the end of the message name. The result of evaluating the expression is passed as the first argument of the message.

Examples:

```
person~age = 39                    /* Same as person~'AGE='(39) */
table[i] = 5                       /* Same as table~'[]='(5,i)  */
```

The expressions are evaluated in the order in which the arguments are passed to the method. That is, the language processor evaluates the `=expression` first. Then it evaluates the argument expressions within any [] pairs from left to right.

Commands to External Environments

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

Environment

The base system for the language processor 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 environments selected depend on the caller. Normally the default environment is the used shell, mostly 'CMD' on Windows® systems. If called from an editor that accepts subcommands from the language processor, the default environment can be that editor.

A REXX program can issue commands—called *subcommands*—to other application programs. For example, a REXX program written for a text editor can inspect a file being edited, issue subcommands to make changes, test return codes to check that the subcommands have been processed as expected, and display messages to the user when appropriate.

An application that uses REXX as a macro language must register its environment with the REXX language processor. See the *Object REXX for Windows: Programming Guide* for a discussion of this mechanism.

Commands

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

```
expression;
```

The expression (which must not be an expression that forms a valid message instruction—see “Message Instructions” on page 36) is evaluated, resulting in a character string value (which can 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 must be enclosed in quotation marks.

The environment then processes the command and 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 processed. A return code usually indicates if a command was successful but can also represent other information. The language processor places this return code in the REXX special variable RC. See “Chapter 19. Special Variables” on page 481.

In addition to setting a return code, the underlying system can also indicate to the language processor if an error or failure occurred. An *error* is a condition raised by a command to which a program that uses that command can respond. 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 to which a program that uses that command cannot respond, 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 12. Conditions and Condition Traps” on page 429). They can 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 79.

The .RS environment symbol can also be used to detect command failures and errors. When the command environment indicates that a command failure has occurred, the REXX environment symbol .RS has the value `-1`. When a command error occurs, .RS has a value of `1`. If the command did not have a FAILURE or ERROR condition, .RS is `0`.

Here is an example of submitting a command. Where the default environment is Windows, the sequence:

```
fname = "CHESHIRE"
exten = "CAT"
"TYPE" fname"."exten
```

would result in passing the string `TYPE CHESHIRE.CAT` to the command processor. On Windows 95, this is COMMAND.COM. On Windows NT, this is CMD.EXE. The simpler expression:

```
"TYPE CHESHIRE.CAT"
```

has the same effect.

On return, the return code placed in RC will have the value 0 if the file CHESHIRE.CAT were typed, or a nonzero value if the file could not be found in the current directory.

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:

```
delete "*".lst          /* not "multiplied by" */
var.003 = anyvalue
type "var.003"          /* not a compound symbol */
```

```
w = any
dir"/w"                 /* not "divided by ANY" */
```

Enclosing an entire message instruction in parentheses causes the message result to be used as a command. Any clause that is a message instruction is not treated as a command. Thus, for example, the clause

```
myfile~linein
```

causes the returned line to be assigned to the variable RESULT, not to be used as a command to an external environment.

Using REXX on Windows

REXX programs can call other REXX programs as external functions or subroutines with the call instruction.

If a program is called with the call instruction, the program runs in the same process as the calling program. If you call another program by a REXX command, the program is executed in a new process and therefore does not share .environment, .local, or the Windows environment.

Examples:

```
call "other.REX"        /* runs in the same process */
"rexx other.REX"        /* runs in a new child process */
"start rexx other.REX"  /* runs in a new detached process */
```

When REXX programs call other REXX programs as commands, the return code of the command is the exit value of the called program provided that this value is a whole number in the range -32768 to 32767. Otherwise, the exit value is ignored and the called program is given a return code of 0.

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 the end of a line implies them.

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 equal (=) character (implying an assignment) or a colon (implying a label). The keywords ELSE, END, OTHERWISE, THEN, and WHEN are treated in the same way. 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).

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.

Blanks adjacent to keywords 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 improve readability:

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

ADDRESS

```
>>──ADDRESS──┬───────────────────────────────┬──;────────────────────────────><
             ├─environment──┬──────────────┬─┤
             │              └─expression───┘ │
             └─┬───────┬──expression1────────┘
               └─VALUE─┘
```

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. (See “Commands to External Environments” on page 37.)

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 ofan external procedure or process that can process commands. The *expression* is evaluated to produce a character string value, and this 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 its original state, thus temporarily changing the destination for a single command. The special variable RC and the environment symbol .RS are set and errors and failures in commands processed in this way are trapped or traced.

Example:

```
ADDRESS CMD "DIR C:\CONFIG.SYS"
```

If you specify only *environment*, a lasting change of destination occurs: all commands (see “Commands” on page 37) that follow are routed to the specified command environment, until the next ADDRESS instruction is processed. The previously selected environment is saved.

Examples:

Assume that the environment for a text editor is registered by the name `EDIT`:

```
address CMD
'DIR C:\AUTOEXEC.BAT'
if rc=0 then 'COPY C:\AUTOEXEC.BAT C:\*.TMP'
address EDIT
```

Subsequent commands are passed to the editor until the next ADDRESS instruction.

Similarly, you can use the VALUE form to make a lasting change to the environment. Here *expression1*, which can be a variable name, is evaluated, and the resulting character string value 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 change of the environment, and the current environment name is saved. After changing the environment, repeated execution of ADDRESS alone, therefore, switches the command destination between two environments. Using a null string for the environment name ("") is the same as using the default environment.

The two environment names are automatically saved across internal and external subroutine and function calls. See the CALL instruction ("CALL" on page 45) 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 305.) The *Object REXX for Windows: Programming Guide* describes the registration of alternative subcommand environments.

ARG

```
►►──ARG──┬───────────────┬──;──────────────────────────────────────►◄
         └─template_list─┘
```

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

```
►►──PARSE UPPER ARG──┬───────────────┬──;──────────────────────────►◄
                     └─template_list─┘
```

The *template_list* can be a single template orlist of templates separated by commas. Each template consists of one or more symbols separated by blanks, patterns, or both.

Unless a subroutine, internal function, or method is processed, the objects passed as parameters to the program are converted to string values and parsed into variables according to the rules described in "Chapter 10. Parsing" on page 403.

If a subroutine, internal function, or method is processed, the data used are the argument objects that the caller passes to the routine.

The language processor converts the objects to strings and translates the 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 objects (typically with different templates). The source objects do 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 object 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.

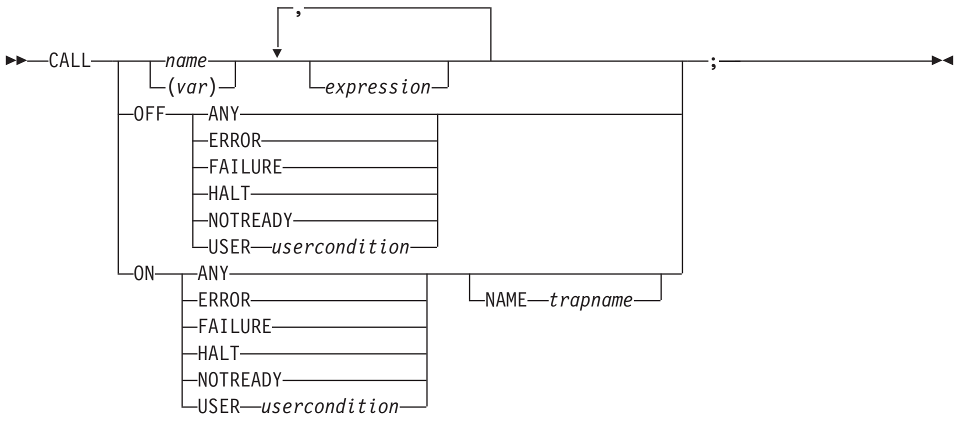
Example:

```
/* 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'                  */
```

Notes:

1. The ARG built-in function can also retrieve or check the arguments. See "ARG (Argument)" on page 305.
2. The USE ARG instruction (see "USE" on page 84) is an alternative way of retrieving arguments. USE ARG performs a direct, one-to-one assignment of argument objects to REXX variables. You should use this when your program needs a direct reference to the argument object, without string conversion or parsing. USE ARG also allows access to both string and non-string argument objects. ARG and PARSE ARG produce string values from the arguments, and the language processor then parses these.

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 12. Conditions and Condition Traps” on page 429.

To call a routine, specify *name*, which must be a literal string or symbol that is taken as a constant. The *usercondition* is a single symbol that is taken as a constant. The *trapname* is a symbol or string taken as a constant. 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 or subroutine that is defined as part of the REXX language.

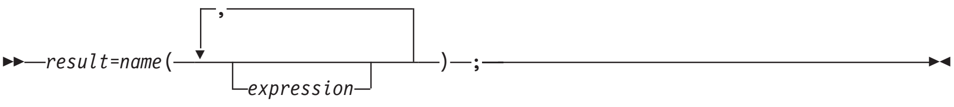
An external routine
A function or subroutine that is neither built-in nor in the same program as the CALL instruction or function call that calls it.

If *name* is a string in which case you specify it 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 external routines are in uppercase. Therefore, write the name in the literal string in uppercase characters.

File names can be in uppercase, lowercase, or mixed case. The search for files is case-insensitive to case. Therefore, when using CALL to run a REXX subroutine contained on a disk file (external routine), the case does not matter.

You can also specify (*var*), a single variable name enclosed in parentheses. The variable is evaluated before any of the argument expressions, and the value is the target of the CALL instruction. The language processor does not translate the variable value into uppercase, so the evaluated name must exactly match any label name. (See “Labels” on page 28 for a description of label names.)

The called routine can optionally return a result. In this case, the CALL instruction is functionally identical with the clause:



If the called routine does not return a result, you get an error if you call it as a function.

You can use any number of *expression*s, separated by commas. The expressions are evaluated from left to right and form the arguments during execution of the routine. Any ARG, PARSE ARG, or USE ARG instruction or ARG built-in function in the called routine accesses these objects while the called routine is running. You can omit expressions, if appropriate, by including extra commas.

The CALL then branches to the routine called *name*, using exactly the same mechanism as function calls. See “Chapter 8. Functions” on page 297. The search order 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. 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 an object that is the result of the routine. (See “ARG (Argument)” on page 305.)

Note: You can call any built-in function as a subroutine. Any result is stored in RESULT. Simply specify `CALL`, the function name (with *no parenthesis*) and any arguments:

```
call length "string"   /* Same as length("string") */
say result             /* Produces: 6              */
```

However, if you include a trailing comma, you must include the semicolon to prevent the interpretation of the last comma as a continuation character.

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. If the CALL instruction calls an external routine written in REXX as a subroutine, you can retrieve any argument strings with the ARG, PARSE ARG, or USE ARG instructions or the ARG built-in function.

For more information on the search order, see “Search Order” on page 299.

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, for example NUMERIC settings, 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 but not a built-in function or external routine, the line number of the CALL instruction is available in the variable SIGL (in the caller’s variable environment). This can be used as a debug aid because it is possible to find out how control reached a routine. Note that if the internal routine uses the PROCEDURE instruction, it needs to EXPOSE SIGL to get access to the line number of the CALL.

After the subroutine processed the RETURN instruction, 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.

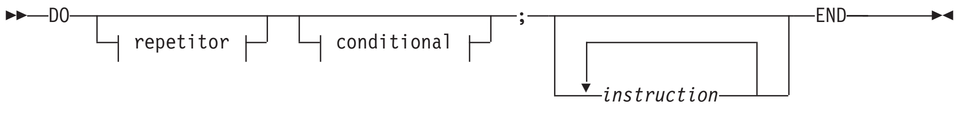
Example:

```
/* 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
```

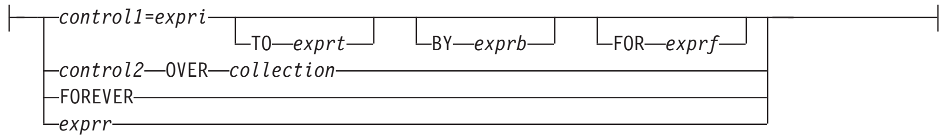
During internal subroutine (and function) execution, all important pieces of information are automatically saved and then restored upon return from the routine. These are:

- **The status of DO loops and other structures**: Executing a SIGNAL within a subroutine is safe because DO loops and other structures that were active when the subroutine was called are not ended. However, 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 without affecting the tracing of the caller. If you want 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) is saved across routines.
- **NUMERIC settings**: The DIGITS, FUZZ, and FORM of arithmetic operations (in "NUMERIC" on page 62) are saved and then restored on return. A subroutine can, therefore, set the precision, for example, that it needs to use without affecting the caller.
- **ADDRESS settings**: The current and previous destinations for commands (see "ADDRESS" on page 42) are saved and 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 313.
- **.RS value**: The value of the .RS environment symbol. (See ".RS" on page 455.)
- **Elapsed-time clocks**: A subroutine inherits the elapsed-time clock from its caller (see "TIME" on page 346), 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 then restored on return.

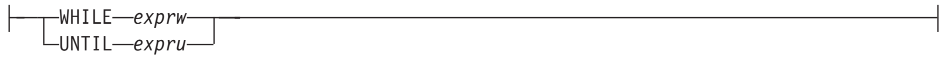
DO



repetitor:



conditional:



DO groups instructions and optionally processes them repetitively. During repetitive execution, a control variable (*control1* or *control2*) can be stepped through some range of values.

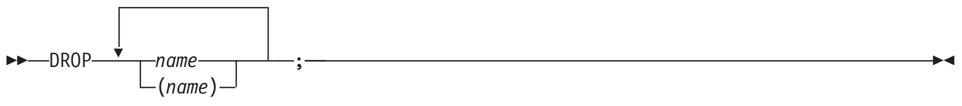
Notes:

1. 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.
2. The *exprw* or *expru* options, if present, can be any expression that evaluates to `1` or `0`.
3. The TO, BY, and FOR phrases can be in any order, if used, and are evaluated in the order in which they are written.
4. The *instruction* can be any instruction, including assignments, commands, message instructions, and keyword instructions (including any of the more complex constructs such as IF, SELECT, and the DO instruction itself).
5. 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 is not followed by an equal sign.
6. The *exprb* option defaults to `1`, if relevant.

7. The *collection* can be any expression that evaluates to an object that supports a MAKEARRAY method.

For more information, refer to “Appendix A. Using the DO Keyword” on page 515.

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 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 character strings separated by blanks, except that no parentheses are allowed. The list needs not contain any names—that is, it can be empty.

Variables are dropped from left to right. It is not an error to specify a name more than once or to drop a variable that is not known. If an exposed variable is named (see “EXPOSE” on page 52 and “PROCEDURE” on page 66), then the original variable 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) assigns the stem variable to a new, empty stem object.

Example:

```
Drop  z.
/* Assigns stem variable z. to a new empty stem object */
```

EXIT

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

EXIT leaves a program unconditionally. Optionally, EXIT returns a result object to the caller. The program is stopped immediately, even if an internal routine is being run. If no internal routine is active, RETURN (see “RETURN” on page 74) and EXIT are identical in their effect on the program running.

If you specify *expression*, it is evaluated and the object 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' */
```

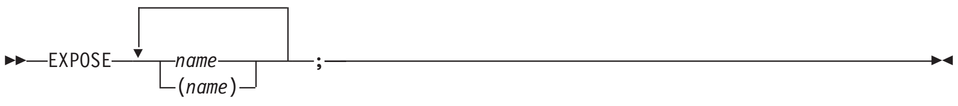
If you do not specify *expression*, no data is passed back to the caller. If the program was called as an external function, this is detected as an error—either immediately (if RETURN was used), or on return to the caller (if EXIT was used).

You can also use EXIT within a method. The method is stopped immediately, and the result object, if specified, is returned to the sender. If the method has previously issued a REPLY instruction (see “REPLY” on page 73), the EXIT instruction must *not* include a result expression.

Notes:

1. 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. The returned string must be a whole number whose value fits in a 16-bit signed integer (within the range -(2**15) to (2**15-1). If the conversion fails, no error is raised, and a return code of 0 is returned.
2. If you do not specify EXIT, EXIT is implied but no result string is returned.

EXPOSE



EXPOSE causes the object variables identified in *name* to be exposed to a method. References to exposed variables, including assigning and dropping, access variables in the current object's variable pool. (An object variable pool is a collection of variables that is associated with an object rather than with any individual method.)Therefore, the values of existing variables are accessible, and any changes are persistent even after RETURN or EXIT from the method.

Any changes a method makes to an object variable pool are immediately visible to any other methods that share the same object variable pool. All other variables that a method uses are local to the method and are dropped on RETURN or EXIT. If an EXPOSE instruction is included, it must be the first instruction of the method.

If parentheses enclose a single *name*, then, after the variable *name* is exposed, the character string value of *name* is immediately used as a subsidiary list of variables. Blanks are not necessary 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 once, or to specify a name that has not been used as a variable.

Example:

```
/* Example of exposing object variables */
myobj = .myclass~new
myobj~c
myobj~d         /* Would display "Z is: 120"               */

::class myclass /* The ::CLASS directive                   */
                /* (see "::CLASS" on page 87)                            */
::method c      /* The ::METHOD directive                  */
                /* (see "::METHOD" on page 89)                            */
expose z
z = 100         /* Would assign 100 to the object variable z */
return

::method d
```

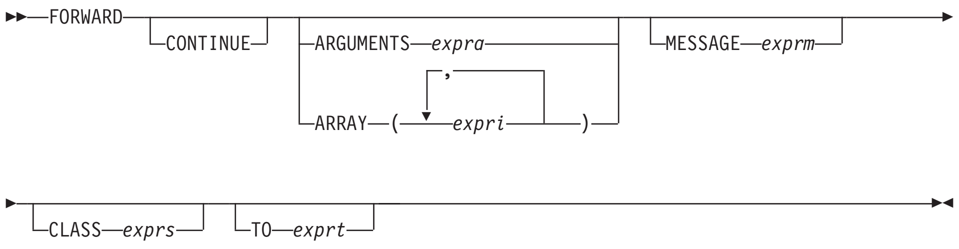
```
expose z
z=z+20          /* Would add 20 to the same object variable z */
say 'Z is:' z
return
```

You can expose an entire collection of compound variables (see "Compound Symbols" on page 33) by specifying their stem in the variable list or a subsidiary list. The variables are exposed for all operations.

Example:

```
expose j k  c. d.
/*  This exposes "J", "K", and all variables whose           */
/*  name starts with "C." or "D."                            */
c.1='7.'        /*  This sets "C.1" in the object            */
                /*  variable pool, even if it did not        */
                /*  previously exist.                        */
```

FORWARD



Note: You can specify the options in any order.

FORWARD forwards the message that caused the currently active method to begin running. The FORWARD instruction can change parts of the forwarded message, such as the target object, the message name, the arguments, and the superclass override.

If you specify the TO option, the language processor evaluates *exprt* to produce a new target object for the forwarded message. The *exprt* is a literal string, constant symbol, or expression enclosed in parentheses. If you do not specify the TO option, the initial value of the REXX special variable SELF is used.

If you specify the ARGUMENTS option, the language processor evaluates *expra* to produce an array object that supplies the set of arguments for the forwarded message. The *expra* can be a literal string, constant symbol, or expression enclosed in parentheses. The ARGUMENTS value must evaluate to a REXX array object.

If you specify the ARRAY option, each *expri* is an expression (use commas to separate the expressions). The language processor evaluates the expression list to produce a set of arguments for the forwarded message. It is an error to use both the ARRAY and the ARGUMENTS options on the same FORWARD instruction.

If you specify neither ARGUMENTS nor ARRAY, the language processor does not change the arguments used to call the method.

If you specify the MESSAGE option, the *exprm* is a literal string, a constant symbol, or an expression enclosed in parentheses. If you specify an expression enclosed in parentheses, the language processor evaluates the expression to obtain its value. The uppercase character string value of the MESSAGE option is the name of the message that the FORWARD instruction issues.

If you do not specify MESSAGE, FORWARD uses the message name used to call the currently active method.

If you specify the CLASS option, the *exprs* is a literal string, a constant symbol, or an expression enclosed in parentheses. This is the class object used as a superclass specifier on the forwarded message.

If you do not specify CLASS, the message is forwarded without a superclass override.

If you do not specify the CONTINUE option, the language processor immediately exits the current method before forwarding the message. Results returned from the forwarded message are the return value from the original message that called the active method (the caller of the method that issued the FORWARD instruction). Any conditions the forwarded message raises are raised in the calling program (without raising a condition in the method issuing the FORWARD instruction).

If you specify the CONTINUE option, the current method does not exit and continues with the next instruction when the forwarded message completes. If the forwarded message returns a result, the language processor assigns it to the special variable RESULT. If the message does not return a result, the language processor drops (uninitializes) the variable RESULT.

The FORWARD instruction passes all or part of an existing message invocation to another method. For example, the FORWARD instruction can forward a message to a different target object, using the same message name and arguments.

Example:

```
::method substr
forward to (self~string)       /* Forward to the string value */
```

You can use FORWARD in an UNKNOWN method to reissue to another object the message that the UNKNOWN method traps.

Example:

```
::method unknown
use arg msg, args
/* Forward to the string value */
/* passing along the arguments */
forward to (self~string) message (msg) arguments (args)
```

You can use FORWARD in a method to forward a message to a superclass's methods, passing the same arguments. This is very common usage in object INIT methods.

Example:

```
::class  savings subclass account
::method init
expose type penalty
forward class (super) continue         /* Send to the superclass      */
type = 'Savings'                       /* Now complete initialization */
penalty = '1% for balance under 500'
```

In the preceding example, the CONTINUE option causes the FORWARD message to continue with the next instruction, rather than exiting the Savings class INIT method.

GUARD

```
>>--GUARD--+-ON--+------------------+--+--;-----------------------------------><
           |     '-WHEN--expression-'  |
           '-OFF--+------------------+-'
                  '-WHEN--expression-'
```

GUARD controls a method's exclusive access to an object.

GUARD ON acquires for an active method exclusive use of its object variable pool. This prevents other methods that also require exclusive use of the same variable pool from running on the same object. If another method has already acquired exclusive access, the GUARD instruction causes the issuing method to wait until the variable pool is available.

GUARD OFF releases exclusive use of the object variable pool. Other methods that require exclusive use of the same variable pool can begin running.

If you specify WHEN, the method delays running until the *expression* evaluates to 1 (true). If the *expression* evaluates to 0 (false), GUARD waits until another method assigns or drops an object variable (that is, a variable named on an EXPOSE instruction) used in the WHEN *expression*. When an object variable changes, GUARD reevaluates the WHEN *expression*. If the *expression* evaluates to true, the method resumes running. If the *expression* evaluates to false, GUARD resumes waiting.

Example:

```
::method c
expose y
if y>0 then
  return 1
else
  return 0
::method d
expose z
guard on when z>0
self~c    /* Reevaluated when Z changes  */
say 'Method D'
```

If you specify WHEN and the method has exclusive access to the object's variable pool, then the exclusive access is released while GUARD is waiting for an object variable to change. Exclusive access is reacquired before the WHEN *expression* is evaluated. Once the WHEN *expression* evaluates to 1 (true), exclusive access is either retained (for GUARD ON WHEN) or released (for GUARD OFF WHEN), and the method resumes running.

Note: If the condition expression cannot be met, GUARD ON WHEN puts the program in a continuous wait condition. This can occur in particular when several activities run concurrently. See “Guarded Methods” on page 446 for more information.

IF

```
>>─IF─expression─┬───┬─THEN─┬───┬─instruction─┬──────────────────────────────┬──><
                 └─;─┘      └─;─┘             └─ELSE─┬───┬─instruction─┘
                                                     └─;─┘
```

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 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 ELSE.

Example:

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

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?'
```

Notes:

1. The *instruction* can be any assignment, message instruction, 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 THEN or ELSE is not equivalent to putting a dummy instruction (as it would be in C). 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 THEN, without a semicolon (;) being required.

INTERPRET

```
►►──INTERPRET──expression──;──────────────────────────────────────────────►◄
```

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

The *expression* is evaluated to produce a character string, 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.

Examples:

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

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

Notes:

1. Labels within the interpreted string are not permanent and are, therefore, an error.
2. Executing the INTERPRET instruction with `TRACE R` or `TRACE I` can be helpful in interpreting the results you get.

   Example:

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

   When this is run, you get the following trace:

```
C:\>REXXC 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!
C:\>
```

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

3. For many purposes, you can use the VALUE function (see “VALUE” on page 350) instead of the INTERPRET instruction. The following line could, therefore, have replaced line 5 in the previous 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.
4. You cannot use a directive (see “Chapter 3. Directives” on page 87) within an INTERPRET instruction.

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 continues with the current repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop, which can 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" */
```

Notes:

1. If specified, *name* must match the symbol naming the control variable in the DO clause in all respects except the 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 the execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. ITERATE cannot be used to continue with 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, contains 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 can be the innermost, and that loop, and any active loops inside it, are 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" */
```

Notes:

1. If specified, *name* must match the symbol naming the control variable in the DO clause in all respects except the 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;─────────────────────────────────────────────────────────────►◄
```

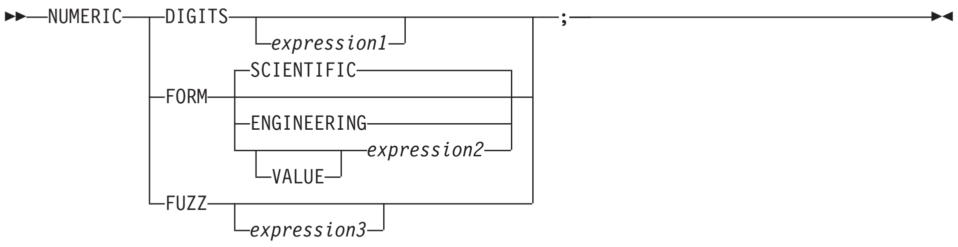
NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause.

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 changes the way in which a program carries out arithmetic operations. The options of this instruction are described in detail in "Chapter 11. Numbers and Arithmetic" on page 421.

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

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

You can retrieve the current NUMERIC DIGITS setting with the DIGITS built-in function. See "DIGITS" on page 322.

NUMERIC FORM
controls the form of exponential notation for the result of arithmetic operations and built-in functions. This can 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 character string 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 325.

NUMERIC FUZZ
controls how many digits, at full precision, are ignored during a numeric comparison operation. (See "Numeric Comparisons" on page 426.) If you omit *expression3*, the default is 0 digits. Otherwise, the character string value result of *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 327.

Note: The three numeric settings are automatically saved across internal subroutine and function calls. See the CALL instruction ("CALL" on page 45) for more details.

PARSE

```
►►──PARSE──┬───────┬──┬──────────┬──┬─ARG───────────────────────────┬──────────────────►
           ├─UPPER─┤  └─CASELESS─┘  ├─LINEIN────────────────────────┤
           └─LOWER─┘                ├─PULL──────────────────────────┤
                                    ├─SOURCE────────────────────────┤
                                    ├─VALUE──┬────────────┬──WITH───┤
                                    │        └─expression─┘         │
                                    ├─VAR──name─────────────────────┤
                                    └─VERSION───────────────────────┘

►──┬───────────────┬──;─────────────────────────────────────────────────────────────►◄
   └─template_list─┘
```

Note: You can specify UPPER and CASELESS or LOWER and CASELESS in either order.

PARSE assigns data from various sources to one or more variables according to the rules of parsing. (See "Chapter 10. Parsing" on page 403.)

If you specify UPPER or LOWER, any character strings to be parsed are first translated. Otherwise no translation takes place during the parsing. If you specify UPPER, the strings are translated to uppercase. If you specify LOWER, the strings are translated to lowercase.

If you specify CASELESS, character string matches during parsing are made independent of the case. This means a letter in uppercase is equal to the same letter in lowercase.

The *template_list* can be a single template or list of templates separated by commas. Each template consists of one or more symbols separated by blanks, patterns, or both.

Each template is applied to a single source string. Specifying several templates is not a syntax error, but only the PARSE ARG variant can supply more than one non-null source string. See "Parsing Several Strings" on page 414 for information on parsing several source strings.

If you do not specify a template, no variables are set but the data is prepared for parsing, if necessary. Thus for PARSE PULL, a data string is removed from the current data 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.

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

PARSE ARG
: parses the strings passed to a program or internal routine as input arguments. (See the ARG instruction in "ARG" on page 43 for details and examples.)

  Note: Parsing uses the argument string values. The USE ARG instruction provides access to string and non-string argument objects. You can also retrieve or check the argument objects to a REXX program or internal routine with the ARG built-in function (see "ARG (Argument)" on page 305).

PARSE LINEIN
: parses the next line of the default input stream. (See "Chapter 16. Input and Output Streams" on page 463 for a discussion of REXX input and output.) PARSE LINEIN is a shorter form of the following instruction:

```
>>--PARSE VALUE LINEIN()WITH--+---------------+--;--------------------><
                              '-template_list-'
```

  If no line is available, program execution usually pauses until a line is

complete. Use PARSE LINEIN only when direct access to the character input stream is necessary. Use the PULL or PARSE PULL instructions for the usual line-by-line dialog with the user 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 331 and “LINEIN (Line Input)” on page 328.

PARSE PULL
parses the next string of the external data queue. If the external data queue is empty, PARSE PULL reads a line of 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 333.) The queue remains active as long as the language processor is active. Other programs in the system can alter the queue and use it to communicate with programs written in REXX. See also the PULL instruction in “PULL” on page 69.

Note: PULL and PARSE PULL read the current data queue. If the queue is empty, they read the default input stream, STDIN (typically, the keyboard).

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

The source string contains the characters `Windows95` or `WindowsNT`, followed by either `COMMAND`, `FUNCTION`, `METHOD`, or `SUBROUTINE`, depending on whether the program was called as a host command or from a function call in an expression or as a method of an object or using the CALL instruction. These two tokens are followed by the complete path specification of the program file.

The string parsed might, therefore, look like this:

```
Windows95 COMMAND C:\MYDIR\REXXTRY.CMD
```
or
```
WindowsNT COMMAND C:\MYDIR\REXXTRY.CMD
```

PARSE VALUE
parses the data, a character string, that is the result of evaluating *expression*. If you specify no *expression*, 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 character string value of the variable *name*. The *name* must be a symbol that is valid as a variable name, which means 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*.

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

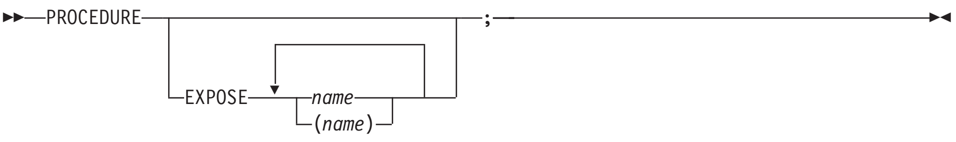
also translates the data from *string* to uppercase before it is parsed.

PARSE VERSION

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

- The string `OBJREXX`
- The language level description, for example `6.00`.
- Three tokens that describe the language processor release date in the same format as the default for the DATE built-in function (see "DATE" on page 318), for example, "27 Sep 1997".

PROCEDURE



PROCEDURE, within an internal routine (subroutine or function), protects the caller's 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 the 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 the PROCEDURE instruction is used, it 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 the *name* is exposed. Any reference to it (including setting and dropping) is made 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 the *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 character string value of *name* is immediately used as a subsidiary list of variables. Blanks are not necessary 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 from left to right. It is not an error to specify a name more than once, 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 of the caller's variables can be made accessible and can be changed, or new variables 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 is placed before `J`, the caller's value of `J` is not visible, so `Z.1` is not 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 a 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 30.)

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
```

Note: Variables can be exposed through several generations of routines if they are included in all intermediate PROCEDURE instructions.

See the CALL instruction and function descriptions in “CALL” on page 45 and “Chapter 8. Functions” on page 297 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 16. Input and Output Streams” on page 463 for a discussion of REXX input and output.) It is a short form of the following instruction:

```
►►──PARSE UPPER PULL──┬───────────────┬──;────────────────────────────►◄
                      └─template_list─┘
```

The current head of the queue is read as one string. Without a *template_list* specified, no further action is taken and the string is thus effectively discarded. The *template_list* can be a single template or list of templates separated by commas, but PULL parses only one source string. Each template consists of one or more symbols separated by blanks, patterns, or both.

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 “Chapter 10. Parsing” on page 403. Use the PARSE PULL instruction if you do not desire uppercase translation.

Note: If the current data queue is empty, PULL reads from the standard input (typically, the keyboard). If there is a PULL from the standard input, the program waits for keyboard input with no prompt. The length of data read by the PULL instruction is restricted to the length of strings contained by variables.

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 in 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 page 64.)

The QUEUED built-in function (see page “QUEUED” on page 333) 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) into the external data queue. (See “Chapter 16. Input and Output Streams” on page 463 for a discussion of REXX input and output.)

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

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 333) 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 16. Input and Output Streams” on page 463 for a discussion of REXX input and output.)

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

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 333) returns the number of lines currently in the external data queue.

RAISE

```
>>──RAISE──┬─condition──────────┬──┬─────────────┬──;──────────────────────><
           ├─ERROR──errorcode───┤  └─┤ options ├─┘
           ├─FAILURE──failurecode─┤
           ├─SYNTAX──number─────┤
           ├─USER──usercondition──┤
           └─PROPAGATE──────────┘
```

options:

```
                                                          ┌─┤ EXIT ├──────────┐
├──┬────────────────────────────┬──┬──────────────────────┬──┼──────────────────┼──┤
   ├─ADDITIONAL──expra──────────┤  └─DESCRIPTION──exprd─┘  └─RETURN──┬───────┬─┘
   │             ┌─,──────┐     │                                    └─exprr─┘
   └─ARRAY──(────▼─expri──┴──)──┘
```

EXIT:

```
├──EXIT──┬───────┬──────────────────────────────────────────────────────────┤
         └─expre─┘
```

Note: You can specify the options ADDITIONAL, ARRAY, DESCRIPTION, RETURN, and EXIT in any order. However, if you specify EXIT without *expre* or RETURN without *exprr*, it must appear last.

RAISE returns or exits from the currently running routine or method and raises a condition in the caller (for a routine) or sender (for a method). See “Chapter 12. Conditions and Condition Traps” on page 429 for details of the actions taken when conditions are raised. The RAISE instruction can raise all conditions that can be trapped.

If you specify *condition*, it is a single symbol that is taken as a constant.

If the ERROR or FAILURE condition is raised, you must supply the associated return code as *errorcode* or *failurecode*, respectively. These can be literal strings, constant symbols, or expressions enclosed in parentheses. If you specify an expression enclosed in parentheses, a subexpression, the language processor evaluates the expression to obtain its character string value.

If the SYNTAX condition is raised, you must supply the associated REXX error number as *number*. This error *number* can be either a REXX major error code or a REXX detailed error code in the form *nn.nnn*. The *number* can be a literal string, a constant symbol, or an expression enclosed in parentheses. If you specify an expression enclosed in parentheses, the language processor evaluates the expression to obtain its character string value.

If a USER condition is raised, you must supply the associated user condition name as *usercondition*. This can be a literal string or a symbol that is taken as a constant.

If you specify the ADDITIONAL option, the language processor evaluates *expra* to produce an object that supplies additional object information associated with the condition. The *expra* can be a literal string, constant symbol, or expression enclosed in parentheses. The ADDITIONAL entry of the condition object and the `"A"` option of the CONDITION built-in function return this additional object information. For SYNTAX conditions, the ADDITIONAL value must evaluate to a REXX array object.

If you specify the ARRAY option, each *expri* is an expression (use commas to separate the expressions). The language processor evaluates the expression list to produce an array object that supplies additional object information associated with the condition. The ADDITIONAL entry of the condition object and the `"A"` option of the CONDITION built-in function return this additional object information as an array of values. It is an error to use both the ARRAY option and the ADDITIONAL option on the same RAISE instruction.

The content of *expra* or *expri* is used as the contents of the secondary error message produced for a *condition*.

If you specify neither ADDITIONAL nor ARRAY, there is no additional object information associated with the condition.

If you specify the DESCRIPTION option, the *exprd* can be a literal string, a constant symbol, or an expression enclosed in parentheses. If you specify an expression enclosed in parentheses, the language processor evaluates the expression to obtain its character string value. This is the description associated with the condition. The `"D"` option of the CONDITION built-in function and the DESCRIPTION entry of the condition object return this string.

If you do not specify DESCRIPTION, the language processor uses a null string as the descriptive string.

If you specify the RETURN or EXIT option, the language processor evaluates the expression *exprr* or *expre*, respectively, to produce a result object that is passed back to the caller or sender as if it were a RETURN or EXIT result. The *expre* or *exprr* is a literal string, constant symbol, or expression enclosed in parentheses. If you specify an expression enclosed in parentheses, the language processor evaluates the expression to obtain its character string value. If you do not specify *exprr* or *expre*, no result is passed back to the caller or sender. In either case, the effect is the same as that of the RETURN or EXIT instruction (see "RETURN" on page 74). Following the return or exit, the appropriate action is taken in the caller or sender (see "Action Taken when a Condition Is Not Trapped" on page 432). If specified, the result value can be obtained from the RESULT entry of the condition object.

Examples:

```
raise syntax 40                     /* Raises syntax error 40              */
raise syntax 40.12 array (1, number) /* Raises syntax error 40, subcode 12   */
                                    /* Passing two substitution values     */
raise syntax (errnum)               /* Uses the value of the variable ERRNUM */
                                    /* as the syntax error number          */
raise user badvalue                 /* Raises user condition BADVALUE      */
```

If you specify PROPAGATE, and there is a currently trapped condition, this condition is raised again in the caller (for a routine) or sender (for a method). Any ADDITIONAL, DESCRIPTION, ARRAY, RETURN, or EXIT information specified on the RAISE instruction replaces the corresponding values for the currently trapped condition. A SYNTAX error occurs if no condition is currently trapped.

Example:

```
signal on syntax
a = 'xyz'
c = a+2              /* Raises the SYNTAX condition            */
:
exit
syntax:
raise propagate      /* Propagates SYNTAX information to caller */
```

REPLY

```
>>──REPLY──┬────────────┬──;──────────────────────────────────────────><
           └─expression─┘
```

REPLY sends an early reply from a method to its caller. The method issuing REPLY returns control, and possibly a result, to its caller to the point from which the message was sent; meanwhile, the method issuing REPLY continues running.

If you specify *expression*, it is evaluated and the object resulting from the evaluation is passed back. If you omit *expression*, no object is passed back.

Unlike RETURN or EXIT, the method issuing REPLY continues to run after the REPLY until it issues an EXIT or RETURN instruction. The EXIT or RETURN must not specify a result expression.

Example:

```
reply 42           /* Returns control and a result    */
call tidyup        /* Can run in parallel with sender */
return
```

Notes:

1. You can use REPLY only in a method.
2. A method can execute only one REPLY instruction.
3. When the method issuing the REPLY instruction is the only method on the current activity with exclusive access to the object's variable pool, the method retains exclusive access on the new activity. When the other methods on the activity also have access, the method issuing REPLY releases its access and reacquires the access on the new activity. This might force the method to wait until the original activity has released its access.

See “Chapter 13. Concurrency” on page 439 for a complete description of concurrency.

RETURN

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

RETURN returns control, and possibly a result, from a REXX program, method, 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 run. (See “EXIT” on page 51.)

If a subroutine is run, *expression* (if any) is evaluated, control is passed back to the caller, and the REXX special variable RESULT is set to the value of *expression*. If you omit *expression*, the special variable RESULT is dropped (becomes uninitialized). The various settings saved at the time of the CALL (for example, tracing and addresses) are also restored. (See “CALL” on page 45.)

If a function is 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 8. Functions” on page 297 for more details.

If a method is processed, the language processor evaluates *expression* (if any) and returns control to the point from which the method's activating message was sent. If called as a term of an expression, *expression* is required. If called as a message instruction, *expression* is optional and is assigned to the REXX special variable RESULT if you specify it. If the method has previously issued a REPLY instruction, the RETURN instruction must not include a result *expression*.

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, which displays it to the user. However, the output destination can depend on the implementation. See “Chapter 16. Input and Output Streams” on page 463 for a discussion of REXX input and output. The string value of the *expression* result is written to the default character output stream. The resulting string can be of any length. If you omit *expression*, the null string is written.

The SAY instruction is a shorter form of the following 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 330 for details of the LINEOUT function.

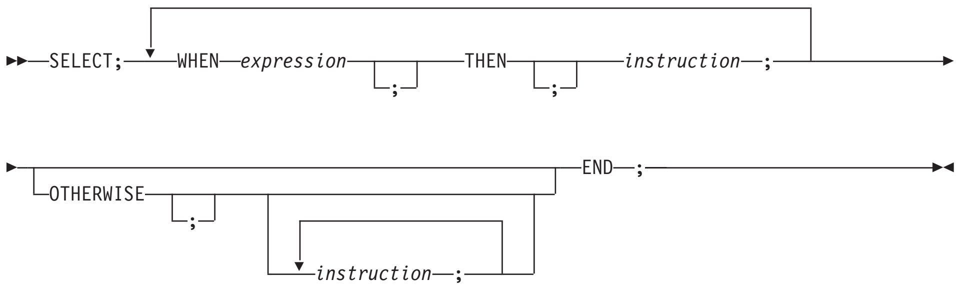
Example:

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

Notes:

1. Data from the SAY instruction is sent to the default output stream (STDOUT). However, the standard rules for redirecting output apply to the SAY output.
2. The SAY instruction does not format data; the operating system and the hardware handle line wrapping. However, formatting is accomplished, the output data remains a single logical line.

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 can be a complex instruction such as IF, DO, or SELECT) is processed and control is then passed to the END. If the result is 0, control is passed to the next WHEN clause.

If none of the WHEN expressions evaluates to 1, control is passed to the instructions, if any, after OTHERWISE. In this situation, the absence of an OTHERWISE produces an error, however, 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 */
```

Notes:

1. The *instruction* can be any assignment, command, message instruction, 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 semicolon (;).

SIGNAL

```
►►──SIGNAL──┬─labelname──────────────────────────────────────┬─;────────────►◄
            ├─┬───────┬─expression───────────────────────────┤
            │ └─VALUE─┘                                      │
            ├─OFF───┬─ANY──────────────────┬─────────────────┤
            │       ├─ERROR────────────────┤                 │
            │       ├─FAILURE──────────────┤                 │
            │       ├─HALT─────────────────┤                 │
            │       ├─LOSTDIGITS───────────┤                 │
            │       ├─NOMETHOD─────────────┤                 │
            │       ├─NOSTRING─────────────┤                 │
            │       ├─NOTREADY─────────────┤                 │
            │       ├─NOVALUE──────────────┤                 │
            │       ├─SYNTAX───────────────┤                 │
            │       └─USER──usercondition──┘                 │
            └─ON───┬─ANY──────────────────┬─┬──────────────────┬─┘
                   ├─ERROR────────────────┤ └─NAME──trapname──┘
                   ├─FAILURE──────────────┤
                   ├─HALT─────────────────┤
                   ├─LOSTDIGITS───────────┤
                   ├─NOMETHOD─────────────┤
                   ├─NOSTRING─────────────┤
                   ├─NOTREADY─────────────┤
                   ├─NOVALUE──────────────┤
                   ├─SYNTAX───────────────┤
                   └─USER──usercondition──┘
```

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 12. Conditions and Condition Traps” on page 429.

To change the flow of control, a label name is derived from *labelname* or taken from the character string 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 specify a symbol for *labelname*, the search looks for a label with uppercase characters. If you specify a literal string, the search uses the literal string directly. You can locate label names with lowercase letters only if you specify the label as a literal string with the same case. Similarly, for SIGNAL VALUE, the lettercase of *labelname* must match exactly. 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 and cannot be resumed.

Control is then passed to the first label in the program that matches the given name, as though the search had started at the beginning of the program.

The *labelname* and *usercondition* are single symbols, which are taken as constants. The *trapname* is a string or symbol taken as a constant.

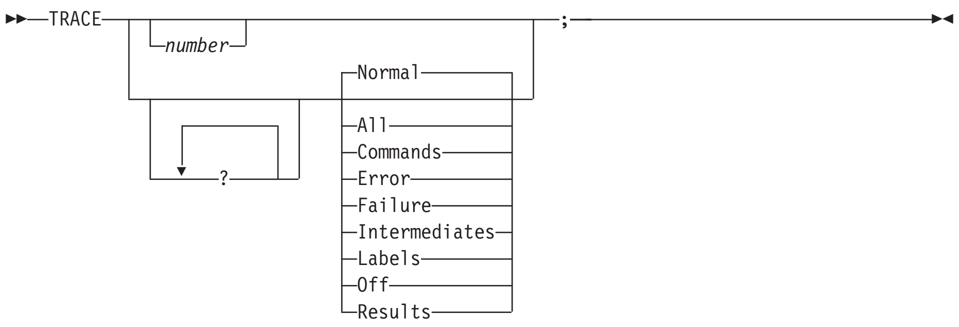
Example:

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

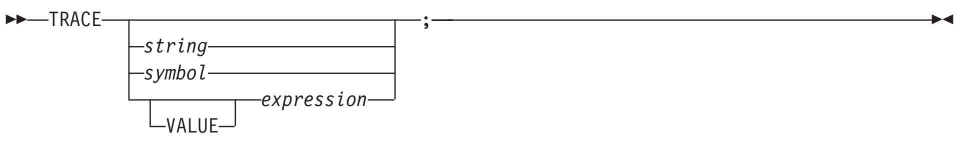
If there are duplicates, control is always passed to the first occurrence of the label in the program.

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.

TRACE



Or, alternatively:



TRACE controls the tracing action (that is, how much is displayed to the user) during the 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.)

Note: TRACE cannot be used in the REXX macrospace. See “Trace in Macrospace” on page 525.

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 in “Alphabetic Character (Word) Options”
- 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 in “Alphabetic Character (Word) Options”

The option that follows TRACE or the character string that is 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 first capitalized letter is needed; all following characters 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 37), tracing also displays the return code from the command.

Error Traces any command resulting in an error or failure after execution (see “Commands” on page 37), together with the return code from the command.

Failure
Traces any command resulting in a failure after execution (see

"Commands" on page 37), 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 the 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 failing command after execution, together with the return code from the command. This is the default setting.

For the default Windows command processor, an attempt to enter an unknown command raises a FAILURE condition. The CMD return code for an unknown command is 1. An attempt to enter a command in an unknown command environment also raises a FAILURE condition; in such a case, the variable RC is set to 30.

Off Traces nothing and resets the special prefix option (described later) to OFF.

Results
Traces all clauses before execution. Displays the final results (in contrast with `Intermediates` option) of the expression evaluation. Also displays values assigned during PULL, ARG, PARSE, and USE instructions. This setting is recommended for general debugging.

Prefix Option

The prefix `?` is valid alone or with one of the alphabetic character options. You can specify the prefix more than once, if desired. Each occurrence of a prefix on an instruction reverses the action of the previous prefix. The prefix must immediately precede the option (no intervening blanks).

The prefix `?` controls interactive debugging. During normal execution, a TRACE option with a prefix of `?` causes interactive debugging to be switched on. (See "Chapter 17. Debugging Aids" on page 475 for full details of this facility.) When interactive debugging 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 or explicit setting of the error condition by the command handler.

Any TRACE instructions in the program being traced are ignored to ensure that you are not taken out of interactive debugging unexpectedly.

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

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

Numeric Options

If interactive debugging is active and 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. (See "Chapter 17. Debugging Aids" on page 475 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 displayed. After that, tracing resumes as before.

Tracing Tips:
- When a loop is traced, the DO clause itself is traced on every iteration of the loop.
- You can retrieve the trace actions currently in effect by using the TRACE built-in function (see "TRACE" on page 348).
- The trace output of commands traced before execution always contains the final value of the command, that is, the string passed to the environment, and the clause generating it.
- Trace actions are automatically saved across subroutine, function, and method calls. See "CALL" on page 45 for more details.

Example: One of the most common traces you will use is:

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

The Format of Trace Output: Every clause traced appears with automatic formatting (indentation) according to its logical depth of nesting, for example. Results, if requested, are indented by two extra spaces and are enclosed in double quotation marks so that leading and trailing blanks are apparent. Any control codes in the data encoding (ASCII values less than '20'x) are replaced by a question mark (?) to avoid screen interference. Results other than strings appear in the string representation obtained by sending them a `STRING` message. The resulting string is enclosed in parentheses. The line number in the program precedes the first clause traced on any line. 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 can be the nonzero return code from a command, the prompt message when interactive debugging is entered, an indication of a syntax error when in interactive debugging.
`>>>`	Identifies the result of an expression (for `TRACE R`) or the value assigned to a variable during parsing, the value returned from a subroutine call, or a value evaluated by execution of a DO loop.
`>.>`	Identifies the value assigned to a placeholder during parsing (see "The Period as a Placeholder" on page 405).

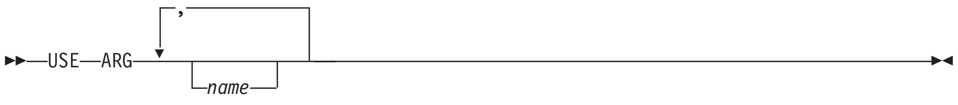
The following prefixes are used only if `TRACE Intermediates` is in effect:

`>C>`	The data traced is the name of a compound variable, after the name has been replaced by the value of the variable but before the variable is used. If no value was assigned to the variable, the trace shows the variable in uppercase characters.
`>F>`	The data traced is the result of a function call.
`>L>`	The data traced is a literal (string, uninitialized variable, or constant symbol).
`>M>`	The data traced is the result of a message.
`>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 and interactive debugging (?) off.

Following a syntax error that SIGNAL ON SYNTAX does not trap, the clause in error is always traced.

USE



USE ARG retrieves the argument objects provided in a program, routine, function, or method and assigns them to variables.

Each *name* must be a valid variable name. The *name*s are assigned from left to right. For each *name* you specify, the language processor assigns it a corresponding argument from the program, routine, function, or method call. If there is no corresponding argument, *name* is dropped.

A USE ARG instruction can be processed repeatedly and it always accesses the same current argument data.

Examples:

```
/* USE Example                      */
/* FRED('Ogof X',1,5) calls function */
Fred: use arg string, num1, num2

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

```
/* Another example, shows how to pass non-string arguments with USE ARG */
/* Pass a stem and an array to a routine to modify one element of each  */
stem.1 = 'Value'
array = .array~of('Item')
say 'Before subroutine:' stem.1 array[1]  /* Shows "Value Item"        */
Call Change_First stem. , array
say 'After subroutine:' stem.1 array[1]   /* Shows "NewValue NewItem"   */
Exit
Change_First: Procedure
  Use Arg substem., subarray
  substem.1 = 'NewValue'
  subarray[1] = 'NewItem'
  Return
```

You can retrieve or check the arguments by using the ARG built-in function (see “ARG (Argument)” on page 305). The ARG and PARSE ARG instructions are alternative ways of retrieving arguments. ARG and PARSE ARG access the string values of arguments. USE ARG performs a direct, one-to-one assignment of arguments to REXX variables. This is preferable when you need an exact copy of the argument, without translation or parsing. USE ARG also allows access to both string and non-string argument objects; ARG and PARSE ARG parse the string values of the arguments.

Chapter 3. Directives

A REXX program contains one or more executable code units. *Directive instructions* separate these executable units. A directive begins with a double colon (::) and is a nonexecutable instruction. For example, it cannot appear in a string for the INTERPRET instruction to be interpreted. The first directive instruction in a program marks the end of the main executable section of the program.

For a program containing directives, all directives are processed first to set up the program's classes, methods, and routines. Then any program code in the main code unit (preceding the first directive) is processed. This code can use any classes, methods, and routines that the directives established.

::CLASS

```
                                                        ┌─SUBCLASS──Object────┐
►►──::CLASS──classname──┬──────────────────────────┬───┼──────────────────────┼──────────────►
                        └─METACLASS──metaclass─────┘   ├─MIXINCLASS──mclass───┤
                                                        └─SUBCLASS──sclass─────┘

►──┬────────┬──┬─────────────────────┬──;──────────────────────────────────────────────────►◄
   └─PUBLIC─┘  └─INHERIT──iclasses───┘
```

Notes:

1. You can specify the options EXTERNAL, METACLASS, MIXINCLASS, SUBCLASS, and PUBLIC in any order.
2. If you specify INHERIT, it must be the last option.

The ::CLASS directive creates a REXX class named *classname*. The *classname* is a literal string or symbol that is taken as a constant. The created class is available to programs through the REXX environment symbol *.classname*. The *classname* acquires all methods defined by subsequent ::METHOD directives until the end of the program or another ::CLASS directive is found. Only null clauses (comments or blank lines) can appear between a ::CLASS directive and any following directive instruction or the end of the program. Only one ::CLASS directive can appear for *classname* in a program.

If you specify the EXTERNAL option, the class is created using information derived from an external source named *extname*. The *extname* is a literal string.

If you specify the METACLASS option, the instance methods of the *metaclass* class become class methods of the *classname* class. (See “Chapter 4. Objects and Classes” on page 95.) The *metaclass* and *classname* are literal strings or symbols that are taken as constants. In the search order for methods, the metaclass methods precede inherited class methods and follow any class methods defined by ::METHOD directives with the CLASS option.

If you specify the PUBLIC option, the class is visible beyond its containing REXX program to any other program that references this program with a ::REQUIRES directive. (See “::REQUIRES” on page 91.) If you do not specify the PUBLIC option, the class is visible only within its containing REXX program. All public classes defined within a program are used before PUBLIC classes created with the same name.

If you specify the SUBCLASS option, the class becomes a subclass of the class *sclass* for inheritance of instance and class methods. The *sclass* is a literal string or symbol that is taken as a constant.

If you specify the MIXINCLASS option, the class becomes a subclass of the class *mclass* for inheritance of instance and class methods. You can add the new class instance and class methods to existing classes by using the INHERIT option on a ::CLASS directive or by sending an INHERIT message to an existing class. If you specify neither the SUBCLASS nor the MIXINCLASS option, the class becomes a non-mixin subclass of the Object class.

If you specify the INHERIT option, the class inherits instance methods and class methods from the classes *iclasses* in their order of appearance (leftmost first). This is equivalent to sending a series of INHERIT messages to the class object, with each INHERIT message (except the first) specifying the preceding class in *iclasses* as the *classpos* argument. (See “INHERIT” on page 164.) As with the INHERIT message, each of the classes in *iclasses* must be a mixin class. The *iclasses* is a blank-separated list of literal strings or symbols that are taken as constants. If you omit the INHERIT option, the class inherits only from *sclass*.

Example:

```
::class rectangle
::method area    /* defined for the RECTANGLE class */
expose width height
return width*height

::class triangle
::method area    /* defined for the TRIANGLE class */
expose width height
return width*height/2
```

The ::CLASS directives in a program are processed in the order in which they appear. If a ::CLASS directive has a dependency on ::CLASS directives that appear later in the program, processing of the directive is deferred until all of the class's dependencies have been processed.

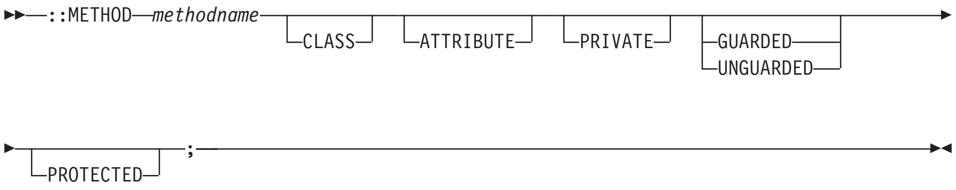
Example:

```
::class savings subclass account  /* requires the ACCOUNT class */
::method type
return "a Savings Account"

::class account
::method type
return "an Account"
```

The Savings class in the preceding example is not created until the Account class that appears later in the program has been created.

Note: If you specify the same ::CLASS *classname* more than once in different programs, the last one is used. Using more than one ::CLASS *classname* in the same program produces an error.

::METHOD



Note: You can specify all options in any order.

The ::METHOD directive creates a method object and defines the method attributes.

A ::METHOD directive starts a method, which is ended by another directive or the end of the program. The ::METHOD is not included in the method source.

The *methodname* is a literal string or a symbol that is taken as a constant. The method is defined as *methodname* in the class specified in the most recent ::CLASS directive. Only one ::METHOD directive can appear for any *methodname* in a class.

A ::CLASS directive is not required before a ::METHOD directive. If no ::CLASS directive precedes ::METHOD, the method is not associated with a class but is accessible to the main (executable) part of a program through the .METHODS built-in object. Only one ::METHOD directive can appear for any method name not associated with a class. See ".METHODS" on page 455 for more details.

If you specify the CLASS option, the method is a *class method*. See "Chapter 4. Objects and Classes" on page 95. The method is associated with the class specified on the most recent ::CLASS directive. The ::CLASS directive is required in this case.

If you specify the PRIVATE option, the method is a *private method*. (Only a message the same object sends can activate the method.) If you omit the PRIVATE option, the method is a *public method* that any sender can activate.

If you specify the UNGUARDED option, the method can be called while other methods are active on the same object. If you do not specify UNGUARDED, the method requires exclusive use of the object variable pool; it can run only if no other method that requires exclusive use of the object variable pool is active on the same object.

If you specify the ATTRIBUTE option, in addition to having a method created as *methodname* in the class specified in the most recent ::CLASS directive, another method is also automatically created in that same class as *methodname=*.

For example, the directive

```
::method name attribute
```

creates two methods, NAME and NAME=. The NAME and NAME= methods are equivalent to the following code sequences:

```
::method 'NAME='
expose name
use arg name

::method name
expose name
return name
```

If you specify the PROTECTED option, the method is a protected method. (See "Chapter 15. The Security Manager" on page 457 for more information.) If you omit the PROTECTED option, the method is not protected.

If you specify ATTRIBUTE, another directive (or the end of the program) must follow the ::METHOD directive.

Example:

```
r = .rectangle~new(20,10)
say 'Area is' r~area       /* Produces "Area is 200" */

::class rectangle

::method area
expose width height
return width*height

::method init
expose width height
use arg width, height

::method perimeter
expose width height
return (width+height)*2
```

Note: It is an error to specify ::METHOD more than once within the same class and use the same *methodname*.

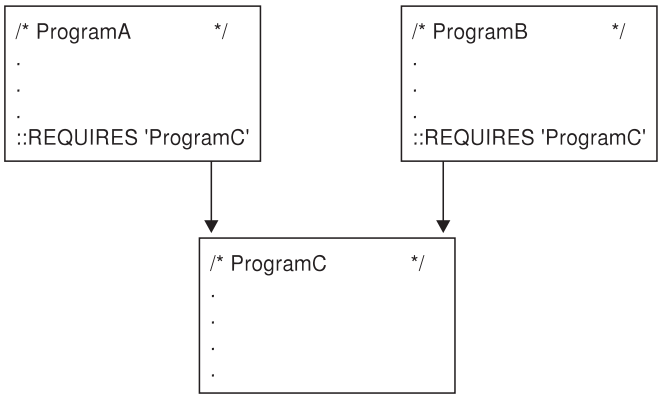
::REQUIRES

```
►►──::REQUIRES──'programname' ──;──────────────────────────────────────────►◄
```

The ::REQUIRES directive specifies that the program requires access to the classes and objects of the REXX program *programname*. All public classes and routines defined in the named program are made available to the executing program. The *programname* is a literal string or a symbol that is taken as a constant. The string or symbol *programname* can be any string or symbol that is valid as the target of a CALL instruction. The program *programname* is called as an external routine with no arguments. The main program code, which precedes the first directive instruction, is run.

Any ::REQUIRES directive must precede all ::CLASS, ::METHOD, and ::ROUTINE directives. The order of ::REQUIRES directives determines the search order for classes and routines defined in the named programs.

The following example illustrates that two programs, ProgramA and ProgramB, can both access classes and routines that another program, ProgramC, contains. (The code at the beginning of ProgramC runs.)



The language processor uses local routine definitions within a program in preference to routines of the same name accessed through ::REQUIRES directives. Local class definitions within a program override classes of the same name in other programs accessed through ::REQUIRES directives.

Another directive, or the end of the program, must follow a ::REQUIRES directive. Only null clauses can appear between them.

::ROUTINE

```
►►──::ROUTINE──routinename──┬────────┬──;───────────────────────────────────►◄
                            └─PUBLIC─┘
```

The ::ROUTINE directive creates named routines within a program. The *routinename* is a literal string or a symbol that is taken as a constant. Only one ::ROUTINE directive can appear for any *routinename* in a program.

A ::ROUTINE directive starts a routine, which is ended by another directive or the end of the program.

If you specify the PUBLIC option, the routine is visible beyond its containing REXX program to any other program that references this program with a ::REQUIRES directive. If you do not specify the PUBLIC option, the routine is visible only within its containing REXX program.

Routines you define with the ::ROUTINE directive behave like external routines. In the search order for routines, they follow internal routines and built-in functions but precede all other external routines.

Example:

```
::class c
::method a
call r 'A'  /* displays "In method A" */

::method b
call r 'B'  /* displays "In method B" */

::routine r
use arg name
say 'In method' name
```

Notes:

1. It is an error to specify ::ROUTINE with the same routine name more than once in the same program. It is not an error to have a local ::ROUTINE with the same name as another ::ROUTINE in another program that the ::REQUIRES directive accesses. The language processor uses the local ::ROUTINE definition in this case.
2. Calling an external REXX program as a function is similar to calling an internal routine. For an external routine, however, the caller's variables are hidden and the internal values (NUMERIC settings, for example) start with their defaults.

Chapter 4. Objects and Classes

This chapter provides an overview of the REXX class structure.

A REXX object consists of object methods and object variables. Sending a message to an object causes the object to perform some action; a method whose name matches the message name defines the action that is performed. Only an object's methods can access the object variables belonging to an object. EXPOSE instructions within an object's methods specify object variables. Any variables not exposed are dropped on return from a method.

You can create an object by sending a message to a class object. An object created from a class is an *instance* of that class. Classes define the methods and method names for their instances. The methods a class defines for its instances are called the *instance methods* of that class. These are the object methods for the instances. Classes can also define *class methods*, which are a class's own object methods.

Note: When referring to object methods (for objects other than classes) or instance methods (for classes), this book uses the term *methods* when the meaning is clear from the context. When referring to object methods and class methods of classes, this book uses the qualified terms to avoid possible confusion.

Types of Classes

There are three kinds of classes:
- Object classes
- Mixin classes
- Abstract classes

The following sections explain these.

Object Classes

An *object class* is like a factory for producing objects. An object class creates objects (instances) and provides methods that these objects can use. An object acquires the instance methods of the class to which it belongs at the time of its creation. If a class gains additional methods, objects created before the definition of these methods do *not* acquire these methods.

Because the object methods also define the object variables, object classes are factories for creating REXX objects. The Array class (see "The Array Class" on page 118) is an example of an object class.

Mixin Classes

Classes can inherit from more than the single superclass fromwhich they were created. This is called *multiple inheritance*. Classes designed to add a set of instance and class methods to other classes are called *mixin classes*, or simply *mixins*.

You can add mixin methods to an existing class by sending an INHERIT message or using the INHERIT option on the ::CLASS directive. (See "Chapter 3. Directives" on page 87.) In either case, the class to be inherited must be a mixin. During both class creation and multiple inheritance, subclasses inherit both class and instance methods from their superclasses.

Mixins are always associated with a *base class*, which is the mixin's first non-mixin superclass. Any subclass of the mixin's base class can (directly or indirectly) inherit a mixin; other classes cannot.

To create a new mixin class, you send a MIXINCLASS message to an existing class or use the ::CLASS directive with the MIXINCLASS option. A mixin class is also an object class and can create instances of the class.

Abstract Classes

Abstract classes provide definitions for instance methods and class methods but are not intended to create instances. Abstract classes often define the message interfaces that subclasses should implement.

You create an abstract class like object or mixin classes. No extra messages or keywords on the ::CLASS directive are necessary. REXX does not prevent users from creating instances of abstract classes.

Metaclasses

A *metaclass* is a class you can use to create another class. The only metaclass that REXX provides is .class, the Class class. The Class class is the metaclass of all the classes REXX provides. This means that instances of .class are themselves classes. The Class class is like a factory for producing the factories that produce objects.

To change the behavior of an object that is an instance, you generally use subclassing. For example, you can create Statarray, a subclass of the Array class (see "The Array Class" on page 118). The Statarray class can include a method for computing a total of all the numeric elements of an array.

```
/* Creating an array subclass for statistics */

::class statarray subclass array public

::method init    /*  Initialize running total and forward to superclass */
expose total
total = 0
```

```
/* "INIT" on page 171  describes the INIT method.  */
forward class (super)

::method put     /*  Modify to increment running total */
expose total
use arg value
total = total + value  /* Should verify that value is numeric!!! */
forward class (super)

::method '[]='   /*  Modify to increment running total */
forward message 'PUT'

::method remove  /*  Modify to decrement running total */
expose total
use arg index
forward message 'AT' continue
total = total - result
forward class (super)

::method average /*  Return the average of the array elements */
expose total
return total / self~items

::method total   /*  Return the running total of the array elements */
expose total
return total
```

You can use this method on the individual array *instances*, so it is an *instance method*.

However, if you want to change the behavior of the factory producing the arrays, you need a new *class* method. One way to do this is to use the ::METHOD directive with the CLASS option. Another way to add a class method is to create a new metaclass that changes the behavior of the Statarray class. A new metaclass is a subclass of .class.

You can use a metaclass by specifying it in a SUBCLASS or MIXINCLASS message or on a ::CLASS directive with the METACLASS option.

If you are adding a highly specialized class method useful only for a particular class, use the ::METHOD directive with the CLASS option. However, if you are adding a class method that would be useful for many classes, such as an instance counter that counts how many instances a class creates, you use a metaclass.

The following examples add a class method that keeps a running total of instances created. The first version uses the ::METHOD directive with the CLASS option. The second version uses a metaclass.

Version 1

```
/* Adding a class method using ::METHOD */

a = .point~new(1,1)                   /* Create some point instances  */
say 'Created point instance' a
b = .point~new(2,2)
say 'Created point instance' b
c = .point~new(3,3)
say 'Created point instance' c
                                  /* Ask the point class how many */
                                  /* instances it has created     */
say 'The point class has created' .point~instances 'instances.'



::class point public                /* Create Point class           */

::method init class
expose instanceCount
instanceCount = 0                   /* Initialize instanceCount     */
forward class (super)               /* Forward INIT to superclass   */

::method new class
expose instanceCount                /* Creating a new instance      */
instanceCount = instanceCount + 1   /* Bump the count               */
forward class (super)               /* Forward NEW to superclass    */

::method instances class
expose instanceCount                /* Return the instance count    */
return instanceCount


::method init
expose xVal yVal                    /* Set object variables         */
use arg xVal, yVal                  /* as passed on NEW             */

::method string
expose xVal yVal                    /* Use object variables         */
return '('xVal','yVal')'            /* to return string value       */
```

Version 2

```
/* Adding a class method using a metaclass  */

a = .point~new(1,1)     /* Create some point instances  */
say 'Created point instance' a
b = .point~new(2,2)
say 'Created point instance' b
c = .point~new(3,3)
say 'Created point instance' c
                                       /* Ask the point class how many */
                                       /* instances it has created     */
say 'The point class has created' .point~instances 'instances.'


::class InstanceCounter subclass class /* Create a new metaclass that */
```

```
                                       /* will count its instances     */
::method init
expose instanceCount
instanceCount = 0                      /* Initialize instanceCount     */
forward class (super)                  /* Forward INIT to superclass   */

::method new
expose instanceCount                   /* Creating a new instance      */
instanceCount = instanceCount + 1      /* Bump the count               */
forward class (super)                  /* Forward NEW to superclass    */

::method instances
expose instanceCount                   /* Return the instance count    */
return instanceCount


::class point public metaclass InstanceCounter  /* Create Point class */
                                       /* using InstanceCounter metaclass */
::method init
expose xVal yVal                       /* Set object variables         */
use arg xVal, yVal                     /* as passed on NEW             */

::method string
expose xVal yVal                       /* Use object variables         */
return '('xVal','yVal')'               /* to return string value       */
```

Creating Classes and Methods

You can define a class using either directives or messages.

To define a class using directives, you place a ::CLASS directive at the end of your source program:

```
::class 'Account'
```

This creates an Account class that is a subclass of the Object class. (See "The Object Class" on page 179 for a description of the Object class.) The string "Account" is a string identifier for the new class.

Now you can use ::METHOD directives to add methods to your new class. The ::METHOD directives must immediately follow the ::CLASS directive that creates the class.

```
::method type
return "an account"

::method 'name='
expose name
use arg name

::method name
expose name
return name
```

This adds the methods TYPE, NAME, and NAME= to the Account class.

You can create a subclass of the Account class and define a method for it:

```
::class 'Savings' subclass account
::method type
return "a savings account"
```

Now you can create an instance of the Savings class with the NEW method (see "NEW" on page 168) and send TYPE, NAME, and NAME= messages to that instance:

```
asav = .savings~new
say asav~type
asav~name = 'John Smith'
```

The Account class methods NAME and NAME= create a pair of access methods to the account object variable NAME. The following directive sequence creates the NAME and NAME= methods:

```
::method 'name='
expose name
use arg name

::method name
expose name
return name
```

You can replace this with a single ::METHOD directive with the ATTRIBUTE option. For example, the directive

```
::method name attribute
```

adds two methods, NAME and NAME= to a class. These methods perform the same function as the NAME and NAME= methods in the original example. The NAME method returns the current value of the object variable NAME; the NAME= method assigns a new value to the object variable NAME.

Using Classes

When you create a new class, it is always a subclass of an existing class. You can create new classes with the ::CLASS directive or by sending the SUBCLASS or MIXINCLASS message to an existing class. If you specify neither the SUBCLASS nor the MIXINCLASS option on the ::CLASS directive, the superclass for the new class is the Object class, and it is not a mixin class.

Example of creating a new class using a message:

```
persistence = .object~mixinclass('Persistence')
myarray=.array~subclass('myarray')~~inherit(persistence)
```

Example of creating a new class using the directive:

```
::class persistence mixinclass object
::class myarray subclass array inherit persistence
```

Scope

A *scope* is the methods and object variables defined in a single class. Only methods defined in a particular scope can access object variables within that scope. This means that object variables in a subclass can have the same names as object variables in a superclass, because the object variables are at different scopes.

Defining Instance Methods with SETMETHOD or ENHANCED

In REXX, methods are usually associated with instances using classes, but it is also possible to add methods directly to an instance using the SETMETHOD (see “SETMETHOD” on page 183) or ENHANCED (see “ENHANCED” on page 164) method.

All subclasses of the Object class inherit SETMETHOD. You can use SETMETHOD to create *one-off* objects, objects that must be absolutely unique so that a class that is capable of creating other instances is not necessary. The Class class also provides an ENHANCED method that lets you create new instances of a class with additional methods. The methods and the object variables defined on an object with SETMETHOD or ENHANCED form a separate scope, like the scopes the class hierarchy defines.

Method Names

A method name can be any string. When an object receives a message, the language processor searches for a method whose name matches the message name in uppercase.

Note: The language processor also translates the specified name of all methods added to objects into uppercase characters.

You must surround a method name with quotation marks when it contains characters that are not allowed in a symbol (for example, the operator characters). The following example creates a new class (the Cost class), defines a new method (%), creates an instance of the Cost class (mycost), and sends a `%` message to `mycost`:

```
cost=.object~subclass('A cost')
cost~define('%', 'expose p; say "Enter a price."; pull p; say p*1.07;')
mycost=cost~new
mycost~'%'       /* Produces:  Enter a price.            */
                 /* If the user specifies a price of 100, */
                 /* produces: 107.00                     */
```

Default Search Order for Method Selection

The search order for a method name matching the message is for:

1. A method the object itself defines with SETMETHOD or ENHANCED.(See “SETMETHOD” on page 183.)
2. A method the object's class defines. (Note that an object acquires the instance methods of the class to which it belongs at the time of its creation. If a class gains additional methods, objects created before the definition of these methods do *not* acquire these methods.)
3. A method that a superclass of the object's class defines. This is also limited to methods that were available when the object was created. The order of the INHERIT (see “INHERIT” on page 164) messages sent to an object's class determines the search order of the superclass method definitions.

This search order places methods of a class before methods of its superclasses so that a class can supplement or override inherited methods.

If the language processor does not find a match for the message name, the language processor checks the object for a method name UNKNOWN. If it exists, the language processor calls the UNKNOWN method and returns as the message result any result the UNKNOWN method returns. The UNKNOWN method arguments are the original message name and a REXX array containing the original message arguments.

If the object does not have an UNKNOWN method, the language processor raises a NOMETHOD condition.

Defining an UNKNOWN Method

When an object that receives a message does not have a matching message name, the language processor checks if the object has a method named UNKNOWN. If the object has an UNKNOWN method, the language processor calls UNKNOWN, passing two arguments. The first argument is the name of the method that was not located. The second argument is an array containing the arguments passed with the original message.

If you define an UNKNOWN method, you can use the following syntax:

```
►►──UNKNOWN(messagename,messageargs)──────────────────────────────────►◄
```

Changing the Search Order for Methods

You can change the usual search order for methods by:

1. Ensuring that the receiver object is the sender object. (You usually do this by specifying the special variable SELF—see page 481.)
2. Specifying a colon and a class symbol after the message name. The class symbol can be a variable name or an environment symbol. It identifies the class object to be used as the starting point for the method search.

The class object must be a superclass of the class defining the active method, or, if you used SETMETHOD to define the active method, the object's own class. The class symbol is usually the special variable SUPER (see page 481) but it can be any environment symbol or variable name whose value is a valid class.

Suppose you create an Account class that is a subclass of the Object class, define a TYPE method for the Account class, and create the Savings class that is a subclass of Account. You could define a TYPE method for the Savings class as follows:

```
savings~define('TYPE', 'return "a savings account"')
```

You could change the search order by using the following line:

```
savings~define('TYPE', 'return self~type:super "(savings)"')
```

This changes the search order so that the language processor searches for the TYPE method first in the Account superclass (rather than in the Savings subclass). When you create an instance of the Savings class (`asav`) and send a TYPE message to `asav`:

```
say asav~type
```

`an account (savings)` is displayed. The TYPE method of the Savings class calls the TYPE method of the Account class, and adds the string `(savings)` to the results.

Public and Private Methods

A method can be public or private. Any object can send a message that runs a *public* method. A *private* method runs only when an object sends a message to itself (that is, using the variable SELF as the message receiver). Private methods include methods at different scopes within the same object. (Superclasses can make private methods available to their subclasses while hiding those methods from other objects.) A private method is like an internal subroutine. It provides common functions to the object methods but is hidden from other programs.

The Class Hierarchy

REXX provides the following classes belonging to the object class:
- Alarm class
- Class class
- Array class
- List class
- Queue class
- Table class
  - Set class
- Directory class
- Relation class

  - Bag class
- Message class
- Method class
- Monitor class
- Stem class
- Stream class
- String class
- Supplier class

(The classes are in a class hierarchy with subclasses indented below their superclasses.)

Note that there might also be other classes in the system.

Initialization

Any object requiring initialization at creation time must define an INIT method. If this method is defined, the class object runs the INIT method after the object is created. If an object has more than one INIT method (for example, it is defined in several classes), each INIT method must forward the INIT message up the hierarchy to complete the object's initialization.

Example:

```
asav = .savings~new(1000.00, 6.25)
say asav~type
asav~name = 'John Smith'

::class Account

::method INIT
expose balance
use arg balance

::method TYPE
return "an account"

::method name attribute

::class Savings subclass Account

::method INIT
expose interest_rate
use arg balance, interest_rate
self~init:super(balance)

::method type
return "a savings account"
```

The NEW method of the Savings class object creates a new Savings object and calls the INIT method of the new object. The INIT method arguments are the arguments specified on the NEW method. In the Savings INIT method, the line:

```
self~init:super(balance)
```

calls the INIT method of the Account class, using just the balance argument specified on the NEW message.

Object Destruction and Uninitialization

Object destruction is implicit. When an object is no longer in use, REXX automatically reclaims its storage. If the object has allocated other system resources, you must release them at this time. (REXX cannot release these resources, because it is unaware that the object has allocated them.)

Similarly, other uninitialization processing may be needed, for example, by a message object holding an unreported error. An object requiring uninitialization should define an UNINIT method. If this method is defined, REXX runs it before reclaiming the object's storage. If an object has more than one UNINIT method (defined in several classes), each UNINIT method is responsible for sending the UNINIT method up the object hierarchy.

Required String Values

REXX requires a string value in a number of contexts within instructions and built-in function calls.

- DO statements containing *exprr* or *exprf*
- Substituted values in compound variable names
- Commands to external environments
- Commands and environment names on ADDRESS instructions
- Strings for ARG, PARSE, and PULL instructions to be parsed
- Parenthesized targets on CALL instructions
- Subsidiary variable lists on DROP, EXPOSE, and PROCEDURE instructions
- Instruction strings on INTERPRET instructions
- DIGITS, FORM, and FUZZ values on NUMERIC instructions
- Options strings on OPTIONS instructions
- Data queue strings on PUSH and QUEUE instructions
- Label names on SIGNAL VALUE instructions
- Trace settings on TRACE VALUE instructions
- Arguments to built-in functions
- Variable references in parsing templates
- Data for PUSH and QUEUE instructions to be processed
- Data for the SAY instruction to be displayed
- REXX dyadic operators when the receiving object (the object to the left of the operator) is a string

If you supply an object other than a string in these contexts, by default the language processor converts it to some string representation and uses this. However, the programmer can cause the language processor to raise the NOSTRING condition when the supplied object does not have an equivalent string value.

To obtain a string value, the language processor sends a REQUEST('STRING') message to the object. Strings and other objects that have string values return the appropriate string value for REXX to use. (This happens automatically for strings and for subclasses of the String class because they inherit a suitable MAKESTRING method from the String class.) For this mechanism to work correctly, you must provide a MAKESTRING method for any other objects with string values.

For other objects without string values (that is, without a MAKESTRING method), the action taken depends on the setting of the NOSTRING condition trap. If the NOSTRING condition is being trapped (see “Chapter 12. Conditions and Condition Traps” on page 429), the language processor raises the NOSTRING condition. If the NOSTRING condition is not being trapped, the language processor sends a STRING message to the object to obtain its readable string representation (see the STRING method of the Object class “STRING” on page 185) and uses this string.

Example:

```
d = .directory~new
say substr(d,5,7)        /* Produces "rectory" from "a Directory" */
signal on nostring
say substr(d,5,7)        /* Raises the NOSTRING condition */
say substr(d~string,3,6)  /* Displays "Direct" */
```

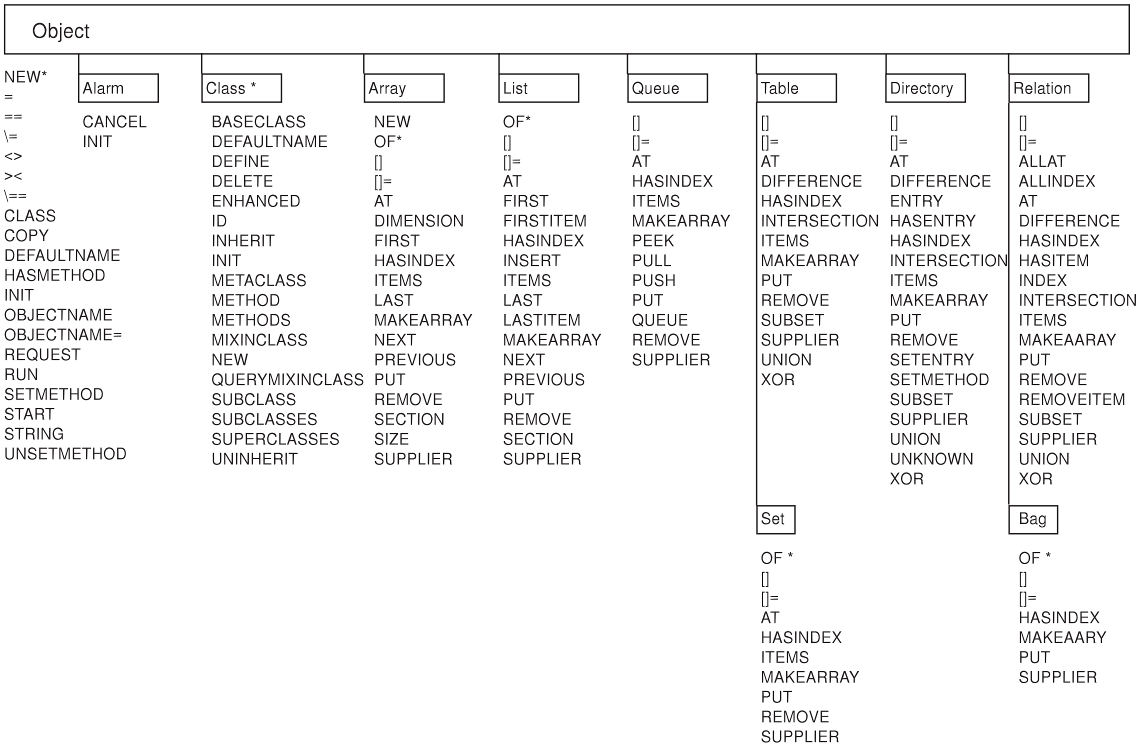
For arguments to REXX object methods, different rules apply. When a method expects a string as an argument, the argument object is sent the REQUEST(’STRING’) message. If REQUEST returns the NIL object, then the method raises an error.

Concurrency

REXX supports *concurrency*, multiple methods running simultaneously on a single object. See “Chapter 13. Concurrency” on page 439 for a full description of concurrency.

Classes and Methods Provided by REXX

The following figure shows all the classes and their methods.



Object	Alarm	Class *	Array	List	Queue	Table	Directory	Relation
NEW*	CANCEL	BASECLASS	NEW	OF*	[]	[]	[]	[]
=	INIT	DEFAULTNAME	OF*	[]	[]=	[]=	[]=	[]=
==		DEFINE	[]	[]=	AT	AT	AT	ALLAT
\=		DELETE	[]=	AT	HASINDEX	DIFFERENCE	DIFFERENCE	ALLINDEX
<>		ENHANCED	AT	FIRST	ITEMS	HASINDEX	ENTRY	AT
><		ID	DIMENSION	FIRSTITEM	MAKEARRAY	INTERSECTION	HASENTRY	DIFFERENCE
\==		INHERIT	FIRST	HASINDEX	PEEK	ITEMS	HASINDEX	HASINDEX
CLASS		INIT	HASINDEX	INSERT	PULL	MAKEARRAY	INTERSECTION	HASITEM
COPY		METACLASS	ITEMS	ITEMS	PUSH	PUT	ITEMS	INDEX
DEFAULTNAME		METHOD	LAST	LAST	PUT	REMOVE	MAKEARRAY	INTERSECTION
HASMETHOD		METHODS	MAKEARRAY	LASTITEM	QUEUE	SUBSET	PUT	ITEMS
INIT		MIXINCLASS	NEXT	MAKEARRAY	REMOVE	SUPPLIER	REMOVE	MAKEAARAY
OBJECTNAME		NEW	PREVIOUS	NEXT	SUPPLIER	UNION	SETENTRY	PUT
OBJECTNAME=		QUERYMIXINCLASS	PUT	PREVIOUS		XOR	SETMETHOD	REMOVE
REQUEST		SUBCLASS	REMOVE	PUT			SUBSET	REMOVEITEM
RUN		SUBCLASSES	SECTION	REMOVE			SUPPLIER	SUBSET
SETMETHOD		SUPERCLASSES	SIZE	SECTION			UNION	SUPPLIER
START		UNINHERIT	SUPPLIER	SUPPLIER			UNKNOWN	UNION
STRING							XOR	XOR
UNSETMETHOD								

Set	Bag
OF *	OF *
[]	[]
[]=	[]=
AT	HASINDEX
HASINDEX	MAKEAARY
ITEMS	PUT
MAKEARRAY	SUPPLIER
PUT	
REMOVE	
SUPPLIER	

* All of the methods under the Class class are both class and instance methods.
NEW and OF are class methods.

Figure 11. Classes and Inheritance of Methods (Part 1 of 2)

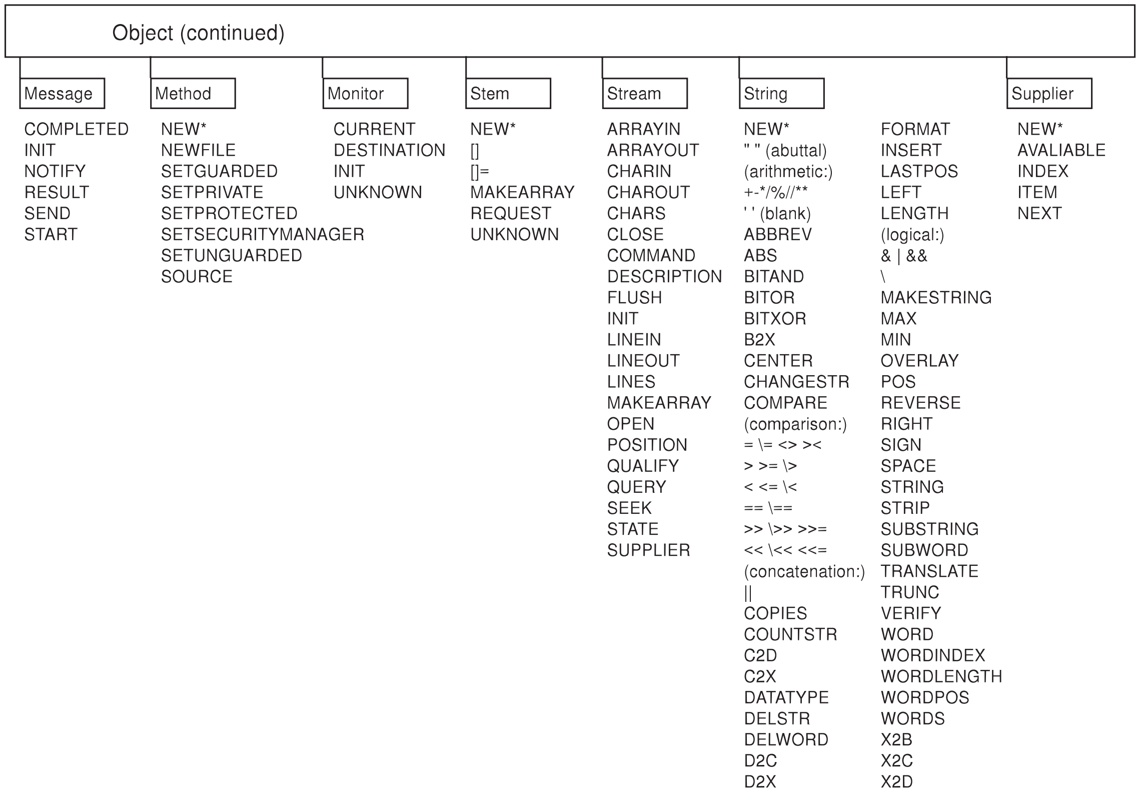


Figure 11. Classes and Inheritance of Methods (Part 2 of 2)

Summary of Methods by Class

The following table lists all the methods and the classes that define them. All methods are instance methods except where noted.

Table 1. Summary of Methods and the Classes Defining Them

Method Name	Class(es) and Page(s)
[]	**Array**, “[]” on page 120 **Bag**, “[]” on page 125 **Directory**, “[]” on page 128 **List**, “[]” on page 135 **Queue**, “[]” on page 139 **Relation**, “[]” on page 143 **Set**, “[]” on page 148 **Stem**, “[]” on page 187 **Table**, “[]” on page 151

Table 1. Summary of Methods and the Classes Defining Them (continued)

Method Name	Class(es) and Page(s)
[]=	**Array** "[]=" on page 120 **Bag**, "[]=" on page 125 **Directory**, "[]=" on page 128 **List**, "[]=" on page 135 **Queue**, "[]=" on page 139 **Relation**, "[]=" on page 143 **Set**, "[]=" on page 148 **Stem**, "[]=" on page 187 **Table**, "[]=" on page 151
ABBREV	**String**, "ABBREV" on page 212
ABS	**String**, "ABS" on page 213
ALLAT	**Relation**, "ALLAT" on page 143
ALLINDEX	**Relation**, "ALLINDEX" on page 143
ARRAYIN	**Stream**, "ARRAYIN" on page 189
ARRAYOUT	**Stream**, "ARRAYOUT" on page 190
AT	**Array**, "AT" on page 120 **Directory**, "AT" on page 128 **List**, "AT" on page 135 **Queue**, "AT" on page 139 **Relation**, "AT" on page 143 **Set**, "AT" on page 149 **Table**, "AT" on page 151
AVAILABLE	**Supplier**, "AVAILABLE" on page 240
BASECLASS	**Class**, "BASECLASS" on page 162
BITAND	**String**, "BITAND" on page 213
BITOR	**String**, "BITOR" on page 214
BITXOR	**String**, "BITXOR" on page 215
B2X	**String**, "B2X" on page 215
CANCEL	**Alarm**, "CANCEL" on page 160
CENTER	**String**, "CENTER/CENTRE" on page 216
CHANGESTR	**String**, "CHANGESTR" on page 216
CHARIN	**Stream**, "CHARIN" on page 190
CHAROUT	**Stream**, "CHAROUT" on page 190
CHARS	**Stream**, "CHARS" on page 191
CLASS	**Object**, "CLASS" on page 180
CLOSE	**Stream**, "CLOSE" on page 191
COMMAND	**Stream**, "COMMAND" on page 191
COMPARE	**String**, "COMPARE" on page 217
COMPLETED	**Message**, "COMPLETED" on page 171
COPIES	**String**, "COPIES" on page 217
COPY	**Object**, "COPY" on page 180

Table 1. Summary of Methods and the Classes Defining Them (continued)

Method Name	Class(es) and Page(s)
COUNTSTR	**String**, “COUNTSTR” on page 218
CURRENT	**Monitor**, “CURRENT” on page 178
C2D	**String**, “C2D” on page 218
C2X	**String**, “C2X” on page 219
DATATYPE	**String**, “DATATYPE” on page 219
DEFAULTNAME	**Class**, “DEFAULTNAME” on page 162 **Object**, “DEFAULTNAME” on page 180
DEFINE	**Class** (class and instance method), “DEFINE” on page 162
DELETE	**Class** (class and instance method), “DELETE” on page 163
DELSTR	**String**, “DELSTR” on page 221
DELWORD	**String**, “DELWORD” on page 221
DESCRIPTION	**Stream**, “DESCRIPTION” on page 198
DESTINATION	**Monitor**, “DESTINATION” on page 178
DIFFERENCE	**Directory**, “DIFFERENCE” on page 131 **Relation**, “DIFFERENCE” on page 145 **Table**, “DIFFERENCE” on page 153
DIMENSION	**Array**, “DIMENSION” on page 120
D2C	**String**, “D2C” on page 222
D2X	**String**, “D2X” on page 223
ENHANCED	**Class** (class and instance method), “ENHANCED” on page 164
ENTRY	**Directory**, “ENTRY” on page 128
FIRST	**Array**, “FIRST” on page 121 **List**, “FIRST” on page 135
FIRSTITEM	**List**, “FIRSTITEM” on page 135
FLUSH	**Stream**, “FLUSH” on page 198
FORMAT	**String**, “FORMAT” on page 223
HASENTRY	**Directory**, “HASENTRY” on page 129

Table 1. Summary of Methods and the Classes Defining Them (continued)

Method Name	**Class(es) and Page(s)**
HASINDEX	**Array**, “HASINDEX” on page 121 **Bag**, “HASINDEX” on page 125 **Directory**, “HASINDEX” on page 129 **List**, “HASINDEX” on page 135 **Queue**, “HASINDEX” on page 140 **Relation**, “HASINDEX” on page 143 **Set**, “HASINDEX” on page 149 **Table**, “HASINDEX” on page 151
HASITEM	**Relation**, “HASITEM” on page 144
HASMETHOD	**Object**, “HASMETHOD” on page 181
ID	**Class** (class and instance method), “ID” on page 164
INDEX	**Relation**, “INDEX” on page 144 **Supplier**, “INDEX” on page 240
INHERIT	**Class** (class and instance method), “INHERIT” on page 164
INIT	**Alarm**, “INIT” on page 160 **Class**, “INIT” on page 165 **Message**, “INIT” on page 171 **Monitor**, “INIT” on page 178 **Object**, “INIT” on page 181 **Stream**, “INIT” on page 198
INSERT	**List**, “INSERT” on page 136 **String**, “INSERT” on page 225
INTERSECTION	**Directory**, “INTERSECTION” on page 131 **Relation**, “INTERSECTION” on page 145 **Table**, “INTERSECTION” on page 153
ITEM	**Supplier**, “ITEM” on page 240
ITEMS	**Array**, “ITEMS” on page 121 **Directory**, “ITEMS” on page 129 **List**, “ITEMS” on page 136 **Queue**, “ITEMS” on page 140 **Relation**, “ITEMS” on page 144 **Set**, “ITEMS” on page 149 **Table**, “ITEMS” on page 152
LAST	**Array**, “LAST” on page 121 **List**, “LAST” on page 136
LASTITEM	**List**, “LASTITEM” on page 137
LASTPOS	**String**, “LASTPOS” on page 225
LEFT	**String**, “LEFT” on page 226
LENGTH	**String**, “LENGTH” on page 226
LINEIN	**Stream**, “LINEIN” on page 198
LINEOUT	**Stream**, “LINEOUT” on page 199

Table 1. Summary of Methods and the Classes Defining Them (continued)

Method Name	Class(es) and Page(s)
LINES	**Stream**, “LINES” on page 199
MAKEARRAY	**Array**, “MAKEARRAY” on page 121 **Bag**, “MAKEARRAY” on page 126 **Directory**, “MAKEARRAY” on page 129 **List**, “MAKEARRAY” on page 137 **Queue**, “MAKEARRAY” on page 140 **Relation**, “MAKEARRAY” on page 144 **Set**, “MAKEARRAY” on page 149 **Stem**, “MAKEARRAY” on page 188 **Stream**, “MAKEARRAY” on page 200 **Table**, “MAKEARRAY” on page 152
MAKESTRING	**String**, “MAKESTRING” on page 227
MAX	**String**, “MAX” on page 227
METACLASS	**Class**, “METACLASS” on page 165
METHOD	**Class** (class and instance method), “METHOD” on page 166
METHODS	**Class** (class and instance method), “METHODS” on page 166
MIN	**String**, “MIN” on page 227
MIXINCLASS	**Class**, “MIXINCLASS” on page 167
NEW	**Array** (class method), “NEW (Class Method)” on page 119 **Class** (class and instance method), “NEW” on page 168 **Method**, “NEW (Class Method)” on page 175 **Object**, “NEW (Class Method)” on page 179 **Stem** (class method), “NEW (Class Method)” on page 187 **String** (class method), “NEW (Class Method)” on page 208 **Supplier** (class method), “NEW (Class Method)” on page 240
NEWFILE	**Method**, “NEWFILE (Class Method)” on page 176
NEXT	**Array**, “NEXT” on page 121 **List**, “NEXT” on page 137 **Supplier**, “NEXT” on page 241
NOTIFY	**Message**, “NOTIFY” on page 172
OBJECTNAME	**Object**, “OBJECTNAME” on page 181
OBJECTNAME=	**Object**, “OBJECTNAME=” on page 181

Table 1. Summary of Methods and the Classes Defining Them (continued)

Method Name	Class(es) and Page(s)
OF	**Array** (class method), “OF (Class Method)” on page 119 **Bag** (class method), “OF (Class Method)” on page 125 **List** (class method), “OF (Class Method)” on page 134 **Set** (class method), “OF (Class Method)” on page 148
OPEN	**Stream**, “OPEN” on page 200
Operator Methods (Arithmetic): +, –, *, /, %, //, **, prefix +, prefix −	**String**, “Arithmetic Methods” on page 208
Operator Methods (Comparison): =, \=, ><, <>, ==, and \==	**Object**, “Operator Methods” on page 179 **String**, 209
Operator Methods (Comparison): >, <, >=, \<, <=, \>, >>, <<, >>=, \<<, <<=, and \>>	**String**, page 211
Operator Methods (Concatenation): "" (abuttal), ||, and " " (blank)	**String**, page 212
Operator Methods (Logical): &, |, &&, and prefix \	**String**, page 211
Operator Methods (Other): == (unary)	**Object**, “Operator Methods” on page 179
OVERLAY	**String**, “OVERLAY” on page 228
PEEK	**Queue**, “PEEK” on page 140
POS	**String**, “POS” on page 228
POSITION	**Stream**, “POSITION” on page 202
PREVIOUS	**Array**, “PREVIOUS” on page 122 **List**, “PREVIOUS” on page 137
PULL	**Queue**, “PULL” on page 140
PUSH	**Queue**, “PUSH” on page 140
PUT	**Array**, “PUT” on page 122 **Bag**, “PUT” on page 126 **Directory**, “PUT” on page 129 **List**, “PUT” on page 137 **Queue**, “PUT” on page 141 **Relation**, “PUT” on page 144 **Set**, “PUT” on page 149 **Table**, “PUT” on page 152
QUALIFY	**Stream**, “QUALIFY” on page 202
QUERY	**Stream**, “QUERY” on page 202
QUERYMIXINCLASS	**Class**, “QUERYMIXINCLASS” on page 168
QUEUE	**Queue**, “QUEUE” on page 141

Table 1. Summary of Methods and the Classes Defining Them (continued)

Method Name	Class(es) and Page(s)
REMOVE	**Array**, “REMOVE” on page 122 **Directory**, “REMOVE” on page 129 **List**, “REMOVE” on page 137 **Queue**, “REMOVE” on page 141 **Relation**, “REMOVE” on page 145 **Set**, “REMOVE” on page 149 **Table**, “REMOVE” on page 152
REMOVEITEM	**Relation**, “REMOVEITEM” on page 145
REQUEST	**Object**, “REQUEST” on page 182 **Stem**, “REQUEST” on page 188
RESULT	**Message**, “RESULT” on page 173
REVERSE	**String**, “REVERSE” on page 229
RIGHT	**String**, “RIGHT” on page 229
RUN	**Object**, “RUN” on page 183
SECTION	**Array**, “SECTION” on page 122 **List**, “SECTION” on page 138
SEEK	**Stream**, “SEEK” on page 204
SEND	**Message**, “SEND” on page 173
SETENTRY	**Directory**, “SETENTRY” on page 130
SETGUARDED	**Method**, “SETGUARDED” on page 176
SETMETHOD	**Directory**, “SETMETHOD” on page 130 **Object**, “SETMETHOD” on page 183
SETPRIVATE	**Method**, “SETPRIVATE” on page 176
SETPROTECTED	**Method**, “SETPROTECTED” on page 176
SETSECURITYMANAGER	**Method**, “SETSECURITYMANAGER” on page 176
SETUNGUARDED	**Method**, “SETUNGUARDED” on page 177
SIGN	**String**, “SIGN” on page 229
SIZE	**Array**, “SIZE” on page 123
SOURCE	**Method**, “SOURCE” on page 177
SPACE	**String**, “SPACE” on page 230
START	**Message**, “START” on page 173 **Object**, “START” on page 184
STATE	**Stream**, “STATE” on page 205
STRING	**Object**, “STRING” on page 185 **String**, “STRING” on page 230
STRIP	**String**, “STRIP” on page 231

Table 1. Summary of Methods and the Classes Defining Them (continued)

Method Name	Class(es) and Page(s)
SUBCLASS	**Class** (class and instance method), "SUBCLASS" on page 168
SUBCLASSES	**Class** (class and instance method), "SUBCLASSES" on page 169
SUBSET	**Directory**, "SUBSET" on page 132 **Relation**, "SUBSET" on page 146 **Table**, "SUBSET" on page 153
SUBSTR	**String**, "SUBSTR" on page 231
SUBWORD	**String**, "SUBWORD" on page 232
SUPERCLASSES	**Class** (class and instance method), "SUPERCLASSES" on page 169
SUPPLIER	**Array**, "SUPPLIER" on page 123 **Bag**, "SUPPLIER" on page 126 **Directory**, "SUPPLIER" on page 130 **List**, "SUPPLIER" on page 138 **Queue**, "SUPPLIER" on page 141 **Relation**, "SUPPLIER" on page 145 **Set**, "SUPPLIER" on page 150 **Stream**, "SUPPLIER" on page 206 **Table**, "SUPPLIER" on page 152
TRANSLATE	**String**, "TRANSLATE" on page 232
TRUNC	**String**, "TRUNC" on page 233
UNINHERIT	**Class** (class and instance method), "UNINHERIT" on page 170
UNION	**Directory**, "UNION" on page 132 **Relation**, "UNION" on page 146 **Table**, "UNION" on page 153
UNKNOWN	**Directory**, "UNKNOWN" on page 131 **Monitor**, "UNKNOWN" on page 178 **Stem**, "UNKNOWN" on page 188
UNSETMETHOD	**Object**, "UNSETMETHOD" on page 185
VERIFY	**String**, "VERIFY" on page 234
WORD	**String**, "WORD" on page 235
WORDINDEX	**String**, "WORDINDEX" on page 235
WORDLENGTH	**String**, "WORDLENGTH" on page 235
WORDPOS	**String**, "WORDPOS" on page 236
WORDS	**String**, "WORDS" on page 236
XOR	**Directory**, "XOR" on page 132 **Relation**, "XOR" on page 146 **Table**, "XOR" on page 154

Table 1. Summary of Methods and the Classes Defining Them (continued)

Method Name	Class(es) and Page(s)
X2B	**String**, “X2B” on page 237
X2C	**String**, “X2C” on page 237
X2D	**String**, “X2D” on page 238

The chapters that follow describe the classes and other objects that REXX provides and their available methods. REXX provides the objects listed in these sections and they are generally available to all methods through environment symbols (see “Environment Symbols” on page 35).

Notes:

1. In the method descriptions in the chapters that follow, methods that return a result begin with the word ‘returns’.
2. For [] and []= methods, the syntax diagrams include the index or indexes within the brackets. These diagrams are intended to show how you can use these methods. For example, to retrieve the first element of a one-dimensional array named Array1, you would typically use the syntax:

   ```
   Array1[1]
   ```

   rather than:

   ```
   Array1~"[]"(1)
   ```

   even though the latter is valid and equivalent. For more information, see “Message Terms” on page 25 and “Message Instructions” on page 28.
3. When the argument of a method must be a specific kind of object (such as array, class, method, or string) the variable you specify must be of the same class as the required object or be able to produce an object of the required kind in response to a conversion message. In particular, subclasses are acceptable in place of superclasses (unless overridden in a way that changes superclass behavior), because they inherit a suitable conversion method from their REXX superclass.

   The REQUEST method of the Object class (see “REQUEST” on page 182) can perform this validation.

Chapter 5. The Collection Classes

A *collection* is an object that contains a number of *items*, which can be any objects. Every item stored in a REXX collection has an associated index that you can use to retrieve the item from the collection with the AT or [] methods.

Each collection defines its own acceptable index types. REXX provides the following collection classes:

Collections that do not have set operations:

Array	A sequenced collection of objects ordered by whole-number indexes. See "The Array Class" on page 118 for details.
List	A sequenced collection that lets you add new items at any position in the sequence. A list generates and returns an index value for each item placed in the list. The returned index remains valid until the item is removed from the list. See "The List Class" on page 133 for details.
Queue	A sequenced collection with the items ordered as a queue. You can remove items from the head of the queue and add items at either its tail or its head. Queues index the items with whole-number indexes, in the order in which the items would be removed. The current head of the queue has index 1, the item after the head item has index 2, up to the number of items in the queue. See "The Queue Class" on page 138 for details.

Collections that have set operations:

Table	A collection with indexes that can be any object. For example, string objects, array objects, alarm objects, or any user-created object can be a table index. The table class determines the index match by using the == comparison method. A table contains no duplicate indexes. See "The Table Class" on page 150 for details.
Directory	A collection with character string indexes. Index comparisons are performed using the string == comparison method. See "The Directory Class" on page 127 for details.
Relation	A collection with indexes that can be any object (as with the table class). A relation can contain duplicate indexes. See "The Relation Class" on page 141 for details.

Set	A collection where the index and the item are the same object. Set indexes can be any object (as with the table class) and each index is unique. See "The Set Class" on page 147 for details.
Bag	A collection where the index and the item are the same object. Bag indexes can be any object (as with the table class) and each index can appear more than once. See "The Bag Class" on page 124 for details.

The following sections describe the individual collection classes in alphabetical order and the methods that they define and inherit. It also describes the concept of set operations.

The Array Class

An *array* is a collection with indexes that are positive whole numbers. You can reference array items by using one or more indexes. The number of indexes is the same as the number of dimensions of the array. This number is called the *dimensionality* of the array.

Array objects are variable-sized. The dimensionality of an array is fixed, but the size of each dimension is variable. When you create an array, you can specify a hint about how many elements you expect to put into the array or the array's dimensionality. However, you do not need to specify a size or dimensionality of an array when you are creating it. You can use any whole-number indexes to reference items in an array.

Methods the Array class defines:

NEW (Class method. Overrides Object class method.)
OF (Class method)
[]
[]=
AT
DIMENSION
FIRST
HASINDEX
ITEMS
LAST
MAKEARRAY
NEXT
PREVIOUS
PUT
REMOVE

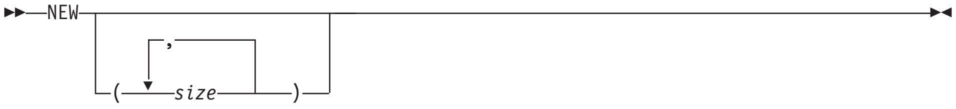
SECTION
SIZE
SUPPLIER

Methods inherited from the Object class:

Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Array class also has available class methods that its metaclass, the Class class, defines.

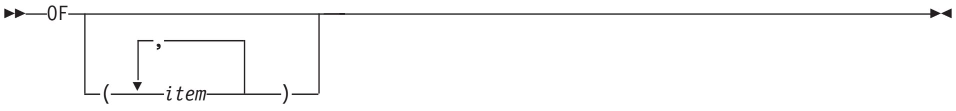
NEW (Class Method)



Returns a new empty array. If you specify any *size*, the size is taken as a hint about how big each dimension should be. The language processor uses this only to allocate the array object initially. For multiple dimension arrays, you can also specify how much space is to be allocated initially for each dimension of the array.

Each *size* argument must be `0` or a positive whole number. If it is `0`, the corresponding dimension is initially empty.

OF (Class Method)



Returns a newly created single-index array containing the specified *item* objects. The first *item* has index 1, the second has index 2, and so on.

If you use the OF method and omit any argument items, the returned array does not include the indexes corresponding to those you omitted.

[]

```
      ┌─,──────┐
      ▼        │
►►──[───index──┴──]──────────────────────────────────────────►◄
```

Returns the same value as the AT method, which follows. See "AT".

[]=

```
      ┌─,──────┐
      ▼        │
►►──[───index──┴──]=value────────────────────────────────────►◄
```

This method is the same as the PUT method, which follows. See "PUT" on page 122.

AT

```
        ┌─,──────┐
        ▼        │
►►──AT(───index──┴──)────────────────────────────────────────►◄
```

Returns the item associated with the specified *index* or *index*es. If the array has no item associated with the specified *index* or *index*es, this method returns the NIL object.

DIMENSION

```
►►──DIMENSION──┬───────┬─────────────────────────────────────►◄
               └─(n)───┘
```

Returns the current size (upper bound) of dimension *n* (a positive whole number). If you omit *n*, this method returns the dimensionality (number of dimensions) of the array. If the number of dimensions has not been determined, DIMENSION returns `0`.

FIRST

```
>>──FIRST──────────────────────────────────────────────────────────><
```

Returns the index of the first item in the array or the NIL object if the array is empty. The FIRST method is valid only for single-index arrays.

HASINDEX

```
              ┌─,──────┐
              ▼        │
>>──HASINDEX(───index──┴──)────────────────────────────────────────><
```

Returns 1 (true) if the array contains an item associated with the specified index or indexes. Returns 0 (false) otherwise.

ITEMS

```
>>──ITEMS──────────────────────────────────────────────────────────><
```

Returns the number of items in the collection.

LAST

```
>>──LAST───────────────────────────────────────────────────────────><
```

Returns the index of the last item in the array or the NIL object if the array is empty. The LAST method is valid only for single-index arrays.

MAKEARRAY

```
>>──MAKEARRAY──────────────────────────────────────────────────────><
```

Returns a single-index array with the same number of items as the receiver object. Any index with no associated item is omitted from the new array.

NEXT

```
>>──NEXT(index)────────────────────────────────────────────────────><
```

Returns the index of the item that follows the array item having index *index* or returns the NIL object if the item having that index is last in the array. The NEXT method is valid only for single-index arrays.

PREVIOUS

```
►►──PREVIOUS(index)──────────────────────────────────────────────────►◄
```

Returns the index of the item that precedes the array item having index *index* or the NIL object if the item having that index is first in the array. The PREVIOUS method is valid only for single-index arrays.

PUT

```
                ┌─────────┐
                ▼         │
►►──PUT(item────,index────┴──)───────────────────────────────────────►◄
```

Makes the object *item* a member item of the array and associates it with the specified *index* or *index*es. This replaces any existing item associated with the specified *index* or *index*es with the new item. If the *index* for a particular dimension is greater than the current size of that dimension, the array is expanded to the new dimension size.

REMOVE

```
            ┌─,──────┐
            ▼        │
►►──REMOVE(───index──┴──)────────────────────────────────────────────►◄
```

Returns and removes the member item with the specified *index* or *index*es from the array. If there is no item with the specified *index* or *index*es, the NIL object is returned and no item is removed.

SECTION

```
►►──SECTION(start──────────────)─────────────────────────────────────►◄
                  └─,items─┘
```

Returns a new array (of the same class as the receiver) containing selected items from the receiver array. The first item in the new array is the item corresponding to index *start* in the receiver array. Subsequent items in the new array correspond to those in the receiver array (in the same sequence). If you specify the whole number *items*, the new array contains only this number of items (or the number of subsequent items in the receiver array, if this is less than *items*). If you do not specify *items*, the new array contains all subsequent items of the receiver array. The receiver array remains unchanged. The SECTION method is valid only for single-index arrays.

SIZE

```
►►──SIZE───────────────────────────────────────────────────────────────►◄
```

Returns the number of items that can be placed in the array before it needs to be extended. This value is the same as the product of the sizes of the dimensions in the array.

SUPPLIER

```
►►──SUPPLIER───────────────────────────────────────────────────────────►◄
```

Returns a supplier object for the collection. After you have obtained a supplier, you can send it messages (see “The Supplier Class” on page 239) to enumerate all the items that were in the array at the time of the supplier’s creation. The supplier enumerates the array items in their sequenced order.

Examples

```
array1=.array~of(1,2,3,4)  /* Loads the array */

/* Alternative way to create and load an array */
array2=.array~new(4)  /* Creates array2, containing 4 items */
do i=1 to 4                 /* Loads the array */
  array2[i]=i
end
```

You can produce the elements loaded into an array, for example:

```
do i=1 to 4
  say array1[i]
end
```

If you omit any argument values before arguments you supply, the corresponding indexes are skipped in the returned array:

```
directions=.array~of('North','South',,'West')
do i=1 to 4                                  /* Produces: North          */
  say directions[i]                          /*           South          */
                                             /*           The NIL object */
end                                          /*           West           */
```

Here is an example using the ~~:

```
z=.array~of(1,2,3)~~put(4,4)
do i = 1 to z~size
  say z[i]              /* Produces:  1 2 3 4 */
end
```

The Bag Class

A *bag* is a collection that restricts the elements to having an item that is the same as the index. Any object can be placed in a bag, and the same object can be placed in a bag several times.

The Bag class is a subclass of the Relation class. In addition to its own methods, it inherits the methods of the Object class and the Relation class.

Methods the Bag class defines:

OF (Class method)
[]
[]= (Overrides Relation class method)
HASINDEX
MAKEARRAY
PUT (Overrides Relation class method)
SUPPLIER

Methods inherited from the Relation class:

ALLAT
ALLINDEX
AT
HASITEM
INDEX
ITEMS
REMOVE
REMOVEITEM

Set-operator methods inherited from the Relation class:

DIFFERENCE
INTERSECTION
SUBSET
UNION
XOR

Methods inherited from the Object class:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==

CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Bag class also has available class methods that its metaclass, the Class class, defines.

OF (Class Method)

```
            ┌─,───────┐
            ▼         │
►►──OF(─────item──────┴──)────────────────────────────────────────►◄
```

Returns a newly created bag containing the specified *item* objects.

[]

```
►►──[index]───────────────────────────────────────────────────────►◄
```

Returns the same value as the AT method in the Relation class. See “AT” on page 143.

[]=

```
►►──[index]=item──────────────────────────────────────────────────►◄
```

This method is the same as the PUT method. See “PUT” on page 126.

HASINDEX

```
►►──HASINDEX(index)───────────────────────────────────────────────►◄
```

Returns 1 (true) if the collection contains any item associated with index *index*, or 0 (false).

MAKEARRAY

```
►►──MAKEARRAY──────────────────────────────────────────────────────────►◄
```

Returns a single-index array containing the index objects. The array indexes range from 1 to the number of items. The collection items appear in the array in an unspecified order. (The program should not rely on any order.)

PUT

```
►►──PUT(item──┬────────┬──)────────────────────────────────────────────►◄
              └─,index─┘
```

Makes the object *item* a member item of the collection and associates it with index *index*. If you specify *index*, it must be the same as *item*.

SUPPLIER

```
►►──SUPPLIER───────────────────────────────────────────────────────────►◄
```

Returns a supplier object for the collection. After you have obtained a supplier, you can send it messages (see "The Supplier Class" on page 239) to enumerate all the items that were in the collection at the time of the supplier's creation. The supplier enumerates the items in an unspecified order. (The program should not rely on any order.)

Examples

```
/* Create a bag of fruit */
fruit = .bag~of('Apple', 'Orange', 'Apple', 'Pear')
say fruit~items         /* How many pieces? (4)      */
say fruit~items('Apple') /* How many apples? (2)      */
fruit~remove('Apple')    /* Remove one of the apples. */
fruit~~put('Banana')~put('Orange') /* Add a couple.   */
say fruit~items         /* How many pieces? (5)      */
```

The Directory Class

A *directory* is a collection with unique indexes that are character strings representing names.

Directories let you refer to objects by name, for example:

```
.environment~array
```

For directories, items are often referred to as *entries*.

Methods the Directory class defines:

[]
[]=
AT
ENTRY
HASENTRY
HASINDEX
ITEMS
MAKEARRAY
PUT
REMOVE
SETENTRY
SETMETHOD (Overrides Object class method)
SUPPLIER
UNKNOWN

Set-operator methods the Directory class defines:

DIFFERENCE
INTERSECTION
SUBSET
UNION
XOR

Methods Inherited from the Object Class:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=

REQUEST
RUN
START
STRING
UNSETMETHOD

Note: The Directory class also has available class methods that its metaclass, the Class class, defines.

[]

```
►►──[name]──────────────────────────────────────────────────────►◄
```

Returns the same item as the AT method, which follows. See “AT”.

[]=

```
►►──[name]=item──────────────────────────────────────────────────►◄
```

This method is the same as the PUT method. See “PUT” on page 129.

AT

```
►►──AT(name)─────────────────────────────────────────────────────►◄
```

Returns the item associated with index *name*. If a method that SETMETHOD supplies is associated with index *name*, the result of running this method is returned. If the collection has no item or method associated with index *name*, this method returns the NIL object.

Example:

```
say .environment~AT('OBJECT')   /* Produces: 'The Object class' */
```

ENTRY

```
►►──ENTRY(name)──────────────────────────────────────────────────►◄
```

Returns the directory entry with name *name* (translated to uppercase). If there is no such entry, *name* returns the item for any method that SETMETHOD supplied. If there is neither an entry nor a method for *name* or for UNKNOWN, the language processor raises an error.

HASENTRY

```
►►──HASENTRY(name)─────────────────────────────────────────────────►◄
```

Returns 1 (true) if the directory has an entry or a method for name *name* (translated to uppercase), or 0 (false).

HASINDEX

```
►►──HASINDEX(name)─────────────────────────────────────────────────►◄
```

Returns 1 (true) if the collection contains any item associated with index *name*, or 0 (false).

ITEMS

```
►►──ITEMS──────────────────────────────────────────────────────────►◄
```

Returns the number of items in the collection.

MAKEARRAY

```
►►──MAKEARRAY──────────────────────────────────────────────────────►◄
```

Returns a single-index array containing the index objects. The array indexes range from 1 to the number of items. The collection items appear in the array in an unspecified order. (The program should not rely on any order.)

PUT

```
►►──PUT(item,name)─────────────────────────────────────────────────►◄
```

Makes the object *item* a member item of the collection and associates it with index *name*. The new item replaces any existing item or method associated with index *name*.

REMOVE

```
►►──REMOVE(name)───────────────────────────────────────────────────►◄
```

Returns and removes the member item with index *name* from a collection. If a method is associated with SETMETHOD for index *name*, REMOVE removes the method and returns the result of running it. If there is no item or method with index *name*, the UNKNOWN method returns the NIL object and removes nothing.

SETENTRY

```
▸▸──SETENTRY(name──┬────────┬──)──────────────────────────────────────▸◂
                   └─,entry─┘
```

Sets the directory entry with name *name* (translated to uppercase) to the object *entry*, replacing any existing entry or method for *name*. If you omit *entry*, this method removes any entry or method with this *name*.

SETMETHOD

```
▸▸──SETMETHOD(name──┬─────────┬──)────────────────────────────────────▸◂
                    └─,method─┘
```

Associates entry name *name* (translated to uppercase) with method *method*. Thus, the language processor returns the result of running *method* when you access this entry. This occurs when you specify *name* on the AT, ENTRY, or REMOVE method. This method replaces any existing item or method for *name*.

You can specify the name UNKNOWN as *name*. Doing so supplies a method to run whenever an AT or ENTRY message specifies a name for which no item or method exists in the collection. This method's first argument is the specified directory index. This method has no effect on the action of any HASENTRY, HASINDEX, ITEMS, REMOVE, or SUPPLIER message sent to the collection.

The *method* can be a string containing a method source line instead of a method object. Alternatively, an array of strings containing individual method lines can be passed. In either case, SETMETHOD creates an equivalent method object.

If you omit *method*, SETMETHOD removes the entry with the specified *name*.

SUPPLIER

```
▸▸──SUPPLIER──────────────────────────────────────────────────────────▸◂
```

Returns a supplier object for the collection. After you have obtained a supplier, you can send it messages (see "The Supplier Class" on page 239) to enumerate all the items that were in the collection at the time of the supplier's creation. The supplier enumerates the items in an unspecified order. (The program should not rely on any order.)

UNKNOWN

```
▶▶──UNKNOWN(messagename,messageargs)──────────────────────────────▶◀
```

Runs either the ENTRY or SETENTRY method, depending on whether *messagename* ends with an equal sign. If *messagename* does not end with an equal sign, this method runs the ENTRY method, passing *messagename* as its argument. The language processor ignores any arguments specified in the array *messageargs*. In this case, UNKNOWN returns the result of the ENTRY method.

If *messagename* does end with an equal sign, this method runs the SETENTRY method, passing the first part of *messagename* (up to, but not including, the final equal sign) as its first argument, and the first item in the array *messageargs* as its second argument. In this case, UNKNOWN returns no result.

DIFFERENCE

```
▶▶──DIFFERENCE(argument)──────────────────────────────────────────▶◀
```

Returns a new collection (of the same class as the receiver) containing only those items from the receiver whose indexes the *argument* collection does *not* contain. The *argument* can be any object described in "The Argument Collection Classes" on page 157. The *argument* must also allow all of the index values in the receiver collection.

INTERSECTION

```
▶▶──INTERSECTION(argument)────────────────────────────────────────▶◀
```

Returns a new collection (of the same class as the receiver) containing only those items from the receiver whose indexes are in *both* the receiver collection and the *argument* collection. The *argument* can be any object described in "The Argument Collection Classes" on page 157. The *argument* must also allow all of the index values in the receiver collection.

SUBSET

```
►►──SUBSET(argument)─────────────────────────────────────────────────►◄
```

Returns 1 (true) if all indexes in the receiver collection are also contained in the *argument* collection; returns 0 (false) otherwise. The *argument* can be any object described in “The Argument Collection Classes” on page 157. The *argument* must also allow all of the index values in the receiver collection.

UNION

```
►►──UNION(argument)──────────────────────────────────────────────────►◄
```

Returns a new collection of the same class as the receiver that contains all the items from the receiver collection and selected items from the *argument* collection. This method includes an item from *argument* in the new collection only if there is no item with the same associated index in the receiver collection and the method has not already included an item with the same index. The order in which this method selects items in *argument* is unspecified. (The program should not rely on any order.) See also the UNION method of the Table (“UNION” on page 153) and Relation (“UNION” on page 146) classes. The *argument* can be any object described in “The Argument Collection Classes” on page 157. The *argument* must also allow all of the index values in the receiver collection.

XOR

```
►►──XOR(argument)────────────────────────────────────────────────────►◄
```

Returns a new collection of the same class as the receiver that contains all items from the receiver collection and the *argument* collection; all indexes that appear in both collections are removed. The *argument* can be any object described in “The Argument Collection Classes” on page 157. The *argument* must also allow all of the index values in the receiver collection.

Examples

```
/******************************************************************************/
/*  A Phone Book Directory program                                            */
/*  This program demonstrates use of the directory class.                     */
/******************************************************************************/

/* Define an UNKNOWN method that adds an abbreviation lookup feature.         */
/* Directories do not have to have an UNKNOWN method.                         */
book = .directory~new~~setmethod('UNKNOWN', .methods['UNKNOWN'])
```

```
book['ANN' ] = 'Ann B. ....... 555-6220'
book['ann' ] = 'Little annie . 555-1234'
book['JEFF'] = 'Jeff G. ...... 555-5115'
book['MARK'] = 'Mark C. ...... 555-5017'
book['MIKE'] = 'Mike H. ...... 555-6123'
book~Rick    = 'Rick M. ...... 555-5110'  /* Same as book['RICK'] = ...       */

Do i over book               /* Iterate over the collection                */
  Say book[i]
end i

Say ''                       /* Index lookup is case sensitive...          */
Say book~entry('Mike')       /* ENTRY method uppercases before lookup      */
Say book['ANN']              /* Exact match                                */
Say book~ann                 /* Message sends uppercase before lookup      */
Say book['ann']              /* Exact match with lowercase index           */

Say ''
Say book['M']                /* Uses UNKNOWN method for lookup             */
Say book['Z']
Exit

/* Define an unknown method to handle indexes not found.                   */
/* Check for abbreviations or indicate listing not found                   */
::Method UNKNOWN
  Parse arg at_index
  value = ''
  Do i over self
    If abbrev(i, at_index) then do
      If value <> '' then value = value', '
      value = value || self~at(i)
    end
  end i
  If value = '' then value = 'No listing found for' at_index
  Return value
```

The List Class

A *list* is a sequenced collection to which you can add new items at any position in the sequence. The collection supplies the list indexes at the time items are added with the INSERT method. The FIRST, LAST, and NEXT methods can also retrieve list indexes. Only indexes the list object generates are valid.

Methods the List class defines:

OF (Class method)
[]
[]=
AT
FIRST

FIRSTITEM
HASINDEX
INSERT
ITEMS
LAST
LASTITEM
MAKEARRAY
NEXT
PREVIOUS
PUT
REMOVE
SECTION
SUPPLIER

Methods inherited from the Object class:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The List class also has available class methods that its metaclass, the Class class, defines.

OF (Class Method)

```
         ┌─,──────┐
         ▼        │
►►──OF(───item────┴──)──────────────────────────────────────►◄
```

Returns a newly created list containing the specified *item* objects in the order specified.

[]

```
►►──[index]──────────────────────────────────────────────────────►◄
```

Returns the same item as the AT method. See “AT”.

[]=

```
►►──[index]=item─────────────────────────────────────────────────►◄
```

This method is the same as the PUT method. See “PUT” on page 137.

AT

```
►►──AT(index)────────────────────────────────────────────────────►◄
```

Returns the item associated with index *index*. If the collection has no item associated with *index*, this method returns the NIL object.

FIRST

```
►►──FIRST────────────────────────────────────────────────────────►◄
```

Returns the index of the first item in the list or the NIL object if the list is empty. The example for INSERT (see “INSERT” on page 136) includes FIRST.

FIRSTITEM

```
►►──FIRSTITEM────────────────────────────────────────────────────►◄
```

Returns the first item in the list or the NIL object if the list is empty.

Example:

```
musketeers=.list~of(Porthos,Athos,Aramis) /* Creates list MUSKETEERS      */
item=musketeers~firstitem                 /* Gives first item in list     */
                                          /* (Assigns "Porthos" to item)  */
```

HASINDEX

```
►►──HASINDEX(index)──────────────────────────────────────────────►◄
```

Returns 1 (true) if the collection contains any item associated with index *index*, or 0 (false).

INSERT

```
>>──INSERT(item──────────)──────────────────────────────────────────────><
                └─,index─┘
```

Returns a list-supplied index for a new item *item*, which is added to the list. The new item follows the existing item with index *index* in the list ordering. If *index* is the NIL object, the new item becomes the first item in the list. If you omit *index*, the new item becomes the last item in the list.

```
musketeers=.list~of(Porthos,Athos,Aramis) /* Creates list MUSKETEERS       */
                                          /* consisting of: Porthos        */
                                          /*                Athos          */
                                          /*                Aramis         */
index=musketeers~first                    /* Gives index of first item     */
musketeers~insert("D'Artagnan",index)     /* Adds D'Artagnan after Porthos  */
                                          /* List is now: Porthos          */
                                          /*              D'Artagnan       */
                                          /*              Athos            */
                                          /*              Aramis           */
/* Alternately, you could use */
musketeers~insert("D'Artagnan",.nil)      /* Adds D'Artagnan before Porthos */
                                          /* List is now:  D'Artagnan      */
                                          /*               Porthos         */
                                          /*               Athos           */
                                          /*               Aramis          */
/* Alternately, you could use */
musketeers~insert("D'Artagnan")           /* Adds D'Artagnan after Aramis   */
                                          /* List is now:  Porthos         */
                                          /*               Athos           */
                                          /*               Aramis          */
                                          /*               D'Artagnan      */
```

ITEMS

```
>>──ITEMS───────────────────────────────────────────────────────────────><
```

Returns the number of items in the collection.

LAST

```
>>──LAST────────────────────────────────────────────────────────────────><
```

Returns the index of the last item in the list or the NIL object if the list is empty.

LASTITEM

```
►►──LASTITEM──────────────────────────────────────────────────────►◄
```

Returns the last item in the list or the NIL object if the list is empty.

MAKEARRAY

```
►►──MAKEARRAY─────────────────────────────────────────────────────►◄
```

Returns a single-index array containing the receiver collection items. The array indexes range from 1 to the number of items. The order in which the collection items appear in the array is the same as their sequence in the list collection.

NEXT

```
►►──NEXT(index)───────────────────────────────────────────────────►◄
```

Returns the index of the item that follows the list item having index *index* or returns the NIL object if the item having that index is last in the list.

PREVIOUS

```
►►──PREVIOUS(index)───────────────────────────────────────────────►◄
```

Returns the index of the item that precedes the list item having index *index* or the NIL object if the item having that index is first in the list.

PUT

```
►►──PUT(item,index)───────────────────────────────────────────────►◄
```

Replaces any existing item associated with the specified *index* with the new item *item*. If the *index* does not exist in the list, an error is raised.

REMOVE

```
►►──REMOVE(index)─────────────────────────────────────────────────►◄
```

Returns and removes from a collection the member item with index *index*. If no item has index *index*, this method returns the NIL object and removes no item.

SECTION

```
>>──SECTION(start─┬────────┬─)──────────────────────────────────────────><
                  └─,items─┘
```

Returns a new list (of the same class as the receiver) containing selected items from the receiver list. The first item in the new list is the item corresponding to index *start* in the receiver list. Subsequent items in the new list correspond to those in the receiver list (in the same sequence). If you specify the whole number *items*, the new list contains only this number of items (or the number of subsequent items in the receiver list, if this is less than *items*). If you do not specify *items*, the new list contains all subsequent items from the receiver list. The receiver list remains unchanged.

SUPPLIER

```
>>──SUPPLIER────────────────────────────────────────────────────────────><
```

Returns a supplier object for the list. If you send appropriate messages to the supplier (see "The Supplier Class" on page 239), the supplier enumerates all the items in the list at the time of the supplier's creation. The supplier enumerates the items in their sequenced order.

The Queue Class

A *queue* is a sequenced collection with whole-number indexes. The indexes specify the position of an item relative to the head (first item) of the queue. Adding or removing an item changes the association of an index to its queue item. You can add items at either the tail or the head of the queue.

Methods the Queue class defines:

[]
[]=
AT
HASINDEX
ITEMS
MAKEARRAY
PEEK
PULL

PUSH
PUT
QUEUE
REMOVE
SUPPLIER

Methods inherited from the Object class:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Queue class also has available class methods that its metaclass, the Class class, defines.

[]

```
►►──[index]──────────────────────────────────────────────────────►◄
```

Returns the same value as the AT method. See "AT".

[]=

```
►►──[index]=item─────────────────────────────────────────────────►◄
```

This method is the same as the PUT method. See "PUT" on page 141.

AT

```
►►──AT(index)────────────────────────────────────────────────────►◄
```

Returns the item associated with index *index*. If the collection has no item associated with *index*, this method returns the NIL object.

HASINDEX

```
►►──HASINDEX(index)──────────────────────────────────────────────────►◄
```

Returns 1 (true) if the collection contains any item associated with index *index*, or 0 (false).

ITEMS

```
►►──ITEMS────────────────────────────────────────────────────────────►◄
```

Returns the number of items in the collection.

MAKEARRAY

```
►►──MAKEARRAY────────────────────────────────────────────────────────►◄
```

Returns a single-index array containing the receiver queue items. The array indexes range from 1 to the number of items. The order in which the queue items appear in the array is the same as their queuing order, with the head of the queue as index 1.

PEEK

```
►►──PEEK─────────────────────────────────────────────────────────────►◄
```

Returns the item at the head of the queue. The collection remains unchanged.

PULL

```
►►──PULL─────────────────────────────────────────────────────────────►◄
```

Returns and removes the item at the head of the queue.

PUSH

```
►►──PUSH(item)───────────────────────────────────────────────────────►◄
```

Adds the object *item* to the head of the queue.

PUT

```
►►──PUT(item,index)──────────────────────────────────────────────────►◄
```

Replaces any existing item associated with the specified *index* with the new item. If the *index* does not exist in the queue, an error is raised.

QUEUE

```
►►──QUEUE(item)──────────────────────────────────────────────────────►◄
```

Adds the object *item* to the tail of the queue.

REMOVE

```
►►──REMOVE(index)────────────────────────────────────────────────────►◄
```

Returns and removes from a collection the member item with index *index*. If no item has index *index*, this method returns the NIL object and removes no item.

SUPPLIER

```
►►──SUPPLIER─────────────────────────────────────────────────────────►◄
```

Returns a supplier object for the collection. After you have obtained a supplier, you can send it messages (see "The Supplier Class" on page 239) to enumerate all the items that were in the queue at the time of the supplier's creation. The supplier enumerates the items in their queuing order, with the head of the queue first.

The Relation Class

A *relation* is a collection with indexes that can be any objects the user supplies. In a relation, each item is associated with a single index, but there can be more than one item with the same index (unlike a table, which can contain only one item for any index).

Methods the Relation class defines:

[]
[]=
ALLAT
ALLINDEX
AT
HASINDEX
HASITEM
INDEX
ITEMS
MAKEARRAY
PUT
REMOVE
REMOVEITEM
SUPPLIER

Set-operator methods the Relation class defines:

DIFFERENCE
INTERSECTION
SUBSET
UNION
XOR

Methods inherited from the Object class:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Relation class also has available class methods that its metaclass, the Class class, defines.

[]

```
►►──[index]──────────────────────────────────────────────────────────►◄
```

Returns the same item as the AT method. See “AT”.

[]=

```
►►──[index]=item─────────────────────────────────────────────────────►◄
```

This method is the same as the PUT method. See “PUT” on page 144.

ALLAT

```
►►──ALLAT(index)─────────────────────────────────────────────────────►◄
```

Returns a single-index array containing all the items associated with index *index*. The indexes of the returned array range from 1 to the number of items. Items in the array appear in an unspecified order.

ALLINDEX

```
►►──ALLINDEX(item)───────────────────────────────────────────────────►◄
```

Returns a single-index array containing all indexes for item *item*, in an unspecified order. (The program should not rely on any order.)

AT

```
►►──AT(index)────────────────────────────────────────────────────────►◄
```

Returns the item associated with index *index*. If the relation contains more than one item associated with index *index*, the item returned is unspecified. (The program should not rely on any particular item being returned.) If the relation has no item associated with index *index*, this method returns the NIL object.

HASINDEX

```
►►──HASINDEX(index)──────────────────────────────────────────────────►◄
```

Returns 1 (true) if the collection contains any item associated with index *index*, or 0 (false).

HASITEM

```
►►──HASITEM(item,index)──────────────────────────────────────────────►◄
```

Returns 1 (true) if the relation contains the member item *item* (associated with index *index*, or 0 (false).

INDEX

```
►►──INDEX(item)──────────────────────────────────────────────────────►◄
```

Returns the index for item *item*. If there is more than one index associated with item *item*, the one this method returns is not defined.

ITEMS

```
►►──ITEMS──┬─────────┬───────────────────────────────────────────────►◄
           └─(index)─┘
```

Returns the number of relation items with index *index*. If you specify no *index*, this method returns the total number of items associated with all indexes in the relation.

MAKEARRAY

```
►►──MAKEARRAY────────────────────────────────────────────────────────►◄
```

Returns a single-index array containing the index objects. The array indexes range from 1 to the number of items. The collection items appear in the array in an unspecified order. (The program should not rely on any order.)

PUT

```
►►──PUT(item,index)──────────────────────────────────────────────────►◄
```

Makes the object *item* a member item of the relation and associates it with index *index*. If the relation already contains any items with index *index*, this method adds a new member item *item* with the same index, without removing any existing member items.

REMOVE

```
►►──REMOVE(index)──────────────────────────────────────────────────────►◄
```

Returns and removes from a relation the member item with index *index*. If the relation contains more than one item associated with index *index*, the item returned and removed is unspecified. If no item has index *index*, this method returns the NIL object and removes nothing.

REMOVEITEM

```
►►──REMOVEITEM(item,index)─────────────────────────────────────────────►◄
```

Returns and removes from a relation the member item *item* (associated with index *index*). If *value* is not a member item associated with index *index*, this method returns the NIL object and removes no item.

SUPPLIER

```
►►──SUPPLIER──┬─────────┬──────────────────────────────────────────────►◄
              └─(index)─┘
```

Returns a supplier object for the collection. After you have obtained a supplier, you can send it messages (see "The Supplier Class" on page 239) to enumerate all the items that were in the collection at the time of the supplier's creation. The supplier enumerates the items in an unspecified order. (The program should not rely on any order.) If you specify *index*, the supplier enumerates all of the items in the relation with the specified index.

DIFFERENCE

```
►►──DIFFERENCE(argument)───────────────────────────────────────────────►◄
```

Returns a new collection (of the same class as the receiver) containing only those items that the *argument* collection does *not* contain (with the same associated index). The *argument* can be any object described in "The Argument Collection Classes" on page 157.

INTERSECTION

```
►►──INTERSECTION(argument)─────────────────────────────────────────────►◄
```

Returns a new collection (of the same class as the receiver) containing only those items that are in *both* the receiver collection and the *argument* collection with the same associated index. The *argument* can be any object described in “The Argument Collection Classes” on page 157.

SUBSET

```
►►──SUBSET(argument)──────────────────────────────────────────────────►◄
```

Returns 1 (true) if all items in the receiver collection are also contained in the *argument* collection with the same associated index; returns `0` (false) otherwise. The *argument* can be any object described in “The Argument Collection Classes” on page 157.

UNION

```
►►──UNION(argument)───────────────────────────────────────────────────►◄
```

Returns a new collection containing all items from the receiver collection and the *argument* collection. The *argument* can be any object described in “The Argument Collection Classes” on page 157.

XOR

```
►►──XOR(argument)─────────────────────────────────────────────────────►◄
```

Returns a new collection of the same class as the receiver that contains all items from the receiver collection and the *argument* collection. All index-item pairs that appear in both collections are removed. The *argument* can be any object described in “The Argument Collection Classes” on page 157.

Examples

```
/* Use a relation to express parent-child relationships */
family = .relation~new
family['Henry'] = 'Peter'    /* Peter is Henry's child   */
family['Peter'] = 'Bridget'  /* Bridget is Peter's child */
family['Henry'] = 'Jane'     /* Jane is Henry's child    */

/* Show all children of Henry recorded in the family relation */
henrys_kids = family~allat('Henry')
Say 'Here are all the listed children of Henry:'
Do kid Over henrys_kids
  Say ' 'kid
End

/* Show all parents of Bridget recorded in the family relation */
```

```
bridgets_parents = family~allindex('Bridget')
Say 'Here are all the listed parents of Bridget:'
Do parent Over bridgets_parents
  Say ' 'parent
End

/* Display all the grandparent relationships we know about. */
checked_for_grandkids = .set~new        /* Records those we have checked      */
Do grandparent Over family              /* Iterate for each index in family   */
  If checked_for_grandkids~hasindex(grandparent)
    Then Iterate                        /* Already checked this one           */
  kids = family~allat(grandparent)      /* Current grandparent's children     */
  Do kid Over kids                      /* Iterate for each item in kids      */
    grandkids = family~allat(kid)       /* Current kid's children             */
    Do grandkid Over grandkids          /* Iterate for each item in grandkids */
      Say grandparent 'has a grandchild named' grandkid'.'
    End
  End
  checked_for_grandkids~put(grandparent) /* Add to already-checked set         */
End
```

The Set Class

A *set* is a collection containing the member items where the index is the same as the item. Any object can be placed in a set. There can be only one occurrence of any object in a set.

The Set class is a subclass of the Table class. In addition to its own methods, it inherits the methods of the Object class (see "The Object Class" on page 179) and the Table class.

Methods the Set class defines:

OF (Class method)
[]
[]=
AT
HASINDEX
ITEMS
MAKEARRAY
PUT
REMOVE
SUPPLIER

Set-operator methods inherited from the Table class:

DIFFERENCE
INTERSECTION

SUBSET
UNION
XOR

Methods inherited from the Object class:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Set class also has available class methods that its metaclass, the Class class, defines.

OF (Class Method)

```
           ┌─,──────┐
           ▼        │
►►──OF(───item──────┴─)──────────────────────────────────────────►◄
```

Returns a newly created set containing the specified *item* objects.

[]

```
►►──[index]──────────────────────────────────────────────────────►◄
```

Returns the same item as the AT method. See “AT” on page 149.

[]=

```
►►──[index]=item─────────────────────────────────────────────────►◄
```

This method is the same as the PUT method. See “PUT” on page 149.

AT

```
►►──AT(index)─────────────────────────────────────────────────────►◄
```

Returns the item associated with index *index*. If the collection has no item associated with *index*, this method returns the NIL object.

HASINDEX

```
►►──HASINDEX(index)───────────────────────────────────────────────►◄
```

Returns 1 (true) if the collection contains any item associated with index *index*, or 0 (false).

ITEMS

```
►►──ITEMS─────────────────────────────────────────────────────────►◄
```

Returns the number of items in the collection.

MAKEARRAY

```
►►──MAKEARRAY─────────────────────────────────────────────────────►◄
```

Returns a single-index array containing the index objects. The array indexes range from 1 to the number of items. The collection items appear in the array in an unspecified order. (The program should not rely on any order.)

PUT

```
►►──PUT(item──┬────────┬──)───────────────────────────────────────►◄
              └─,index─┘
```

Makes the object *item* a member item of the collection and associates it with index *index*. If you specify *index*, it must be the same as *item*.

REMOVE

```
►►──REMOVE(index)─────────────────────────────────────────────────►◄
```

Returns and removes from a collection the member item with index *index*. If no item has index *index*, this method returns the NIL object and removes no item.

SUPPLIER

```
►►—SUPPLIER————————————————————————————————————————————————————►◄
```

Returns a supplier object for the collection. After you have obtained a supplier, you can send it messages (see “The Supplier Class” on page 239) to enumerate all the items that were in the collection at the time of the supplier's creation. The supplier enumerates the items in an unspecified order. (The program should not rely on any order.)

The Table Class

A *table* is a collection with indexes that can be any object the user supplies. In a table, each item is associated with a single index, and there can be only one item for each index (unlike a relation, which can contain more than one item with the same index).

Methods the Table class defines:

[]
[]=
AT
HASINDEX
ITEMS
MAKEARRAY
PUT
REMOVE
SUPPLIER

Set-operator methods the Table class defines:

DIFFERENCE
INTERSECTION
SUBSET
UNION
XOR

Methods inherited from the Object class:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==

CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Table class also has available class methods that its metaclass, the Class class, defines.

[]

```
▶▶──[index]────────────────────────────────────────────────────────▶◀
```

Returns the same item as the AT method. See “AT”.

[]=

```
▶▶──[index]=item───────────────────────────────────────────────────▶◀
```

This method is the same as the PUT method. See “PUT” on page 152.

AT

```
▶▶──AT(index)──────────────────────────────────────────────────────▶◀
```

Returns the item associated with index *index*. If the collection has no item associated with *index*, this method returns the NIL object.

HASINDEX

```
▶▶──HASINDEX(index)────────────────────────────────────────────────▶◀
```

Returns 1 (true) if the collection contains any item associated with index *index*, or 0 (false).

ITEMS

```
►►──ITEMS──────────────────────────────────────────────────────────►◄
```

Returns the number of items in the collection.

MAKEARRAY

```
►►──MAKEARRAY──────────────────────────────────────────────────────►◄
```

Returns a single-index array containing the index objects. The array indexes range from 1 to the number of items. The collection items appear in the array in an unspecified order. (The program should not rely on any order.)

PUT

```
►►──PUT(item,index)────────────────────────────────────────────────►◄
```

Makes the object *item* a member item of the collection and associates it with index *index*. The new item replaces any existing items associated with index *index*.

REMOVE

```
►►──REMOVE(index)──────────────────────────────────────────────────►◄
```

Returns and removes from a collection the member item with index *index*. If no item has index *index*, this method returns the NIL object and removes no item.

SUPPLIER

```
►►──SUPPLIER───────────────────────────────────────────────────────►◄
```

Returns a supplier object for the collection. After you have obtained a supplier, you can send it messages (see "The Supplier Class" on page 239) to enumerate all the items that were in the collection at the time of the supplier's creation. The supplier enumerates the items in an unspecified order. (The program should not rely on any order.)

DIFFERENCE

```
►►──DIFFERENCE(argument)─────────────────────────────────────────────►◄
```

Returns a new collection (of the same class as the receiver) containing only those index-item pairs of the receiver whose indexes the *argument* collection does *not* contain. The *argument* can be any object described in "The Argument Collection Classes" on page 157. The *argument* must also allow all of the index values in the receiver collection.

INTERSECTION

```
►►──INTERSECTION(argument)───────────────────────────────────────────►◄
```

Returns a new collection (of the same class as the receiver) containing only those index-item pairs of the receiver whose indexes are in *both* the receiver collection and the *argument* collection. The *argument* can be any object described in "The Argument Collection Classes" on page 157. The *argument* must also allow all of the index values in the receiver collection.

SUBSET

```
►►──SUBSET(argument)─────────────────────────────────────────────────►◄
```

Returns 1 (true) if all indexes in the receiver collection are also contained in the *argument* collection; returns 0 (false) otherwise. The *argument* can be any object described in "The Argument Collection Classes" on page 157. The *argument* must also allow all of the index values in the receiver collection.

UNION

```
►►──UNION(argument)──────────────────────────────────────────────────►◄
```

Returns a new collection of the same class as the receiver that contains all the items from the receiver collection and selected items from the *argument* collection. This method includes an item from *argument* in the new collection only if there is no item with the same associated index in the receiver collection and the method has not already included an item with the same index. The order in which this method selects items in *argument* is unspecified. (The program should not rely on any order.) See also the UNION method of the Directory (see "UNION" on page 132) and Relation (see "UNION" on page 146) classes. The *other* can be any object described in "The

Argument Collection Classes” on page 157. The *argument* must also allow all of the index values in the receiver collection.

XOR

```
▸▸──XOR(argument)──────────────────────────────────────────────▸◂
```

Returns a new collection of the same class as the receiver that contains all items from the receiver collection and the *argument* collection; all indexes that appear in both collections are removed. The *argument* can be any object described in “The Argument Collection Classes” on page 157. The *argument* must also allow all of the index values in the receiver collection.

The Concept of Set Operations

The following sections describe the concept of set operations to help you work with set operators, in particular if the receiver collection class differs from the argument collection class.

REXX provides the following set-operator methods:

- DIFFERENCE
- INTERSECTION
- SUBSET
- UNION
- XOR

These methods are only available to instances of the following collection classes:

- Directory
- Table and its subclass Set
- Relation and its subclass Bag

The collection classes Array, List, and Queue do not have set-operator methods but their instances can be used as the argument collections.

Set operations have the following form:

```
result = receiver~setoperator(argument)
```

where:

receiver
: is the collection receiving the set-operator message. It can be an instance of the Directory, Relation, Table, Set, or Bag collection class.

setoperator
: is the set-operator method used.

argument
: is the argument collection supplied to the method. It can be an instance of one of the receiver collection classes or of a collection class that does not have set-operator methods, namely Array, List, or Queue.

The resulting collection is of the same class as the receiver collection.

The Principles of Operation

A set operation is performed by iterating over the elements of the receiver collection to compare each element of the receiver collection with each element of the argument collection. The element is defined as the tuple <*index*,*item*> (see "Determining the Identity of an Item" on page 156). Depending on the set-operator method and the result of the comparison, an element of the receiver collection is, or is not, included in the resulting collection. A receiver collection that allows for duplicate elements can, depending on the set-operator method, also accept elements of the argument collection after they have been coerced to the type of the receiver collection.

The following examples are to help you understand the semantics of set operations. The collections are represented as a list of elements enclosed in curly brackets. The list elements are separated by a comma.

Set Operations on Collections without Duplicates

Assume that the example sets are `A={a,b}` and `B={b,c,d}`. The result of a set operation is another set. The only exception is a subset resulting in a Boolean .true or .false. Using the collection `A` and `B`, the different set operators produce the following:

UNION operation
: All elements of `A` and `B` are united:

```
A UNION B = {a,b,c,d}
```

DIFFERENCE operation
: The resulting collection contains all elements of the first set except for those that also appear in the second set. The system iterates over the elements of the second set and removes them from the first set one by one.

```
A DIFFERENCE B = {a}
B DIFFERENCE A = {c,d}
```

XOR operation
: The resulting collection contains all elements of the first set that are not in the second set and all elements of the second set that are not in the first set:

```
A XOR B = {a,c,d}
```

INTERSECTION operation

The resulting collection contains all elements that appear in both sets:

```
A INTERSECTION B = {b}
```

SUBSET operation

Returns `.true` if the first set contains only elements that also appear in the second set, otherwise it returns `.false`:

```
A SUBSET B = .false
B SUBSET A = .false
```

Set-Like Operations on Collections with Duplicates

Assume that the example bags are `A={a,b,b}` and `B={b,b,c,c,d}`. The result of any set-like operation is a collection, in this case a bag. The only exception is SUBSET resulting in a Boolean .true or .false. Using the collections `A` and `B`, the different set-like operators produce the following:

UNION operation

All elements of `A` and `B` are united:

```
A UNION B = {a,b,b,b,b,c,c,d}
```

DIFFERENCE operation

The resulting collection contains all elements of the first bag except for those that also appear in the second bag. The system iterates over the elements of the second bag and removes them from the first bag one by one.

```
A DIFFERENCE B = {a}
B DIFFERENCE A = {c,c,d}
```

XOR operation

The resulting collection contains all elements of the first bag that are not in the second bag and all elements of the second bag that are not in the second bag:

```
A XOR B = {a,c,c,d}
```

INTERSECTION operation

The resulting collection contains all elements that appear in both bags:

```
A INTERSECTION B = {b,b}
```

SUBSET operation

Returns `.true` if the first set contains only elements that also appear in the second set, otherwise it returns `.false`:

```
A SUBSET B = .false
B SUBSET A = .false
```

Determining the Identity of an Item

Set operations require the definition of the identity of an element to determine whether a certain element exists in the receiver collection. The element of a collection is conceived as the tuple <*index,item*>. The *index* is used as the identification tag associated with the item. Depending on the collection class, the index is an instance of a particular class, for example, the string class for a directory element, an integer for an array, or any arbitrary class for a relation. The Array class is an exception because it can be multidimensional having more than one index. However, as a collection, it is conceptionally linearized by the set operator.

For collections of collection classes that require unique indexes, namely the Set, Table, and Directory classes, an item is identified by its *index*. For collections of collection classes that allow several items to have the same index, namely the Relation class, an item is identified by both its *index* and its *item*. For the Bag and the Set subclasses, where several items can have the same index but *index* and *item* must be identical, the item is identified by its *index*. According to this concept, an item of a collection is identified as follows:

- HASINDEX(*index*) for Bag, Directory, Set, and Table collections
- HASITEM(*item,index)* for the Relation collections

Items of the Array, List, and Queue collections are identified by the *item*, not the *index*. The index is only used as a means to access the item but carries no semantics. In a Queue collection class, for example, the index of a particular item changes when another item is added to the queue and therefore is not a permanent identification of an item.

The Argument Collection Classes

A argument collection can be an instance of any collection class, including the Array, List, and Queue classes, which do not have set-operator methods.

If the collection does not contain a UNION method, the following must apply:

- The collection must support the MAKEARRAY method so that the set or set-like operator can iterate over the supplied elements.
- The collection must conceptionally be coerced into a bag-like collection before the set operation. Conceptionally, sparse arrays are condensed and multidimensional arrays are linearized.

Collections having the UNION method must support the SUPPLIER method.

The Receiver Collection Classes

In addition to the set and set-like methods, a collection must support the following methods to qualify as a receiver collection:

- Methods for collections not allowing elements with duplicate indexes:

  HASINDEX

  PUT or []=

  REMOVE

  ITEMS

- Methods for collections allowing elements with duplicate indexes:

  HASITEM; for bags, HASINDEX is sufficient

  AT or []

  PUT or []=

  REMOVEITEM; for bags, REMOVE is sufficient

  ITEMS

Classifying Collections

To determine whether the items in a collection class can be used in a set operation, check the following criteria:

- Is an object a collection?

  To answer this question, send the HASMETHOD method with parameter "hasindex" to *object*:

  ```
  ::ROUTINE isCollection
  use arg object
  return object~hasmethod("hasindex")
  ```

  This function returns TRUE if the object is an instance of the Array, List, Queue, Set, Bag, Relation, or Table collection class.
- Does the collection class have set-operator methods?

  To answer this question, send the HASMETHOD method with parameter "union" to *object*:

  ```
  ::ROUTINE hasSetOperators
  use arg object
  return object~hasmethod("union")
  ```

  This function returns TRUE if the object is an instance of the Set, Bag, Relation, or Table collection class.

Chapter 6. Other Classes

This chapter describes the following classes:
- Alarm class
- Class class
- Message class
- Method class
- Monitor class
- Object class
- Stem class
- Stream class
- String class
- Supplier class
- WindowsProgramManager class
- WindowsRegistry class
- WindowsEventLog class
- WindowsManager class
- WindowObject class
- MenuObject class
- WindowsClipboard class
- OLEObject class

The Alarm Class

An *alarm* object provides timing and notification capability by supplying a facility to send any message to any object at a given time. You can cancel an alarm before it sends its message.

The Alarm class is a subclass of the Object class.

Methods the Alarm class defines:

CANCEL
INIT (Overrides Object class method)

Methods inherited from the Object class:

NEW (Class Method)
Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD

OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Alarm class also has available class methods that its metaclass, the Class class, defines.

CANCEL

```
►►──CANCEL──────────────────────────────────────────────────────────►◄
```

Cancels the pending alarm request represented by the receiver. This method takes no action if the specified time has already been reached.

INIT

```
►►──INIT(atime,message)─────────────────────────────────────────────►◄
```

Sets up an alarm for a future time *atime*. At this time, the alarm object sends the message that *message*, a message object, specifies. (See “The Message Class” on page 170.) The *atime* is a string. You can specify this in the default format (`'hh:mm:ss'`) or as a number of seconds starting at the present time. If you use the default format, you can specify a date in the default format (`'dd Mmm yyyy'`) after the time with a single blank separating the time and date. Leading and trailing blanks are not allowed in the *atime*. If you do not specify a date, the language processor uses the first future occurrence of the specified time. You can use the CANCEL method to cancel a pending alarm. See “Initialization” on page 104 for more information.

Examples

The following code sets up an alarm at 5:10 p.m. on October 8, 1996. (Assume today's date is October 5, 1996.)

```
/* Alarm Examples */

PersonalMessage=.MyMessageClass~new('Call the Bank')
msg=.message~new(PersonalMessage,'RemindMe')

a=.alarm~new('17:10:00 8 Oct 1996', msg)
exit
/* "::CLASS" on page 87  describes the ::CLASS directive */
/* "::METHOD" on page 89  describes the ::METHOD directive */
```

```
::CLASS MyMessageClass public
::Method init
expose inmsg
use arg inmsg
::Method RemindMe
expose inmsg
say 'It is now' 'TIME'('C')'.Please 'inmsg
/* On the specified data and time, displays the following message: */
/* 'It is now 5:10pm. Please Call the Bank' */
```

For the following example, the user uses the same code as in the preceding example to define `msg`, a message object to run at the specified time. The following code sets up an alarm to run the `msg` message object in 30 seconds from the current time:

```
a=.alarm~new(30,msg)
```

The Class Class

The Class class is like a factory producing the factories that produce objects. It is a subclass of the Object class. The instance methods of the Class class are also the class methods of all classes.

Methods the Class class defines: (They are all both class and instance methods.)

BASECLASS
DEFAULTNAME (Overrides Object class method)
DEFINE
DELETE
ENHANCED
ID
INHERIT
INIT (Overrides Object class method)
METACLASS
METHOD
METHODS
MIXINCLASS
NEW (Overrides Object class method)
QUERYMIXINCLASS
SUBCLASS
SUBCLASSES
SUPERCLASSES
UNINHERIT

Methods inherited from the Object class:

Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
HASMETHOD
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

BASECLASS

```
▸▸──BASECLASS──────────────────────────────────────────────────────────────▸◂
```

Returns the base class associated with the class. If the class is a mixin class, the base class is the first superclass that is not also a mixin class. If the class is not a mixin class, the base class is the class receiving the BASECLASS message.

DEFAULTNAME

```
▸▸──DEFAULTNAME────────────────────────────────────────────────────────────▸◂
```

Returns a short human-readable string representation of the class. The string returned is of the form

`The id class`

where *id* is the identifier assigned to the class when it was created.

Examples:

```
say .array~defaultname     /* Displays "The Array class"   */
say .account~defaultname   /* Displays "The ACCOUNT class" */
say .savings~defaultname   /* Displays "The Savings class" */

::class account            /* Name is all upper case       */
::class 'Savings'          /* String name is mixed case    */
```

DEFINE

```
▸▸──DEFINE(methodname─┬─────────┬─)────────────────────────────────────────▸◂
                      └─,method─┘
```

Incorporates the method object *method* in the receiver class’s collection of instance methods. The language processor translates the method name *methodname* to uppercase. Using the DEFINE method replaces any existing definition for *methodname* in the receiver class.

If you omit *method*, the method name *methodname* is made unavailable for the receiver class. Sending a message of that name to an instance of the class causes the UNKNOWN method (if any) to be run.

The *method* argument can be a string containing a method source line instead of a method object. Alternatively, you can pass an array of strings containing individual method lines. Either way, DEFINE creates an equivalent method object.

Notes:

1. The classes REXX provides do not permit changes or additions to their method definitions.
2. The DEFINE method is a protected method.

Example:

```
bank_account=.object~subclass('Account')
bank_account~define('TYPE','return "a bank account"')
```

DELETE

```
►►──DELETE(methodname)──────────────────────────────────────────────►◄
```

Removes the receiver class's definition for the method name *methodname*. If the receiver class defined *methodname* as unavailable with the DEFINE method, this definition is nullified. If the receiver class had no definition for *methodname*, no action is taken.

Notes:

1. The classes REXX provides do not permit changes or additions to their method definitions.
2. DELETE deletes only methods the target class defines. You cannot delete inherited methods the target's superclasses define.
3. The DELETE method is a protected method.

Example:

```
myclass=.object~subclass('Myclass')        /* After creating a class  */
myclass~define('TYPE','return "my class"') /* and defining a method   */
myclass~delete('TYPE')                     /* this deletes the method */
```

ENHANCED

```
►►──ENHANCED(methods──┬─────────────────┬──)──────────────────────────►◄
                      │  ┌────────────┐ │
                      │  ▼            │ │
                      └───,argument───┴─┘
```

Returns an enhanced new instance of the receiver class, with object methods that are the instance methods of the class, enhanced by the methods in the collection *methods*. The collection indexes are the names of the enhancing methods, and the items are the method objects (or strings or arrays of strings containing method code). (See the description of DEFINE in "DEFINE" on page 162.) You can use any collection that supports a SUPPLIER method.

ENHANCED sends an INIT message to the created object, passing the *argument*s specified on the ENHANCED method.

Example:

```
/* Set up rclass with class method or methods you want in your */
/* remote class */
rclassmeths = .directory~new

rclassmeths['DISPATCH']=d_source     /* d_source must have code for a  */
                                     /* DISPATCH  method.              */
/* The following sends INIT('Remote Class') to a new instance */
rclass=.class~enhanced(rclassmeths,'Remote Class')
```

ID

```
►►──ID────────────────────────────────────────────────────────────────►◄
```

Returns the class identity (instance) string. (This is the string that is an argument on the SUBCLASS and MIXINCLASS methods.) The string representations of the class and its instances contain the class identity.

Example:

```
myobject=.object~subclass('my object')  /* Creates a subclass    */
say myobject~id                         /* Produces: 'my object' */
```

INHERIT

```
►►──INHERIT(classobj──┬───────────┬──)────────────────────────────────►◄
                      └─,classpos─┘
```

Causes the receiver class to inherit the instance and class methods of the class object *classobj*. The *classpos* is a class object that specifies the position of the new superclass in the list of superclasses. (You can use the SUPERCLASSES method to return the immediate superclasses.)

The new superclass is inserted in the search order after the specified class. If the *classpos* class is not found in the set of superclasses, an error is raised. If you do not specify *classpos*, the new superclass is added to the end of the superclasses list.

Inherited methods can take precedence only over methods defined at or above the base class of the *classobj* in the class hierarchy. Any subsequent change to the instance methods of *classobj* takes immediate effect for all the classes that inherit from it.

The new superclass *classobj* must be created with the MIXINCLASS option of the ::CLASS directive or the MIXINCLASS method and the base class of the *classobj* must be a direct superclass of the receiver object. The receiver must not already descend from *classobj* in the class hierarchy and vice versa.

The method search order of the receiver class after INHERIT is the same as before INHERIT, with the addition of *classobj* and its superclasses (if not already present).

Notes:

1. You cannot change the classes that REXX provides by sending INHERIT messages.
2. The INHERIT method is a protected method.

Example:

```
room~inherit(.location)
```

INIT

```
►►──INIT(classid)──────────────────────────────────────────────────►◄
```

Sets the receiver class identity to the string *classid*. You can use the ID method (described previously) to return this string, which is the class identity. See “Initialization” on page 104 for more information.

METACLASS

```
►►──METACLASS──────────────────────────────────────────────────────►◄
```

Returns the receiver class's default metaclass. This is the class used to create subclasses of this class when you send SUBCLASS or MIXINCLASS messages (with no metaclass arguments). If the receiver class is an object class (see “Object Classes” on page 95), this is also the class used to create the receiver class. The instance methods of the default metaclass are the class methods of the receiver class. For more information about class methods, see “Object Classes” on page 95. See also the description of the SUBCLASS method in “SUBCLASS” on page 168.

METHOD

```
►►──METHOD(methodname)──────────────────────────────────────────────────►◄
```

Returns the method object for the receiver class's definition for the method name *methodname*. If the receiver class defined *methodname* as unavailable, this method returns the NIL object. If the receiver class did not define *methodname*, the language processor raises an error.

Example:

```
/* Create and retrieve the method definition of a class */
myclass=.object~subclass('My class')   /* Create a class         */
mymethod=.method~new(' ','Say arg(1)') /* Create a method object */
myclass~define('ECHO',mymethod)        /* Define it in the class */
method_source = myclass~method('ECHO')~source     /* Extract it  */
say method_source                 /* Says 'an Array'            */
say method_source[1]              /* Shows the method source code */
```

METHODS

```
►►──METHODS──┬────────────────┬──────────────────────────────────────────►◄
             └─(class_object)─┘
```

Returns a supplier object for all the instance methods of the receiver class and its superclasses, if you specify no argument. If *class_object* is the NIL object, METHODS returns a supplier object for only the instance methods of the receiver class. If you specify a *class_object*, this method returns a supplier object containing only the instance methods that *class_object* defines. If you send appropriate messages to a supplier object, the supplier enumerates all the instance methods existing at the time of the supplier's creation. (See “The Supplier Class” on page 239 for details.)

Note: Methods that have been hidden with a SETMETHOD or DEFINE method are included with the other methods that METHODS returns. The hidden methods have the NIL object for the associated method.

Example:

```
objsupp=.object~methods
do while objsupp~available
say objsupp~index          /* Says all instance methods */
objsupp~next               /* of the Object class       */
end
```

MIXINCLASS

```
>>-MIXINCLASS(classid-+----------------------------+-)--------------><
                      +-,metaclass-+-----------+-+
                                   +-,methods--+
```

Returns a new mixin subclass of the receiver class. You can use this method to create a new mixin class that is a subclass of the superclass to which you send the message. The *classid* is a string that identifies the new mixin subclass. You can use the ID method to retrieve this string.

The *metaclass* is a class object. If you specify *metaclass*, the new subclass is an instance of *metaclass*. (A *metaclass* is a class that you can use to create a class, that is, a class whose instances are classes. The Class class and its subclasses are metaclasses.)

If you do not specify a *metaclass*, the new mixin subclass is an instance of the default metaclass of the receiver class. For subclasses of the Object class, the default metaclass is the Class class.

The *methods* is a collection whose indexes are the names of methods and whose items are method objects (or strings or arrays of strings containing method code). If you specify *methods*, the new class is enhanced with class methods from this collection. (The metaclass of the new class is not affected.)

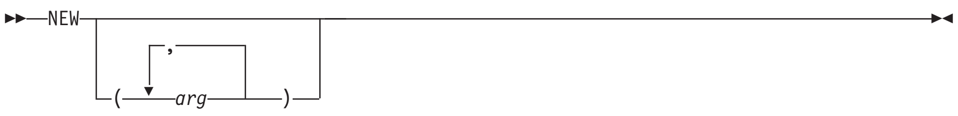
The METACLASS method returns the metaclass of a class.

The method search order of the new subclass is the same as that of the receiver class, with the addition of the new subclass at the start of the order.

Example:

```
buyable=.object~mixinclass('Buyable')  /* New subclass is buyable    */
                                       /* Superclass is Object class */
```

NEW

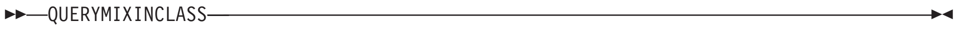


Returns a new instance of the receiver class, whose object methods are the instance methods of the class. This method initializes a new instance by running its INIT methods. (See “Initialization” on page 104.) NEW also sends an INIT message. If you specify *arg*s, NEW passes these arguments on the INIT message.

Example:

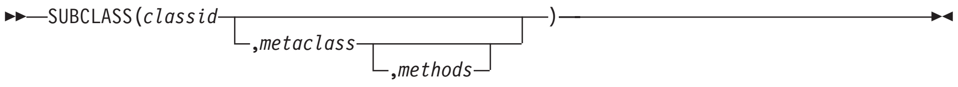
```
/* NEW method example */
a = .account~new            /* -> Object variable balance=0          */
y = .account~new(340.78)    /* -> Object variable balance=340.78     */
                            /*    plus free toaster oven             */
::class account subclass object
::method INIT               /* Report time each account created      */
                            /* plus free toaster when more than $100  */
Expose balance
Arg opening_balance
Say 'Creating' self~objectname 'at time' time()
If datatype(opening_balance, 'N') then balance = opening_balance
else balance = 0
If balance > 100 then Say '  You win a free toaster oven'
```

QUERYMIXINCLASS



Returns 1 (true) if the class is a mixin class, or 0 (false).

SUBCLASS



Returns a new subclass of the receiver class. You can use this method to create a new class that is a subclass of the superclass to which you send the message. The *classid* is a string that identifies the subclass. (You can use the ID method to retrieve this string.)

The *metaclass* is a class object. If you specify *metaclass*, the new subclass is an instance of *metaclass*. (A *metaclass* is a class that you can use to create a class, that is, a class whose instances are classes. The Class class and its subclasses are metaclasses.)

If you do not specify a *metaclass*, the new subclass is an instance of the default metaclass of the receiver class. For subclasses of the Object class, the default metaclass is the Class class.

The *methods* is a collection whose indexes are the names of methods and whose items are method objects (or strings or arrays of strings containing method code). If you specify *methods*, the new class is enhanced with class methods from this collection. (The metaclass of the new class is not affected.)

The METACLASS method returns the metaclass of a class.

The method search order of the new subclass is the same as that of the receiver class, with the addition of the new subclass at the start of the order.

Example:

```
room=.object~subclass('Room')   /* Superclass is .object     */
                                /* Subclass is room          */
                                /* Subclass identity is Room */
```

SUBCLASSES

```
►►──SUBCLASSES──────────────────────────────────────────────────────►◄
```

Returns the immediate subclasses of the receiver class in the form of a single-index array of the required size, in an unspecified order. (The program should not rely on any order.) The array indexes range from 1 to the number of subclasses.

SUPERCLASSES

```
►►──SUPERCLASSES────────────────────────────────────────────────────►◄
```

Returns the immediate superclasses of the receiver class in the form of a single-index array of the required size. The immediate superclasses are the original class used on a SUBCLASS or a MIXINCLASS method, plus any additional superclasses defined with the INHERIT method. The array is in the order in which the class has inherited the classes. The original class used on a SUBCLASS or MIXINCLASS method is the first item of the array. The array indexes range from 1 to the number of superclasses.

Example:

```
z=.class~superclasses
/* To obtain the information this returns, you could use:     */
do i over z
  say i
end
```

UNINHERIT

```
►►──UNINHERIT(classobj)──────────────────────────────────────────────►◄
```

Nullifies the effect of any previous INHERIT message sent to the receiver for the class *classobj*.

Note: You cannot change the classes that REXX provides by sending UNINHERIT messages.

Example:

```
location=.object~mixinclass('Location')
room=.object~subclass('Room')~~inherit(location) /* Creates subclass */
/* and specifies inheritance */
room~UNINHERIT(location)
```

The Message Class

A *message object* provides for the deferred or asynchronous sending of a message. You can create a message object by using the NEW or ENHANCED method of the Message class or the START method of the Object class (see "START" on page 184). The Message class is a subclass of the Object class.

Methods the Message class defines:

COMPLETED
INIT (Overrides Object class method)
NOTIFY
RESULT
SEND
START (Overrides Object class method)

Methods inherited from the Object class:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME

HASMETHOD
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
STRING
UNSETMETHOD

Note: The Message class also has available class methods that its metaclass, the Class class, defines.

COMPLETED

```
►►──COMPLETED──────────────────────────────────────────────────────────►◄
```

Returns 1 if the message object has completed its message, or 0. You can use this method instead of sending RESULT and waiting for the message to complete.

INIT

```
►►──INIT(target,messagename──┬─────────────────────────────┬──)────────►◄
                             │              ┌────────────┐ │
                             │              ▼            │ │
                             ├─,Individual──┬────────────┴─┤
                             │              └─,argument─┘  │
                             └─,Array,argument─────────────┘
```

Initializes the message object for sending the message name *messagename* to object *target*.

The *messagename* can be a string or an array. If *messagename* is an array object, its first item is the name of the message and its second item is a class object to use as the starting point for the method search. For more information, see Table 1 on page 108.

If you specify the Individual or Array option, any remaining arguments are arguments for the message. (You need to specify only the first letter; the language processor ignores all characters following it.)

Individual	If you specify this option, specifying *argument* is optional. The language processor passes any *argument*s as message arguments to *target* in the order you specify them.
Array	If you specify this option, you must specify an *argument*,

which is an array object. (See "The Array Class" on page 118.) The language processor then passes the member items of the array to *target*. When the language processor passes the arguments taken from the array, the first argument is at index 1, the second argument at index 2, and so on. If you omitted any indexes when creating the array, the language processor omits their corresponding message arguments when passing the arguments.

If you specify neither Individual nor Array, the message sent has no arguments.

Note: This method does not send the message *messagename* to object *target*. The SEND or START method (described later) sends the message.

NOTIFY

```
►►──NOTIFY(message)──────────────────────────────────────────────────►◄
```

Requests notification about the completion of processing of the message SEND or START. The message object *message* is sent as the notification. You can use NOTIFY to request any number of notifications. After the notification message, you can use the RESULT method to obtain any result from the messages SEND or START.

Example:

```
/* Event-driven greetings */

.prompter~new~prompt(.nil)

:class prompter

::method prompt
  expose name
  use arg msg

  if msg \= .nil then do
    name = msg~result
    if name = 'quit' then return
    say 'Hello,' name
  end

  say 'Enter your name ("quit" to quit):'

  /* Send the public default object .INPUT a LINEIN message asynchronously */
  msg=.message~new(.input,'LINEIN')~~start

  /* Sends self~prompt(msg) when data available   */
  msg~notify(.message~new(self,'PROMPT','I',msg))

  /* Don't leave until user has entered "quit"    */
  guard on when name='quit'
```

RESULT

```
►►──RESULT──────────────────────────────────────────────────────────────►◄
```

Returns the result of the message SEND or START. If message processing is not yet complete, this method waits until it completes. If the message SEND or START raises an error condition, this method also raises an error condition.

Example:

```
/* Example using RESULT method */
string='700'                 /* Create a new string object, string   */
bond=string~start('REVERSE') /* Create a message object, bond, and   */
                             /* start it.  This sends a REVERSE      */
                             /* message to string, giving bond       */
                             /* the result.                          */

/* Ask bond for the result of the message */
say 'The result of message was' bond~result     /* Result is 007     */
```

SEND

```
►►──SEND──┬────────────┬────────────────────────────────────────────────►◄
          └─(target)───┘
```

Returns the result (if any) of sending the message. If you specify *target*, this method sends the message to *target*. Otherwise, this method sends the message to the *target* you specified when the message object was created. SEND does not return until message processing is complete.

You can use the NOTIFY method to request notification that message processing is complete. You can use the RESULT method to obtain any result from the message.

START

```
►►──START──┬────────────┬───────────────────────────────────────────────►◄
           └─(target)───┘
```

Sends the message to start processing at a specific target whereas the sender continues processing. If you specify *target*, this method sends the message to *target*. Otherwise, this method sends the message to the *target* that you specified when the message object was created. This method returns as soon as possible and does not wait until message processing is complete. When message processing is complete, the message object retains any result and holds it until the sender requests it by sending a RESULT message. You can use the NOTIFY method to request notification that message processing is complete.

Example

```
/* Using Message class methods */
/* Note: In the following example, ::METHOD directives define class Testclass */
                                                /* with method SHOWMSG       */

ez=.testclass~new                 /* Creates a new instance of Testclass     */
mymsg=ez~start('SHOWMSG','Hello, Ollie!',5)     /* Creates and starts        */
                                                /* message mymsg to send     */
                                                /* SHOWMSG to ez             */

/* Continue with main processing while SHOWMSG runs concurrently             */
do 5
  say 'Hello, Stan!'
end

/* Get final result of the SHOWMSG method from the mymsg message object      */
say mymsg~result
say 'Goodbye, Stan...'
exit

::class testclass public            /* Directive defines Testclass           */

::method showmsg                    /* Directive creates new method SHOWMSG  */
use arg text,reps                   /* class Testclass                       */
do reps
  say text
end
reply 'Bye Bye, Ollie...'
return
```

The following output is possible:

```
Hello, Ollie!
Hello, Stan!
Hello, Ollie!
Hello, Stan!
Hello, Ollie!
Hello, Stan!
Hello, Ollie!
Hello, Stan!
Hello, Ollie!
Hello, Stan!
Bye Bye, Ollie...
Goodbye, Stan...
```

The Method Class

The Method class creates method objects from REXX source code. It is a subclass of the Object class.

Methods the Method class defines:

NEW (Class method. Overrides Object class method.)
NEWFILE (Class method)
SETGUARDED
SETPRIVATE
SETPROTECTED
SETSECURITYMANAGER
SETUNGUARDED
SOURCE

Methods inherited from the Object class:

Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Method class also has available class methods that its metaclass, the Class class, defines.

NEW (Class Method)

```
►►──NEW(name,source)──────────────────────────────────────────►◄
```

Returns a new instance of method class, which is an executable representation of the code contained in the *source*. The *name* is a string. The *source* can be a single string or an array of strings containing individual method lines.

NEWFILE (Class Method)

```
>>──NEWFILE(filename)────────────────────────────────────────────────────><
```

Returns a new instance of method class, which is an executable representation of the code contained in the file *filename*. The *filename* is a string.

For an example of the use of this method, see the code example 'Server implements Security Manager' in Chapter 15.

SETGUARDED

```
>>──SETGUARDED───────────────────────────────────────────────────────────><
```

Reverses any previous SETUNGUARDED messages, restoring the receiver to the default guarded status. If the receiver is already guarded, a SETGUARDED message has no effect.

SETPRIVATE

```
>>──SETPRIVATE───────────────────────────────────────────────────────────><
```

Specifies that a method is a private method. Only a message that an object sends to itself can run a private method. If a method object does not receive a SETPRIVATE message, the method is a public method. (Any object can send a message to run a public method. See "Public and Private Methods" on page 103 for details.)

SETPROTECTED

```
>>──SETPROTECTED─────────────────────────────────────────────────────────><
```

Specifies thata method is a protected method. If a method object does not receive a SETPROTECTED message, the method is not protected. (See "Chapter 15. The Security Manager" on page 457 for details.)

SETSECURITYMANAGER

```
>>──SETSECURITYMANAGER──┬────────────────────────────┬───────────────────><
                        └─(security_manager_object)─┘
```

Replaces the existing security manager with the specified *security_manager_object*. If *security_manager_object* is omitted, any existing security manager is removed.

SETUNGUARDED

```
►►──SETUNGUARDED──────────────────────────────────────────────────►◄
```

Lets an object run a method even when another method is active on the same object. If a method object does not receive a SETUNGUARDED message, it requires exclusive use of its object variable pool. A method can be active for an object only when no other method requiring exclusive access to the object's variable pool is active in the same object. This restriction does not apply if an object sends itself a message to run a method and it already has exclusive use of the same object variable pool. In this case, the method runs immediately and has exclusive use of its object variable pool, regardless of whether it received a SETUNGUARDED message.

SOURCE

```
►►──SOURCE────────────────────────────────────────────────────────►◄
```

Returns the method source code as a single-index array of source lines. If the source code is not available, SOURCE returns an array of zero items.

The Monitor Class

The Monitor class forwards messages to a destination object. It is a subclass of the Object class.

Methods the Monitor class defines:

CURRENT
DESTINATION
INIT (Overrides Object class method)
UNKNOWN

Methods inherited from the Object class:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME

HASMETHOD
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Monitor class also has available class methods that its metaclass, the Class class, defines.

CURRENT

```
►►──CURRENT──────────────────────────────────────────────────────────►◄
```

Returns the current destination object.

DESTINATION

```
►►──DESTINATION──┬───────────────┬───────────────────────────────────►◄
                 └─(destination)─┘
```

Returns a new destination object. If you specify *destination*, this becomes the new destination for any forwarded messages. If you omit *destination*, the previous destination object becomes the new destination for any forwarded messages.

INIT

```
►►──INIT──┬───────────────┬──────────────────────────────────────────►◄
          └─(destination)─┘
```

Initializes the newly created monitor object.

UNKNOWN

```
►►──UNKNOWN(messagename,messageargs)─────────────────────────────────►◄
```

Reissues or forwards to the current monitor destination all unknown messages sent to a monitor object. For additional information, see "Defining an UNKNOWN Method" on page 102.

Examples

```
.local~setentry('output',.monitor~new(.stream~new('my.new')~~command('open nobuffer')))

/* The following sets the destination */
previous_destination=.output~destination(.stream~new('my.out')~~command('open write'))
/* The following resets the destination */
.output~destination

.output~destination(.STDOUT)
current_output_destination_stream_object=.output~current
```

The Object Class

The Object class is the root of the class hierarchy. The instance methods of the Object class are, therefore, available on all objects.

Methods the Object class defines:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Object class also has available class methods that its metaclass, the Class class, defines.

NEW (Class Method)

```
►►──NEW──────────────────────────────────────────────────────────────►◄
```

Returns a new instance of the receiver class.

Operator Methods

```
►►──comparison_operator(argument)──────────────────────────────────────►◄
```

Note: The *argument* is optional for the == operator.

Returns 1 (true) or 0 (false), the result of performing a specified comparison operation. If you specify the == operator and omit *argument*, a string representation is returned representing a hash value for Set, Bag, Table, Relation, and Directory.

For the Object class, the arguments must match the receiver object. If they do not match the receiver object, you can define subclasses of the Object class to match the arguments.

The comparison operators you can use in a message are:

=, == True if the terms are the same object.

\=, ><, <>, \==
True if the terms are not the same object (inverse of =).

CLASS

```
►►──CLASS──────────────────────────────────────────────────────────────►◄
```

Returns the class object that received the message that created the object.

COPY

```
►►──COPY───────────────────────────────────────────────────────────────►◄
```

Returns a copy of the receiver object. The copied object has the same methods as the receiver object and an equivalent set of object variables, with the same values.

Example:

```
myarray=.array~of('N','S','E','W')
directions=myarray~copy   /* Copies array myarray to array directions */
```

DEFAULTNAME

```
►►──DEFAULTNAME────────────────────────────────────────────────────────►◄
```

Returns a short human-readable string representation of the object. The exact form of this representation depends on the object and might not alone be sufficient to reconstruct the object. All objects must be able to produce a short string representation of themselves in this way, even if the object does not have a string value. See "Required String Values" on page 105 for more information. The DEFAULTNAME method of the Object class returns a string that identifies the class of the object, for example, an Array or a Directory. See also "OBJECTNAME" and "STRING" on page 185. See "OBJECTNAME=" for an example using DEFAULTNAME.

HASMETHOD

```
►►──HASMETHOD(methodname)──────────────────────────────────────────►◄
```

Returns 1 (true) if the receiver object has a method named *methodname* (translated to uppercase) or if the target method is a private method. Otherwise, it returns `0` (false).

Note: If you call the *methodname* method although it is private, you receive error 97 `Object method not found` although HASMETHOD returns 1 (true).

INIT

```
►►──INIT───────────────────────────────────────────────────────────►◄
```

Performs any required object initialization. Subclasses of the Object class can override this method.

OBJECTNAME

```
►►──OBJECTNAME─────────────────────────────────────────────────────►◄
```

Returns the receiver object's name that the OBJECTNAME= method sets. If the receiver object does not have a name, this method returns the result of the DEFAULTNAME method. See "Required String Values" on page 105 for more information. See the OBJECTNAME= method for an example using OBJECTNAME.

OBJECTNAME=

```
►►──OBJECTNAME=(newname)───────────────────────────────────────────►◄
```

Sets the receiver object’s name to the string *newname*.

Example:

```
points=.array~of('N','S','E','W')
say points~objectname        /* (no change yet) Says: "an Array"    */
points~objectname=('compass') /* Changes obj name POINTS to "compass"*/
say points~objectname        /* Shows new obj name. Says: "compass" */
say points~defaultname       /* Default is still available.         */
                             /* Says "an Array"                     */
say points                   /* Says string representation of       */
                             /* points "compass"                    */
say points[3]                /* Says: "E"Points is still an array   */
                             /* of 4 items                          */
```

REQUEST

```
►►──REQUEST(classid)──────────────────────────────────────────────►◄
```

Returns an object of the *classid* class, or the NIL object if the request cannot be satisfied.

This method first compares the identity of the object's class (see the ID method of the Class class in “ID” on page 164) to *classid*. If they are the same, the receiver object is returned as the result. Otherwise, REQUEST tries to obtain and return an object satisfying *classid* by sending the receiver object the conversion message `MAKE` with the string *classid* appended (converted to uppercase). For example, a `REQUEST('string')` message causes a MAKESTRING message to be sent. If the object does not have the required conversion method, REQUEST returns the NIL object.

The conversion methods cause objects to produce different representations of themselves. The presence or absence of a conversion method defines an object's capability to produce the corresponding representations. For example, lists can represent themselves as arrays, because they have a MAKEARRAY method, but they cannot represent themselves as directories, because they do not have a MAKEDIRECTORY method. Any conversion method must return an object of the requested class. For example, MAKEARRAY must return an array. The language processor uses the MAKESTRING method to obtain string values in certain contexts; see “Required String Values” on page 105.

RUN

```
>>--RUN(method--+---------------------------------+--)-------------------><
                |                .-------------.  |
                |                V             |  |
                +-,Individual-------+-----------+-+
                |                   '-,argument-' |
                '-,Array,argument-----------------'
```

Runs the method object *method* (see "The Method Class" on page 175). The *method* has access to the object variables of the receiver object, as if the receiver object had defined the method by using SETMETHOD.

If you specify the Individual or Array option, any remaining *argument*s are arguments for the method. (You need to specify only the first letter; the language processor ignores all characters following it.)

Individual
Passes any remaining arguments to the method as arguments in the order you specify them.

Array Requires *argument*, which is an array object. (See "The Array Class" on page 118.) The language processor passes the member items of the array to the method as arguments. The first argument is at index 1, the second argument at index 2, and so on. If you omitted any indexes when creating the array, the language processor omits their corresponding arguments when passing the arguments.

If you specify neither Individual nor Array, the method runs without arguments.

The *method* argument can be a string containing a method source line instead of a method object. Alternatively, you can pass an array of strings containing individual method lines. In either case, RUN creates an equivalent method object.

Notes:
1. The RUN method is a private method. See the SETPRIVATE method in "SETPRIVATE" on page 176 for details.
2. The RUN method is a protected method.

SETMETHOD

```
>>--SETMETHOD(methodname--+---------+--)---------------------------------><
                          '-,method-'
```

Adds a method to the receiver object's collection of object methods. The *methodname* is the name of the new method. (The language processor translates this name to uppercase.) If you previously defined a method with the same name using SETMETHOD, the new method replaces the earlier one. If you omit *method*, SETMETHOD makes the method name *methodname* unavailable for the receiver object. In this case, sending a message of that name to the receiver object runs the UNKNOWN method (if any).

The *method* can be a string containing a method source line instead of a method object. Or it can be an array of strings containing individual method lines. In either case, SETMETHOD creates an equivalent method object.

Notes:

1. The SETMETHOD method is a private method. See the SETPRIVATE method in "SETPRIVATE" on page 176 for details.
2. The SETMETHOD method is a protected method.

START

```
                         ┌──────────────────┐
                         ▼                  │
►►──START(messagename────┴──┬────────────┬──┴──)──────────────────────────►◄
                            └─,argument──┘
```

Returns a message object (see "The Message Class" on page 170) and sends it a START message to start concurrent processing. The object receiving the message *messagename* processes this message concurrently with the sender's continued processing.

The *messagename* can be a string or an array. If *messagename* is an array object, its first item is the name of the message and its second item is a class object to use as the starting point for the method search. For more information, see Table 1 on page 108.

The language processor passes any *argument*s to the receiver as arguments for *messagename* in the order you specify them.

When the receiver object has finished processing the message, the message object retains its result and holds it until the sender requests it by sending a RESULT message. For further details, see "START" on page 173.

Example:

```
world=.object~new
msg=world~start('HELLO')               /* same as next line     */
msg=.message~new(world,'HELLO')~~start  /* same as previous line */
```

STRING

```
>>--STRING------------------------------------------------------------><
```

Returns a human-readable string representation of the object. The exact form of this representation depends on the object and might not alone be sufficient to reconstruct the object. All objects must be able to produce a string representation of themselves in this way.

The object's string representation is obtained from the OBJECTNAME method (which can in turn use the DEFAULTNAME method). See also the OBJECTNAME method ("OBJECTNAME" on page 181) and the DEFAULTNAME method ("DEFAULTNAME" on page 180).

The distinction between this method, the MAKESTRING method (which obtains string values—see "MAKESTRING" on page 227) and the REQUEST method (see "REQUEST" on page 182) is important. All objects have a STRING method, which returns a *string representation* (human-readable form) of the object. This form is useful in tracing and debugging. Only those objects that have information with a meaningful string form have a MAKESTRING method to return this value. For example, directory objects have a readable string representation (a Directory), but no string value, and, therefore, no MAKESTRING method.

Of the classes that REXX provides, only the String class has a MAKESTRING method. Any subclasses of the String class inherit this method by default, so these subclasses also have string values. Any other class can also provide a string value by defining a MAKESTRING method.

UNSETMETHOD

```
>>--UNSETMETHOD(methodname)-------------------------------------------><
```

Cancels the effect of all previous SETMETHODs for method *methodname*. It also removes any method *methodname* introduced with ENHANCED when the object was created. If the object has received no SETMETHOD method, no action is taken.

Notes:

1. The UNSETMETHOD method is a private method. See the SETPRIVATE method in "SETPRIVATE" on page 176 for details.
2. The UNSETMETHOD method is a protected method.

The Stem Class

A *stem* object is a collection with unique indexes that are character strings.

Stems are automatically created whenever a REXX stem variable or REXX compound variable is used. For example:

```
a.1 = 2
```

creates a new stem collection and assigns it to the REXX variable `A.`; it also assigns the value `2` to entry `1` in the collection.

In addition to the items assigned to the collection indexes, a stem also has a default value that is used for all uninitialized indexes of the collection. You can assign a default value to the stem with the []= method and retrieve the default value with the [] method.

In addition to the methods defined in the following, the Stem class removes the methods =, ==, \=, \==, <>, and >< using the DEFINE method.

Methods the Stem class defines:

NEW (Class method. Overrides Object class method.)
[]
[]=
MAKEARRAY
REQUEST (Overrides Object class method)
UNKNOWN

Methods inherited from the Object class:

Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Stem class also has available class methods that its metaclass, the Class class, defines.

NEW (Class Method)

```
►►──NEW──┬────────┬──────────────────────────────────────────────►◄
         └─(name)─┘
```

Returns a new stem object. If you specify a string *name*, this value is used to create the derived name of compound variables. The default stem name is a null string.

[]

```
        ┌─,──────────┐
        ▼            │
►►──[───┬───────┬────┴──]────────────────────────────────────────►◄
        └─index─┘
```

Returns the item associated with the specified *index*es. Each *index* is an expression; use commas to separate the expressions. The language processor concatenates the *index* expression string values, separating them with a period (.), to create a derived index. A null string ("") is used for any omitted expressions. The resulting string references the stem item. If the stem has no item associated with the specified final *index*, the stem default value is returned. If a default value has not been set, the stem name concatenated with the final index string is returned.

If you do not specify *index*, the stem default value is returned. If no default value has been assigned, the stem name is returned.

Note: You cannot use the [] method in a DROP or PROCEDURE instruction or in a parsing template.

[]=

```
        ┌─,─────────┐
        ▼           │
►►──[───┬───────┬───┴──]=value───────────────────────────────────►◄
        └─index─┘
```

Makes the value a member item of the stem collection and associates it with the specified index. If you specify no *index* expressions, a new default stem value is assigned. Assigning a new default value will re-initialize the stem and remove all existing assigned indexes.

MAKEARRAY

```
>>──MAKEARRAY──────────────────────────────────────────────────────────><
```

Returns an array of all stem indexes that currently have an associated value. The items appear in the array in an unspecified order. (The program should not rely on any order.)

REQUEST

```
>>──REQUEST(classid)───────────────────────────────────────────────────><
```

Returns the result of the Stem class MAKEARRAY method, if the requested class is ARRAY. For all other classes, REQUEST forwards the message to the default value of the stem and returns this result. This method requests conversion to a specific class. All conversion requests except ARRAY are forwarded to the current stem default value.

UNKNOWN

```
>>──UNKNOWN─(messagename,messageargs)──────────────────────────────────><
```

Reissues or forwards to the current stem default value all unknown messages sent to a stem collection. For additional information, see "Defining an UNKNOWN Method" on page 102.

The Stream Class

A *stream* object allows external communication from REXX. (See "Chapter 16. Input and Output Streams" on page 463 for a discussion of REXX input and output.)

The Stream class is a subclass of the Object class.

Methods the Stream class defines:

ARRAYIN
ARRAYOUT
CHARIN
CHAROUT
CHARS
CLOSE
COMMAND

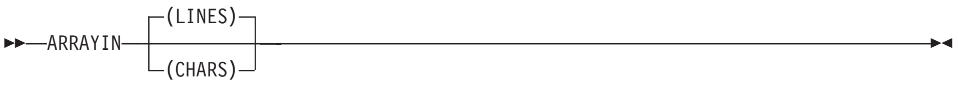
DESCRIPTION
FLUSH
INIT (Overrides Object class method)
LINEIN
LINEOUT
LINES
MAKEARRAY
OPEN
POSITION
QUALIFY
QUERY
SEEK
STATE
SUPPLIER

Methods inherited from the Object class:

NEW (Class method)
Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME>
HASMETHOD
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Stream class also has available class methods that its metaclass, the Class class, defines.

ARRAYIN



Returns a fixed array that contains the data of the stream in line or character format, starting from the current read position. The line format is the default.

If you have used the CHARIN method, the first line can be a partial line.

ARRAYOUT

```
                         ┌─,──LINES─┐
►►──ARRAYOUT(array──┼──────────┼──)────────────────────────────────►◄
                         └─,──CHARS─┘
```

Returns a stream object that contains the data from *array*.

CHARIN

```
►►──CHARIN──┬─────────────────────────────┬────────────────────────►◄
             └─(──┬───────┬──┬─────────┬──)─┘
                  └─start─┘  └─,length─┘
```

Returns a string of up to *length* characters from the character input stream receiving the message. The language processor advances the read pointer. If you omit *length*, it defaults to 1. If you specify *start*, this positions the read pointer before reading. If the stream is not already open, the language processor tries to open the stream for reading and writing. If that fails, the language processor opens the stream for read only. (See "CHARIN (Character Input)" on page 310 for information about the CHARIN built-in function.)

CHAROUT

```
►►──CHAROUT──┬──────────────────────────────┬──────────────────────►◄
              └─(──┬────────┬──┬────────┬──)─┘
                   └─string─┘  └─,start─┘
```

Returns the count of characters remaining after trying to write *string* to the character output stream receiving the message. The language processor advances the write pointer.

The *string* can be the null string. In this case, CHAROUT writes no characters to the stream and returns `0`. If you omit *string*, CHAROUT writes no characters to the stream and returns `0`. The language processor closes the stream.

If you specify *start*, this positions the write pointer before writing. If the stream is not already open, the language processor tries to open the stream for reading and writing. If that fails, the language processor opens the stream for write only. (See "CHAROUT (Character Output)" on page 311 for information about the CHAROUT built-in function.)

CHARS

```
►►──CHARS──────────────────────────────────────────────────────────►◄
```

Returns the total number of characters remaining in the character input stream receiving the message. The default input stream is STDIN. The count includes any line separator characters, if these are defined for the stream. In the case of persistent streams, it is the count of characters from the current read position. (See "Chapter 16. Input and Output Streams" on page 463 for a discussion of REXX input and output.) The total number of characters remaining cannot be determined for some streams (for example, STDIN). For these streams. the CHARS method returns `1` to indicate that data is present, or `0` if no data is present. For Windows devices, CHARS always returns 1. (See "CHARS (Characters Remaining)" on page 312 for information about the CHARS built-in function.)

CLOSE

```
►►──CLOSE──────────────────────────────────────────────────────────►◄
```

Closes the stream that receives the message. CLOSE returns `READY:` if closing the stream is successful, or an appropriate error message. If you have tried to close an unopened file, then the CLOSE method returns a null string ("").

COMMAND

```
►►──COMMAND(stream_command)────────────────────────────────────────►◄
```

Returns a string after performing the specified *stream_command*. The returned string depends on the *stream_command* performed and can be the null string. The following *stream_command*s:

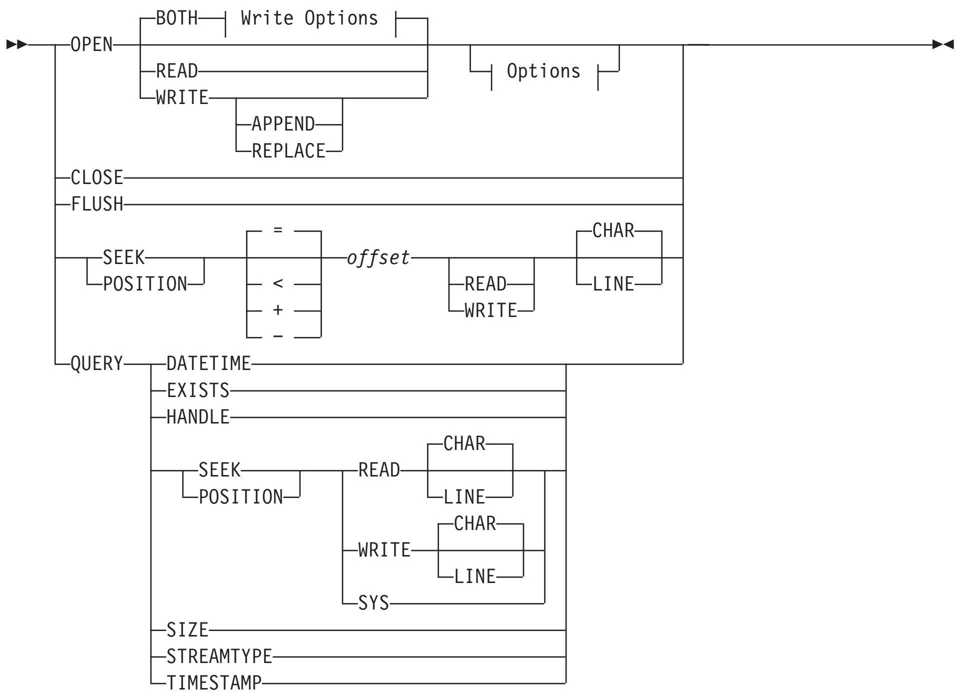
- Open a stream for reading, writing, or both
- Close a stream at the end of an operation
- Move the line read or write position within a persistent stream (for example, a file)
- Get information about a stream

If the method is unsuccessful, it returns an error message string in the same form that the DESCRIPTION method uses.

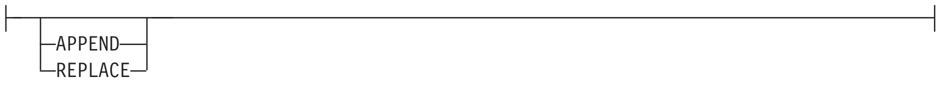
For most error conditions, the additional information is in the form of a numeric return code. This return code is the value of `ERRNO` that is set whenever one of the file system primitives returns with a `-1`.

Command Strings

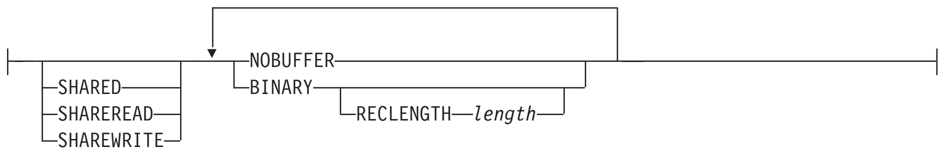
The argument *stream_command* can be any expression that the language processor evaluates to a command string that corresponds to the following diagram:



Write Options:



Options:



OPEN Opens the stream object receiving the message and returns `READY:`. (If unsuccessful, the previous information about return codes applies.) The default for OPEN is to open the stream for both reading and writing data, for example: `'OPEN BOTH'`. To specify that the *stream_name* receiving the message can be only read or written to, add READ or WRITE, to the command string.

The following is a description of the options for OPEN:

READD Opens the stream only for reading.

WRITE Opens the stream only for writing.

BOTH Opens the stream for both reading and writing. (This is the default.) The language processor maintains separate read and write pointers.

APPEND Positions the write pointer at the end of the stream. The write pointer cannot be moved anywhere within the extent of the file as it existed when the file was opened.

REPLACE Sets the write pointer to the beginning of the stream and truncates the file. In other words, this option deletes all data that was in the stream when opened.

SHARED Enables another process to work with the stream in a shared mode. This mode must be compatible with the shared mode (SHARED, SHAREREAD, or SHAREWRITE) used by the process that opened the stream.

SHAREREAD Enables another process to read the stream in a shared mode.

SHAREWRITE

Enables another process to write the stream in a shared mode.

NOBUFFER Turns off buffering of the stream. All data written to the stream is flushed immediately to the operating system for writing. This option can have a severe impact on output performance. Use it only when data integrity is a concern, or to force interleaved output to a stream to appear in the exact order in which it was written.

BINARY
Opens the stream in binary mode. This means that line end characters are ignored; they are treated like any other byte of data. This is intended to force file operations that are compatible with other REXX language processors that run on record-based systems, or to process binary data using the line operations.

Note: Specifying the BINARY option for a stream that does not exist but is opened for writing also requires the RECLENGTH option to be specified. Omitting the RECLENGTH option in this case raises an error condition.

RECLENGTH *length*
Allows the specification of an exact length for each line in a stream. This allows line operations on binary-mode streams to operate on individual fixed-length records. Without this option, line operations on binary-mode files operate on the entire file (for example, as if you specified the `RECLENGTH` option with a length equal to that of the file). The *length* must be 1 or greater.

Examples:

```
stream_name~Command('open')
stream_name~Command('open write')
stream_name~Command('open read')
stream_name~Command('open read shared')
```

CLOSE
closes the stream object receiving the message. The COMMAND method with the CLOSE option returns `READY:` if the receiving stream object is successfully closed or an appropriate error message otherwise. If an attempt to close an unopened file occurs, then the COMMAND method with the CLOSE option returns a null string ("").

FLUSH	forces any data currently buffered for writing to be written to this stream.
SEEK *offset*	sets the read or write position to a given number (*offset*) within a persistent stream. If the stream is open for both reading and writing and you do not specify READ or WRITE, both the read and write positions are set.

Note: See "Chapter 16. Input and Output Streams" on page 463 for a discussion of read and write positions in a persistent stream.

To use this command, you must first open the receiving stream object (with the OPEN stream command described previously or implicitly with an input or output operation). One of the following characters can precede the *offset* number.

=	explicitly specifies the *offset* from the beginning of the stream. This is the default if you supply no prefix. For example, an *offset* of 1 with the LINE option means the beginning of the stream.
<	specifies *offset* from the end of the stream.
+	specifies *offset* forward from the current read or write position.
-	specifies *offset* backward from the current read or write position.

The COMMAND method with the SEEK option returns the new position in the stream if the read or write position is successfully located, or an appropriate error message.

The following is a description of the options for SEEK:

READ	specifies that this command sets the read position.
WRITE	specifies that this command sets the write position.
CHAR	specifies the positioning in terms of characters. This is the default.
LINE	specifies the positioning in terms of lines. For non-binary streams, this is potentially an operation that can take a long time to complete because, in most cases, the file must be scanned from the top to count the line-end

characters. However, for binary streams with a specified record length, the new resulting line number is simply multiplied by the record length before character positioning. See "Line versus Character Positioning" on page 469 for a detailed discussion of this issue.

Note: If you do line positioning in a file open only for writing, you receive an error message.

Examples:

```
stream_name~Command('seek =2 read')
stream_name~Command('seek +15 read')
stream_name~Command('seek -7 write line')
fromend  = 125
stream_name~Command('seek <'fromend read)
```

POSITION is a synonym for SEEK.

Used with these *stream_command*s, the COMMAND method returns specific information about a stream. Except for QUERY HANDLE and QUERY POSITION, the language processor returns the query information even if the stream is not open. The language processor returns the null string for nonexistent streams.

QUERY DATETIME
Returns the date and time stamps of a stream in US format. For example:

```
stream_name~Command('query datetime')
```

A sample output might be:

```
11-12-95 03:29:12
```

QUERY EXISTS
Returns the full path specification of the stream object receiving the message, if it exists, or a null string. For example:

```
stream_name~Command('query exists')
```

A sample output might be:

```
c:\data\file.txt
```

QUERY HANDLE
Returns the handle associated with the open stream that is the receiving stream object. For example:

```
stream_name~Command('query handle')
```

A sample output might be:

```
3
```

QUERY POSITION
Returns the current read or write position for the receiving stream object, as qualified by the following options:

READ Returns the current read position.

WRITE Returns the current write position.

Note: If the stream is open for both reading and writing, this returns the read position by default. Otherwise, this returns the appropriate position by default.

CHAR Returns the position in terms of characters. This is the default.

LINE Returns the position in terms of lines. For non-binary streams, this operation can take a long time to complete. This is because the language processor starts tracking the current line number if not already doing so, and, thus, might require a scan of the stream from the top to count the line-end characters. See "Line versus Character Positioning" on page 469 for a detailed discussion of this issue. For example:

```
stream_name~Command('query position write')
```

A sample output might be:

```
247
```

SYS Returns the operating system stream position in terms of characters.

QUERY SEEK
Is a synonym for QUERY POSITION.

QUERY SIZE
Returns the size in bytes of a persistent stream that is the receiving stream object. For example:

```
stream_name~Command('query size')
```

A sample output might be:

```
1305
```

QUERY STREAMTYPE
Returns a string indicating whether the receiving stream object is `PERSISTENT`, `TRANSIENT`, or `UNKNOWN`.

QUERY TIMESTAMP
Returns the date and time stamps of the receiving stream object in an international format. This is the preferred method of getting date and time because it provides the full 4-digit year. For example:

```
stream_name~Command('query timestamp')
```

A sample output might be:

```
1995-11-12 03:29:12
```

DESCRIPTION

```
►►──DESCRIPTION───────────────────────────────────────────────────────►◄
```

Returns any descriptive string associated with the current state of the stream or the NIL object if no descriptive string is available. The DESCRIPTION method is identical with the STATE method except that the string that DESCRIPTION returns is followed by a colon and, if available, additional information about ERROR or NOTREADY states. (The STATE method in "STATE" on page 205 describes these states.)

FLUSH

```
►►──FLUSH─────────────────────────────────────────────────────────────►◄
```

Returns `READY:`. It forces any data currently buffered for writing to be written to the stream receiving the message.

INIT

```
►►──INIT(name)────────────────────────────────────────────────────────►◄
```

Initializes a stream object for a stream named *name*, but does not open the stream. See "Initialization" on page 104 for more information.

LINEIN

```
►►──LINEIN──┬─────────────────────────┬───────────────────────────────►◄
            └─(──┬──────┬──┬────────┬─)─┘
                 └─line─┘  └─,count─┘
```

Returns the next *count* lines. The count must be `0` or `1`. The language processor advances the read pointer. If you omit *count*, it defaults to `1`. If you specify *line*, this positions the read pointer before reading. If the stream is not already open, the language processor tries to open the stream for reading and writing. If that fails, the language processor opens the stream for read-only. (See “LINEIN (Line Input)” on page 328 for information about the LINEIN built-in function.)

LINEOUT

```
>>──LINEOUT──┬────────────────────────────┬──────────────────────────────────><
             └─(──┬────────┬──┬───────┬──)─┘
                  └─string─┘  └─,line─┘
```

Returns 0 if successful in writing *string* to the character output stream receiving the message or 1 if an error occurs while writing the line. The language processor advances the write pointer. If you omit *string*, the language processor closes the stream. If you specify *line*, this positions the write pointer before writing. If the stream is not already open, the language processor tries to open the stream for reading and writing. If that fails, the language processor opens the stream for write-only. (See “LINEOUT (Line Output)” on page 330 for information about the LINEOUT built-in function.)

LINES

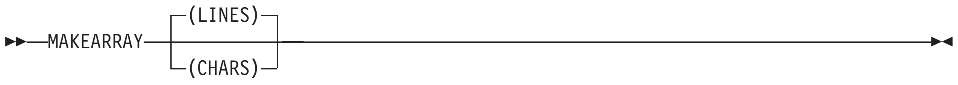
```
>>──LINES────────────────────────────────────────────────────────────────────><
```

Returns the number of completed lines that remain in the character input stream receiving the message. If the stream has already been read with CHARIN, this can include an initial partial line. For persistent streams the count starts at the current read position. In effect, LINES reports whether a read action of CHARIN (see “CHARIN” on page 190) or LINEIN (see “LINEIN” on page 198) will succeed. (For an explanation of input and output, see “Chapter 16. Input and Output Streams” on page 463.)

For QUEUE, LINES returns the actual number of lines. (See “LINES (Lines Remaining)” on page 331 for information about the LINES built-in function.)

Note: The CHARS method returns the number of characters in a persistent stream or the presence of data in a transient stream. The LINES method determines the actual number of lines by scanning the stream starting at the current position and counting the lines. For large streams, this can be a time-consuming operation. Therefore, avoid the use of the LINES method in the condition of a loop reading a stream. It is recommended that you use the CHARS method (see “CHARS” on page 191) or the LINES built-in function for this purpose.

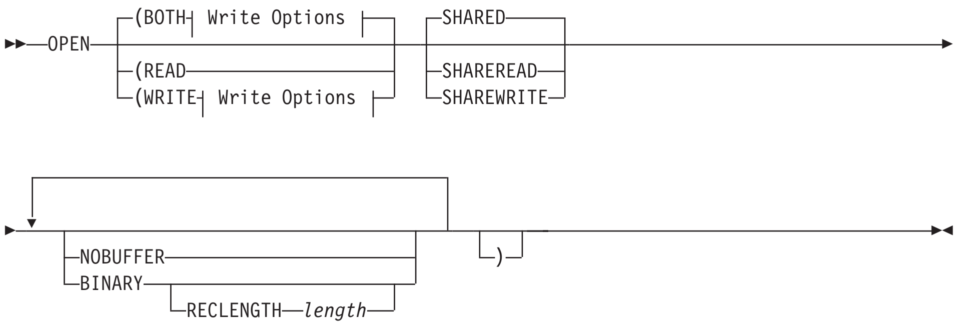
MAKEARRAY



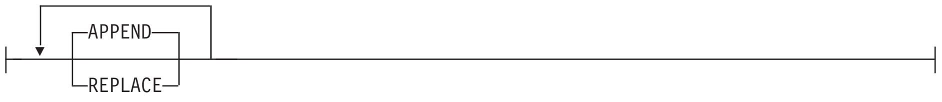
Returns a fixed array that contains the data of the stream in line or character format, starting from the current read position. The line format is the default.

If you have used the CHARIN method, the first line can be a partial line.

OPEN



Write Options:



Opens the stream to which you send the message and returns `READY:`. If the method is unsuccessful, it returns an error message string in the same form that the DESCRIPTION method uses.

For most error conditions, the additional information is in the form of a numeric return code. This return code is the value of `ERRNO`, which is set whenever one of the file system primitives returns with a `-1`.

By default, OPEN opens the stream for both reading and writing data, for example: `'OPEN BOTH'`. To specify that the stream receiving the message can be only read or only written to, specify READ or WRITE.

The options for the OPEN method are:

READ
Opens the stream only for reading.

WRITE
Opens the stream only for writing.

BOTH
Opens the stream for both reading and writing. (This is the default.) The language processor maintains separate read and write pointers.

APPEND
Positions the write pointer at the end of the stream. (This is the default.) The write pointer cannot be moved anywhere within the extent of the file as it existed when the file was opened.

REPLACE
Sets the write pointer to the beginning of the stream and truncates the file. In other words, this option deletes all data that was in the stream when opened.

SHARED
Enables another process to work with the stream in a shared mode. (This is the default.) This mode must be compatible with the shared mode (SHARED, SHAREREAD, or SHAREWRITE) used by the process that opened the stream.

SHAREREAD
Enables another process to read the stream in a shared mode.

SHAREWRITE
Enables another process to write the stream in a shared mode.

NOBUFFER
Turns off buffering of the stream. All data written to the stream is flushed immediately to the operating system for writing. This option can have a severe impact on output performance. Use it only when data integrity is a concern, or to force interleaved output to a stream to appear in the exact order in which it was written.

BINARY
Opens the stream in binary mode. This means that line-end characters are ignored; they are treated like any other byte of data. This is intended to force file operations that are compatible with other REXX language processors that run on record-based systems, or to process binary data using the line operations.

Note: Specifying the BINARY option for a stream that does not exist but is opened for writing also requires the RECLENGTH option to be specified. Omitting the RECLENGTH option in this case raises an error condition.

RECLENGTH *length*
Allows the specification of an exact length for each line in a

stream. This allows line operations on binary-mode streams to operate on individual fixed-length records. Without this option, line operations on binary-mode files operate on the entire file (for example, as if you specified the RECLENGTH option with a length equal to that of the file). The *length* must be 1 or greater.

Examples:

```
stream_name~OPEN
stream_name~OPEN('write')
stream_name~OPEN('read')
```

POSITION

```
                 ┌─ = ─┐                      ┌─CHAR─┐
►►──POSITION(────┼─────┼──offset───┬─READ──┬──┼──────┼──)────────────────────────────►◄
                 ├─ < ─┤           └─WRITE─┘  └─LINE─┘
                 ├─ + ─┤
                 └─ − ─┘
```

POSITION is a synonym for SEEK. (See “SEEK” on page 204.)

QUALIFY

```
►►──QUALIFY─────────────────────────────────────────────────────────────────────►◄
```

Returns the stream's fully qualified name. The stream need not be open.

QUERY

```
►►──QUERY(──┬─DATETIME──────────────────────────────────┬──)────────────────────►◄
            ├─EXISTS────────────────────────────────────┤
            ├─HANDLE────────────────────────────────────┤
            │                          ┌─CHAR─┐         │
            ├──┬─SEEK─────┬──┬─READ────┼──────┼───┬─────┤
            │  └─POSITION─┘  │         └─LINE─┘   │     │
            │                │          ┌─CHAR─┐  │     │
            │                ├─WRITE────┼──────┼──┤     │
            │                │          └─LINE─┘  │     │
            │                └─SYS────────────────┘     │
            ├─SIZE──────────────────────────────────────┤
            ├─STREAMTYPE────────────────────────────────┤
            └─TIMESTAMP─────────────────────────────────┘
```

Used with these options, the QUERY method returns specific information about a stream. Except for QUERY HANDLE and QUERY POSITION, the language processor returns the query information even if the stream is not open. The language processor returns the null string for nonexistent streams.

DATETIME
: returns the date and time stamps of the receiving stream object in US format. For example:

```
stream_name~query('datetime')
```

A sample output might be:

```
11-12-98 03:29:12
```

EXISTS
: returns the full path specification of the receiving stream object, if it exists, or a null string. For example:

```
stream_name~query('exists')
```

A sample output might be:

```
c:\data\file.txt
```

HANDLE
: returns the handle associated with the open stream that is the receiving stream object. For example:

```
stream_name~query('handle')
```

A sample output might be:

```
3
```

POSITION
: returns the current read or write position for the receiving stream object, as qualified by the following options:

READ	returns the current read position.
WRITE	returns the current write position. **Note:** If the stream is open for both reading and writing, this returns the read position by default. Otherwise, this returns the appropriate position by default.
CHAR	returns the position in terms of characters. This is the default.
LINE	returns the position in terms of lines. For non-binary streams, this operation can take a long time to complete. This is because the language processor starts tracking the current line number if not already doing so, and, thus, might require a scan of the stream from the top to count the line-end characters. See

"Line versus Character Positioning" on page 469 for a detailed discussion of this issue. For example:

```
stream_name~query('position write')
```

A sample output might be:

```
247
```

SYS returns the operating system stream position in terms of characters.

SIZE returns the size, in bytes, of a persistent stream that is the receiving stream object. For example:

```
stream_name~query('size')
```

A sample output might be:

```
1305
```

STREAMTYPE
returns a string indicating whether the receiving stream object is PERSISTENT, TRANSIENT, or UNKNOWN.

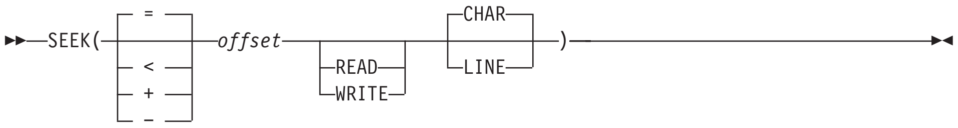
TIMESTAMP
returns the date and time stamps of the receiving stream object in an international format. This is the preferred method of getting the date and time because it provides the full 4-digit year. For example:

```
stream_name~query('timestamp')
```

A sample output might be:

```
1998-11-12 03:29:12
```

SEEK



Sets the read or write position to a given number (*offset*) within a persistent stream. If the stream is open for both reading and writing and you do not specify READ or WRITE, both the read and write positions are set.

Note: See "Chapter 16. Input and Output Streams" on page 463 for a discussion of read and write positions in a persistent stream.

To use this method, you must first open the receiving stream object (with the OPEN method described previously or implicitly with an input or output operation). One of the following characters can precede the *offset* number:

= Explicitly specifies the *offset* from the beginning of the stream. This is the default if you supply no prefix. For example, an *offset* of 1 means the beginning of the stream.

< Specifies *offset* from the end of the stream.

+ Specifies *offset* forward from the current read or write position.

- Specifies *offset* backward from the current read or write position.

The SEEK method returns the new position in the stream if the read or write position is successfully located, or an appropriate error message.

The following is a description of the options for SEEK:

READ specifies that the read position be set.

WRITE specifies that the write position be set.

CHAR specifies that positioning be done in terms of characters. This is the default.

LINE specifies that the positioning be done in terms of lines. For non-binary streams, this is potentially an operation that can take a long time to complete because, in most cases, the file must be scanned from the top to count the line-end characters. However, for binary streams with a specified record length, the new resulting line number is simply multiplied by the record length before character positioning. See “Line versus Character Positioning” on page 469 for a detailed discussion of this issue.

Note: If you do line positioning in a file open only for writing, you receive an error message.

Examples:

```
stream_name~seek('=2 read')
stream_name~seek('+15 read')
stream_name~seek('-7 write line')
fromend  = 125
stream_name~seek('<'fromend read)
```

STATE

```
►►──STATE──────────────────────────────────────────────────────────►◄
```

Returns a string that indicates the current state of the specified stream.

The returned strings 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 472. You might be able to obtain additional information about the error with the DESCRIPTION method or by calling the STREAM function with a request for the description.
NOTREADY	The stream is known to be in such a state that the usual input or output operations attempted upon would raise the NOTREADY condition. (See “Errors during Input and Output” on page 472.) For example, a simple input stream can have a defined length. An attempt to read that stream (with CHARIN or LINEIN, perhaps) beyond that limit can make the stream unavailable until the stream has been closed (for example, with LINEOUT(*name*)) and then reopened.
READY	The stream is known to be in such a state that the usual input or output operations might be attempted. This is the usual state for a stream, although it does not guarantee that any particular operation will succeed.
UNKNOWN	The state of the stream is unknown. This generally means that the stream is closed or has not yet been opened.

SUPPLIER

```
►►──SUPPLIER─────────────────────────────────────────────────────────►◄
```

Returns a supplier object for the stream. When you send appropriate messages to the supplier object (see “The Supplier Class” on page 239), it enumerates all the lines in the stream object. The supplier enumerates the items in their line order.

The String Class

String objects represent character-string data values. A character string value can have any length and contain any characters. If you are familiar with earlier versions of REXX you might find the notation for functions more convenient than the notation for methods. See “Chapter 8. Functions” on page 297 for function descriptions.

The String class is a subclass of the Object class.

Methods the String class defines:

NEW (Class method. Overrides Object class method)
Arithmetic methods: +, -, *, /, %, //, **
Comparison methods: =, \=, <>, ><, ==, \== (Override Object class methods)
Comparison methods: >, <, >=, \<, <=, \>, >>, <<, >>=, \<<, <<=, \>>
Logical methods: &, |, &&, \
Concatenation methods: "" (abuttal), " " (blank), ||
ABBREV
ABS
BITAND
BITOR
BITXOR
B2X
CENTER (or CENTRE)
CHANGESTR
COMPARE
COPIES
COUNTSTR
C2D
C2X
DATATYPE
DELSTR
DELWORD
D2C
D2X
FORMAT
INSERT
LASTPOS
LEFT
LENGTH
MAKESTRING
MAX
MIN
OVERLAY
POS
REVERSE
RIGHT
SIGN
SPACE
STRING (Overrides Object class method)
STRIP
SUBSTR
SUBWORD
TRANSLATE
TRUNC

VERIFY
WORD
WORDINDEX
WORDLENGTH
WORDPOS
WORDS
X2B
X2C
X2D

Methods inherited from the Object class:

CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
UNSETMETHOD

Note: The String class also has available class methods that its metaclass, the Class class, defines.

NEW (Class Method)

```
▸▸──NEW(stringvalue)──────────────────────────────────────────────▸◂
```

Returns a new string object initialized with the characters in *stringvalue*.

Arithmetic Methods

```
▸▸──arithmetic_operator(argument)─────────────────────────────────▸◂
```

Note: For the prefix - and prefix + operators, omit the parentheses and *argument*.

Returns the result of performing the specified arithmetic operation on the receiver object. The receiver object and the *argument* must be valid numbers (see “Numbers” on page 15). The *arithmetic_operator* can be:

+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Integer division (divide and return the integer part of the result)
//	Remainder (divide and return the remainder—not modulo, because the result can be negative)
**	Exponentiation (raise a number to a whole-number power)

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

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

See "Chapter 11. Numbers and Arithmetic" on page 421 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 might have been rounded.

Examples:

```
5+5     ->   10
8-5     ->    3
5*2     ->   10
6/2     ->    3
9//4    ->    1
9%4     ->    2
2**3    ->    8
+5      ->    5            /* Prefix +  */
-5      ->   -5            /* Prefix -  */
```

Comparison Methods

```
►►—comparison_operator(argument)——————————————————————————————►◄
```

Returns 1 (true) or 0 (false), the result of performing the specified comparison operation. The receiver object and the *argument* are the terms compared. Both must be string objects.

The comparison operators you can use in a message are:

=	True if the terms are equal (for example, numerically or when padded)

\=, ><, <>
True if the terms are not equal (inverse of =)

>	Greater than
<	Less than
>=	Greater than or equal to
\<	Not less than
<=	Less than or equal to
\>	Not greater than

Examples:

```
5=5        ->    1          /* equal           */

42\=41     ->    1          /* All of these are */
42><41     ->    1          /* "not equal"      */
42<>41     ->    1

13>12      ->    1          /* Variations of    */
12<13      ->    1          /* less than and    */
13>=12     ->    1          /* greater than     */
12\<13     ->    0
12<=13     ->    1
12\>13     ->    1
```

All strict comparison operations have one of the characters doubled that define the operator. The == and \== operators check whether two strings match exactly. The two strings must be identical (character by character) and of the same length to be considered strictly equal.

The strict comparison operators such as >> or << carry out a simple character-by-character comparison. There is 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 a leading substring of another, then it is smaller than (less than) the other. The strict comparison operators do not attempt to perform a numeric comparison on the two operands.

For all the other comparison operators, if *both* terms are numeric, the language processor does a numeric comparison (ignoring, for example, leading zeros—see "Numeric Comparisons" on page 426). Otherwise, it treats both terms as character strings, ignoring leading and trailing blanks and padding the shorter string on the right with blanks.

Character comparison and strict comparison operations are both case-sensitive, and for both the exact collating order can depend on the character set. In an ASCII environment, the digits are lower than the alphabetic characters, and lowercase alphabetic characters are higher than uppercase alphabetic characters.

The strict comparison operators you can use in a message are:

==	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

Examples:

```
'space'=='space'     ->     1         /* Strictly equal     */

'space'\==' space'   ->     1         /* Strictly not equal */

'space'>>' space'    ->     1         /* Variations of      */
' space'<<'space'    ->     1         /* strictly greater   */
'space'>>=' space'   ->     1         /* than and less than */
'space'\<<' space'   ->     1
' space'<<='space'   ->     1
' space'\>>'space'   ->     1
```

Logical Methods

```
►►──logical_operator(argument)──────────────────────────────────────────►◄
```

Note: For NOT (prefix \), omit the parentheses and *argument*.

Returns 1 (true) or 0 (false), the result of performing the specified logical operation. The receiver object and the *argument* are character strings that evaluate to 1 or 0.

The *logical_operator* can be:

&	AND (Returns 1 if both terms are true.)
|	Inclusive OR (Returns 1 if either term or both terms are true.)
&&	Exclusive OR (Returns 1 if either term, but not both terms, is true.)

Prefix \ Logical NOT (Negates; 1 becomes `0`, and `0` becomes 1.)

Examples:

```
1&0     ->    0
1|0     ->    1
1&&0    ->    1
\1      ->    0
```

Concatenation Methods

```
►►—concatenation_operator(argument)——————————————————————————————————————►◄
```

Concatenates the receiver object with *argument*. (See “String Concatenation” on page 20.) The *concatenation_operator* can be:

"" concatenates without an intervening blank. The abuttal operator "" is the null string. The language processor uses the abuttal to concatenate two terms that another operator does not separate.

`||` concatenates without an intervening blank.

“ ” concatenates with one blank between the receiver object and the *argument*. (The operator “ ” is a blank.)

Examples:

```
num=33
say num"%"                    ->  33%          /* abuttal */
say num~''('%')                ->  33%

say "R"||"EXX"                ->  REXX         /* ||      */

say object rexx               ->  OBJECT REXX  /* blank   */
say 'OBJECT'~' '('REXX')      ->  OBJECT REXX
```

ABBREV

```
►►—ABBREV(info—┬─────────┬—)——————————————————————————————————————————————►◄
               └─,length─┘
```

Returns `1` if *info* is equal to the leading characters of the receiving string 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*.

Examples:

```
'Print'~ABBREV('Pri')      ->    1
'PRINT'~ABBREV('Pri')      ->    0
'PRINT'~ABBREV('PRI',4)    ->    0
'PRINT'~ABBREV('PRY')      ->    0
'PRINT'~ABBREV('')         ->    1
'PRINT'~ABBREV('',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.

Example:

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

:
  otherwise nop;
end;
```

(See “ABBREV (Abbreviation)” on page 304 for information about the ABBREV built-in function.)

ABS

```
>>--ABS--------------------------------------------------------><
```

Returns the absolute value ofthe receiving string. The result has no sign and is formatted according to the current NUMERIC settings.

Examples:

```
12.3~abs      ->     12.3
'-0.307'~abs  ->     0.307
```

(See “ABS (Absolute Value)” on page 304 for information about the ABS built-in function.)

BITAND

```
>>--BITAND--+-----------------------+--------------------------><
            '-(string--+------+--)-'
                       '-,pad-'
```

Returns a string composed of the receiver string and the argument *string* 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 you omit the *pad* character, 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 you provide *pad*, it extends the shorter of the two strings on the right before the logical operation. The default for *string* is the zero-length (null) string.

Examples:

```
'12'x~BITAND                    ->    '12'x
'73'x~BITAND('27'x)             ->    '23'x
'13'x~BITAND('5555'x)           ->    '1155'x
'13'x~BITAND('5555'x,'74'x)     ->    '1154'x
'pQrS'~BITAND(,'DF'x)           ->    'PQRS'      /* ASCII   */
```

(See “BITAND (Bit by Bit AND)” on page 307 for information about the BITAND built-in function.)

BITOR

```
>>--BITOR--+--------------------------+-------------------------------------------><
           +--(string--+--------+--)--+
                       +--,pad--+
```

Returns a string composed of the receiver string and the argument *string* logically inclusive-ORed, 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 you omit the *pad* character, 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 you provide *pad*, it extends the shorter of the two strings on the right before the logical operation. The default for *string* is the zero-length (null) string.

Examples:

```
'12'x~BITOR                    ->    '12'x
'15'x~BITOR('24'x)             ->    '35'x
'15'x~BITOR('2456'x)           ->    '3556'x
'15'x~BITOR('2456'x,'F0'x)     ->    '35F6'x
'1111'x~BITOR(,'4D'x)          ->    '5D5D'x
'pQrS'~BITOR(,'20'x)           ->    'pqrs' /* ASCII   */
```

(See “BITOR (Bit by Bit OR)” on page 308 for information about the BITOR built-in function.)

BITXOR

```
>>--BITXOR--+-----------------------+-------------------------------><
            '-(string--+------+--)--'
                       '-,pad-'
```

Returns a string composed of the receiver string and the argument *string* logically eXclusive-ORed, 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 you omit the *pad* character, 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 you provide *pad*, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for *string* is the zero-length (null) string.

Examples:

```
'12'x~BITXOR                      ->  '12'x
'12'x~BITXOR('22'x)               ->  '30'x
'1211'x~BITXOR('22'x)             ->  '3011'x
'1111'x~BITXOR('444444'x)         ->  '555544'x
'1111'x~BITXOR('444444'x,'40'x)   ->  '555504'x
'1111'x~BITXOR(,'4D'x)            ->  '5C5C'x
'C711'x~BITXOR('222222'x,' ')     ->  'E53302'x  /* ASCII  */
```

(See "BITXOR (Bit by Bit Exclusive OR)" on page 308 for information about the BITXOR built-in function.)

B2X

```
>>--B2X-------------------------------------------------------------><
```

Returns a string, in character format, that represents the receiving binary string converted to hexadecimal.

The receiving string is a string of binary (`0` or `1`) digits. It can be of any length. It can optionally include blanks (at 4-digit boundaries only, not leading or trailing). These are to improve readability; the language processor ignores them.

The returned string uses uppercase alphabetic characters for the values `A`–`F` and does not include blanks.

If the receiving binary string is a null string, B2X returns a null string. If the number of binary digits in the receiving string is not a multiple of four, the language processor adds up to three 0 digits on the left before the conversion to make a total that is a multiple of four.

Examples:

```
'11000011'~B2X     ->   'C3'
'10111'~B2X        ->   '17'
'101'~B2X          ->   '5'
'1 1111 0000'~B2X  ->   '1F0'
```

You can combine B2X with the methods X2D and X2C to convert a binary number into other forms.

Example:

```
'10111'~B2X~X2D  ->   '23'   /* decimal 23 */
```

(See “B2X (Binary to Hexadecimal)” on page 309 for information about the B2X built-in function.)

CENTER/CENTRE

```
>>──┬─CENTER(─┬──length──┬──────────┬──)──────────────────────────────><
    └─CENTRE(─┘          └─,──pad───┘
```

Returns a string of length *length* with the receiving string centered in it. The language processor adds *pad* characters as necessary to make up length. The *length* must be a positive whole number or zero. The default *pad* character is blank. If the receiving 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.

Note: To avoid errors because of the difference between British and American spellings, this method can be called either CENTRE or CENTER.

Examples:

```
abc~CENTER(7)               ->    '  ABC  '
abc~CENTER(8,'-')           ->    '--ABC---'
'The blue sky'~CENTRE(8)    ->    'e blue s'
'The blue sky'~CENTRE(7)    ->    'e blue '
```

(See “CENTER (or CENTRE)” on page 309 for information about the CENTER built-in function.)

CHANGESTR

```
>>──CHANGESTR(needle,newneedle)───────────────────────────────────────><
```

Returns a copy of the receiver object in which *newneedle* replaces all occurrences of *needle*.

Here are some examples:

```
101100~CHANGESTR('1','')     ->    '000'
101100~CHANGESTR('1','X')    ->    'X0XX00'
```

(See “CHANGESTR” on page 310 for information about the CHANGESTR built-in function.)

COMPARE

```
►►──COMPARE(string──┬──────┬──)────────────────────────────────────►◄
                    └─,pad─┘
```

Returns 0 if the argument *string* is identical to the receiving string. 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.

Examples:

```
'abc'~COMPARE('abc')         ->    0
'abc'~COMPARE('ak')          ->    2
'ab '~COMPARE('ab')          ->    0
'ab '~COMPARE('ab',' ')      ->    0
'ab '~COMPARE('ab','x')      ->    3
'ab-- '~COMPARE('ab','-')    ->    5
```

(See “COMPARE” on page 313 for information about the COMPARE built-in function.)

COPIES

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

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

Examples:

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

(See “COPIES” on page 315 for information about the COPIES built-in function.)

COUNTSTR

```
►►──COUNTSTR(needle)──────────────────────────────────────────────►◄
```

Returns a count of the occurrences of *needle* in the receiving string that do not overlap.

Here are some examples:

```
'101101'~COUNTSTR('1')       ->    4
'J0KKK0'~COUNTSTR('KK')      ->    1
```

(See “COUNTSTR” on page 315 for information about the COUNTSTR built-in function.)

C2D

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

Returns the decimal value of the binary representation of the receiving 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 specify *n*, it is the length of the returned result. If you do not specify *n*, the receiving string is processed as an unsigned binary number. If the receiving string is null, C2D returns `0`.

Examples:

```
'09'X~C2D      ->        9
'81'X~C2D      ->      129
'FF81'X~C2D    ->    65409
''~C2D         ->        0
'a'~C2D        ->       97     /*  ASCII   */
```

If you specify *n*, the receiving string is taken as a signed number expressed in *n* characters. The number is positive if the leftmost bit is off, and negative if the leftmost bit is on. In both cases, it is converted to a whole number, which can therefore be negative. The receiving string is padded on the left with '00'x characters (not “sign-extended”), or truncated on the left to *n* characters. This padding or truncation is as though `receiving_string~RIGHT(`*n*`,'00'x)` had been processed. If *n* is `0`, C2D always returns `0`.

Examples:

```
'81'X~C2D(1)      ->     -127
'81'X~C2D(2)      ->      129
'FF81'X~C2D(2)    ->     -127
'FF81'X~C2D(1)    ->     -127
'FF7F'X~C2D(1)    ->      127
'F081'X~C2D(2)    ->    -3967
'F081'X~C2D(1)    ->     -127
'0031'X~C2D(0)    ->        0
```

(See "C2D (Character to Decimal)" on page 316 for information about the C2D built-in function.)

C2X

```
►►──C2X──────────────────────────────────────────────────────────────►◄
```

Returns a string, in character format, that represents the receiving string converted to hexadecimal. The returned string contains twice as many bytes as the receiving string. On an ASCII system, sending a C2X message to the receiving string 1 returns 31 because '31'X is the ASCII representation of 1.

The returned string has uppercase alphabetic characters for the values A–F and does not include blanks. The receiving string can be of any length. If the receiving string is null, C2X returns a null string.

Examples:

```
'0123'X~C2X    ->    '0123'   /* '30313233'X     in ASCII */
'ZD8'~C2X      ->    '5A4438' /* '354134343338'X in ASCII */
```

(See "C2X (Character to Hexadecimal)" on page 316 for information about the C2X built-in function.)

DATATYPE

```
►►──DATATYPE──┬──────────┬──────────────────────────────────────────►◄
              └─(type)───┘
```

Returns NUM if you specify no argument and the receiving string is a valid REXX number that can be added to 0 without error. It returns CHAR if the receiving string is not a valid number.

If you specify *type*, it returns 1 if the receiving string matches the type. Otherwise, it returns 0. If the receiving string is null, the method returns 0 (except when the *type* is X or B, for which DATATYPE returns 1 for a null string). The following are valid *type*s. You need to specify only the capitalized letter, or the number of the last type listed. The language processor ignores all characters following it.

Alphanumeric returns 1 if the receiving string contains only characters from the ranges `a–z`, `A–Z`, and `0–9`.

Binary returns 1 if the receiving string contains only the characters `0` or `1`, or a blank. Blanks can appear only between groups of 4 binary characters. It also returns 1 if *string* is a null string, which is a valid binary string.

Lowercase returns 1 if the receiving string contains only characters from the range `a–z`.

Mixed case returns 1 if the receiving string contains only characters from the ranges `a–z` and `A–Z`.

Number returns 1 if `receiving_string~DATATYPE` returns `NUM`.

Symbol returns 1 if the receiving string is a valid symbol, that is, if SYMBOL(string) does not return `BAD`. (See "Symbols" on page 14.) Note that both uppercase and lowercase alphabetic characters are permitted.

Uppercase returns 1 if the receiving string contains only characters from the range `A–Z`.

Variable returns 1 if the receiving string could appear on the left-hand side of an assignment without causing a SYNTAX condition.

Whole number
returns 1 if the receiving string is a whole number under the current setting of NUMERIC DIGITS.

heXadecimal returns 1 if the receiving 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 the receiving string is a null string.

9 Digits returns 1 if `receiving_string~DATATYPE('W')` returns 1 when NUMERIC DIGITS is set to 9.

Examples:

```
' 12 '~DATATYPE          ->   'NUM'
''~DATATYPE              ->   'CHAR'
'123*'~DATATYPE          ->   'CHAR'
'12.3'~DATATYPE('N')     ->    1
'12.3'~DATATYPE('W')     ->    0
'Fred'~DATATYPE('M')     ->    1
''~DATATYPE('M')         ->    0
```

```
'Fred'~DATATYPE('L')     ->    0
'?20K'~DATATYPE('s')     ->    1
'BCd3'~DATATYPE('X')     ->    1
'BC d3'~DATATYPE('X')    ->    1
```

Note: The DATATYPE method tests the meaning or type of characters in a string, independent of the encoding of those characters (for example, ASCII or EBCDIC).

(See "DATATYPE" on page 317 for information about the DATATYPE built-in function.)

DELSTR

```
>>──DELSTR(n──┬─────────┬──)──────────────────────────────────────><
              └─,length─┘
```

Returns a copy of the receiving 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 method 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 the receiving string, the method returns the receiving string unchanged.

Examples:

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

(See "DELSTR (Delete String)" on page 321 for information about the DELSTR built-in function.)

DELWORD

```
>>──DELWORD(n──┬─────────┬──)─────────────────────────────────────><
               └─,length─┘
```

Returns a copy of the receiving 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 the receiving string, the method deletes the remaining words in the receiving 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 the receiving string, the method returns the receiving string unchanged. The string deleted includes any blanks following the final word involved but none of the blanks preceding the first word involved.

Examples:

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

(See “DELWORD (Delete Word)” on page 321 for information about the DELWORD built-in function.)

D2C

```
>>──D2C──┬─────┬──────────────────────────────────────────────────────><
         └─(n)─┘
```

Returns a string, in character format, that is the ASCII representation of the receiving string, a decimal number. If you specify *n*, it is the length of the final result in characters; leading blanks are added to the returned string. The *n* must be a positive whole number or zero.

The receiving string must not have more digits than the current setting of NUMERIC DIGITS.

If you omit *n*, the receiving string 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.

Examples:

```
'65'~D2C        ->   'A'      /* '41'x is an ASCII 'A'    */
'65'~D2C(1)     ->   'A'
'65'~D2C(2)     ->   ' A'
'65'~D2C(5)     ->   '    A'
'109'~D2C       ->   'm'      /* '6D'x  is an ASCII 'm'   */
'-109'~D2C(1)   ->   'ô'      /* '93'x  is an ASCII 'ô'   */
'76'~D2C(2)     ->   ' L'     /* '4C'x  is an ASCII ' L'  */
'-180'~D2C(2)   ->   ' L'
```

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

(See “D2C (Decimal to Character)” on page 323 for information about the D2C built-in function.)

D2X

```
►►──D2X─┬─────┬──────────────────────────────────────────────────────►◄
        └─(n)─┘
```

Returns a string, in character format, that represents the receiving string, a decimal number converted to hexadecimal. The returned string uses uppercase alphabetic characters for the values A–F and does not include blanks.

The receiving string must not have more digits than the current setting of NUMERIC DIGITS.

If you specify *n*, it is the length of the final result in characters. After conversion the returned 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. If you specify *n*, it must be a positive whole number or zero.

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

Examples:

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

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

(See "D2X (Decimal to Hexadecimal)" on page 323 for information about the D2X built-in function.)

FORMAT

```
►►──FORMAT─┬──────────────────────────────────────────────────┬──────►◄
           └─(─before─┬─────────────────────────────────────┬─)─┘
                      └─,─┬───────┬──┬────────────────────┬─┘
                          └─after─┘  └─,─┬──────┬─┬───────┬─┘
                                         └─expp─┘ └─,expt─┘
```

Returns the receiving string, a number, rounded and formatted.

The number is first rounded according to standard REXX rules, as though the operation `receiving_string+0` had been carried out. If you specify no arguments the result of the method is the same as the result of this operation. If you specify any options, the number is formatted as described in the following.

The *before* and *after* options describe how many characters are to be used for the integer and decimal parts of the result. If you omit either or both of them, the number of characters 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.

Examples:

```
'3'~FORMAT(4)            ->    '   3'
'1.73'~FORMAT(4,0)       ->    '   2'
'1.73'~FORMAT(4,3)       ->    '   1.730'
'-.76'~FORMAT(4,1)       ->    '  -0.8'
'3.03'~FORMAT(4)         ->    '   3.03'
' - 12.73'~FORMAT(,4)    ->    '-12.7300'
' - 12.73'~FORMAT        ->    '-12.73'
'0.000'~FORMAT           ->    '0'
```

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. *expp* sets the number of places for the exponent part; the default is to use as many as needed (which can be zero). *expt* specifies when the exponential expression is used. The default is the current setting of NUMERIC DIGITS.

If *expp* is `0`, the number is not an exponential expression. If *expp* is not large enough to contain the exponent, an error results.

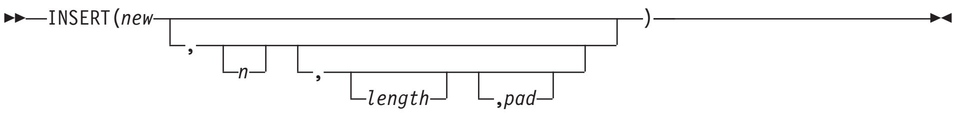
If the number of places needed for the integer or decimal part exceeds *expt* or twice *expt*, respectively, exponential notation is used. If *expt* is `0`, exponential notation is always used unless the exponent would be `0`. (If *expp* is `0`, this overrides a `0` value of *expt*.) If the exponent 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, the number is not an exponential expression.

Examples:

```
'12345.73'~FORMAT(,,2,2)    ->    '1.234573E+04'
'12345.73'~FORMAT(,3,,0)    ->    '1.235E+4'
'1.234573'~FORMAT(,3,,0)    ->    '1.235'
'12345.73'~FORMAT(,,3,6)    ->    '12345.73'
'1234567e5'~FORMAT(,3,0)    ->    '123456700000.000'
```

(See “FORMAT” on page 325 for information about the FORMAT built-in function.)

INSERT



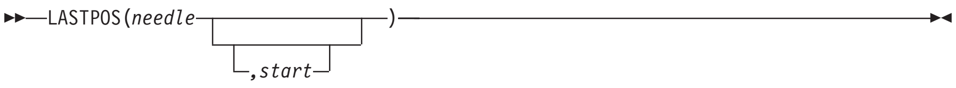
Inserts the string *new*, padded or truncated to length *length*, into the receiving string. after the *n*th character. The default value for *n* is `0`, which means insertion at the beginning of the string. If specified, *n* and *length* must be positive whole numbers or zero. If *n* is greater than the length of the receiving string, the string *new* is padded at the beginning. 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.

Examples:

```
'abc'~INSERT('123')             ->    '123abc'
'abcdef'~INSERT(' ',3)          ->    'abc def'
'abc'~INSERT('123',5,6)         ->    'abc  123   '
'abc'~INSERT('123',5,6,'+')     ->    'abc++123+++'
'abc'~INSERT('123',,5,'-')      ->    '123--abc'
```

(See “INSERT” on page 327 for information about the INSERT built-in function.)

LASTPOS



Returns the position of the last occurrence of a string, *needle*, in the receiving string. (See also “POS” on page 228.) It returns `0` if *needle* is the null string or not found. By default, the search starts at the last character of the receiving string and scans backward. You can override this by specifying *start*, the point at which the backward scan starts. The *start* must be a positive whole number and defaults to `receiving_string~length` if larger than that value or omitted.

Examples:

```
'abc def ghi'~LASTPOS(' ')      ->    8
'abcdefghi'~LASTPOS(' ')        ->    0
'efgxyz'~LASTPOS('xy')          ->    4
'abc def ghi'~LASTPOS(' ',7)    ->    4
```

(See "LASTPOS (Last Position)" on page 327 for information about the LASTPOS built-in function.)

LEFT

```
>>──LEFT(length──┬──────┬─)──────────────────────────────────────────────><
                 └─,pad─┘
```

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

```
>>──SUBSTR(string,1,length──┬──────┬─)───────────────────────────────────><
                            └─,pad─┘
```

Examples:

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

(See "LEFT" on page 328 for information about the LEFT built-in function.)

LENGTH

```
>>──LENGTH───────────────────────────────────────────────────────────────><
```

Returns the length of the receiving string.

Examples:

```
'abcdefgh'~LENGTH     ->    8
'abc defg'~LENGTH     ->    8
''~LENGTH             ->    0
```

(See "LENGTH" on page 328 for information about the LENGTH built-in function.)

MAKESTRING

```
>>──MAKESTRING──────────────────────────────────────────────────────────><
```

Returns a string with the same string value as the receiver object. If the receiver is an instance of a subclass of the String class, this method returns an equivalent string object. If the receiver is a string object (not an instance of a subclass of the String class), this method returns the receiver object. See “Required String Values” on page 105.

MAX

```
>>──MAX──┬──────────────────────┬───────────────────────────────────────><
         │     ┌─,──────┐       │
         │     ▼        │       │
         └─(───number───┴───)───┘
```

Returns the largest number from among the receiver and any arguments. The number that MAX returns is formatted according to the current NUMERIC settings. You can specify any number of *number*s.

Examples:

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

(See “MAX (Maximum)” on page 332 for information about the MAX built-in function.)

MIN

```
>>──MIN──┬──────────────────────┬───────────────────────────────────────><
         │     ┌─,──────┐       │
         │     ▼        │       │
         └─(───number───┴───)───┘
```

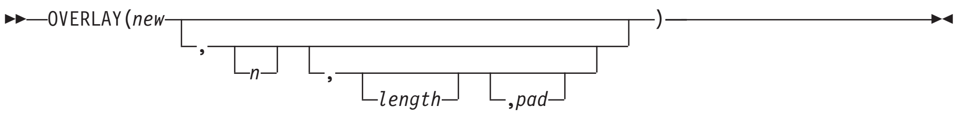
Returns the smallest number from among the receiver and any arguments. The number that MIN returns is formatted according to the current NUMERIC settings. You can specify any number of *number*s.

Examples:

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

(See "MIN (Minimum)" on page 332 for information about the MIN built-in function.)

OVERLAY



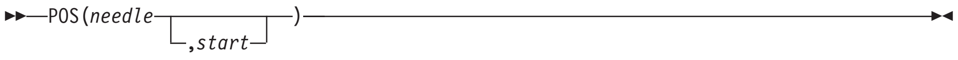
Returns the receiving string, which, starting at the *n*th character, is overlaid with the string *new*, padded or truncated to length *length*. The overlay can extend beyond the end of the receiving string. If you specify *length*, it must be a positive whole number or zero. The default value for *length* is the length of *new*. If *n* is greater than the length of the receiving 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.

Examples:

```
'abcdef'~OVERLAY(' ',3)         ->    'ab def'
'abcdef'~OVERLAY('.',3,2)       ->    'ab. ef'
'abcd'~OVERLAY('qq')            ->    'qqcd'
'abcd'~OVERLAY('qq',4)          ->    'abcqq'
'abc'~OVERLAY('123',5,6,'+')    ->    'abc+123+++'
```

(See "OVERLAY" on page 333 for information about the OVERLAY built-in function.)

POS



Returns the position in the receiving string of another string, *needle*. (See also "LASTPOS" on page 225.) It returns `0` if *needle* is the null string or is not found or if *start* is greater than the length of the receiving string. By default, the search starts at the first character of the receiving string (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.

Examples:

```
'Saturday'~POS('day')       ->    6
'abc def ghi'~POS('x')      ->    0
'abc def ghi'~POS(' ')      ->    4
'abc def ghi'~POS(' ',5)    ->    8
```

(See "POS (Position)" on page 333 for information about the POS built-in function.)

REVERSE

```
>>--REVERSE-----------------------------------------------------------><
```

Returns the receiving string reversed.

Examples:

```
'ABc.'~REVERSE    ->    '.cBA'
'XYZ '~REVERSE    ->    ' ZYX'
```

(See "REVERSE" on page 335 for information about the REVERSE built-in function.)

RIGHT

```
>>--RIGHT(length--+------+--)-----------------------------------------><
                  '-,pad-'
```

Returns a string of length *length* containing the rightmost *length* characters of the receiving 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.

Examples:

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

(See "RIGHT" on page 335 for information about the RIGHT built-in function.)

SIGN

```
>>--SIGN--------------------------------------------------------------><
```

Returns a number that indicates the sign of the receiving string, which is a number. The receiving string is first rounded according to standard REXX rules, as though the operation `receiving_string+0` had been carried out. It returns `-1` if the receiving string is less than `0`, `0` if it is `0`, and `1` if it is greater than `0`.

Examples:

```
'12.3'~SIGN       ->     1
' -0.307'~SIGN    ->    -1
0.0~SIGN          ->     0
```

(See “SIGN” on page 335 for information about the SIGN built-in function.)

SPACE

```
>>--SPACE--+----------------+-----------------------------------------><
           '-(n--+------+--)'
                 '-,pad-'
```

Returns the blank-delimited words in the receiving 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.

Examples:

```
'abc  def  '~SPACE           ->    'abc def'
'  abc def'~SPACE(3)         ->    'abc   def'
'abc  def  '~SPACE(1)        ->    'abc def'
'abc  def  '~SPACE(0)        ->    'abcdef'
'abc  def  '~SPACE(2,'+')    ->    'abc++def'
```

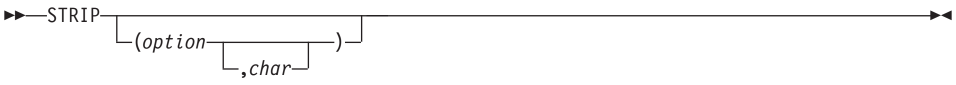
(See “SPACE” on page 336 for information about the SPACE built-in function.)

STRING

```
>>--STRING------------------------------------------------------------><
```

Returns a string with the same string value as the receiver object. If the receiver is an instance of a subclass of the String class, this method returns a string having an equivalent value. If the receiver is a string (but is not an instance of a subclass of the String class), this method returns the receiver object. See also the STRING method of the Object class in “STRING” on page 185.

STRIP



Returns the receiving string with leading characters, trailing characters, or both, removed, based on the *option* you specify. The following are valid *option*s. (You need to specify only the first capitalized letter; the language processor ignores all characters following it.)

Both	Removes both leading and trailing characters. This is the default.
Leading	Removes leading characters.
Trailing	Removes trailing characters.

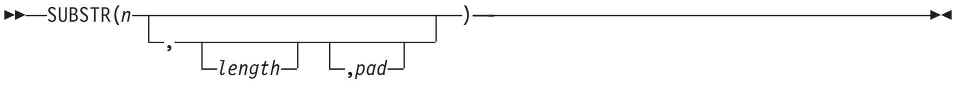
The *char* specifies the character to be removed, and the default is a blank. If you specify *char*, it must be exactly one character long.

Examples:

```
'  ab c  '~STRIP         ->    'ab c'
'  ab c  '~STRIP('L')    ->    'ab c  '
'  ab c  '~STRIP('t')    ->    '  ab c'
'12.7000'~STRIP(,0)      ->    '12.7'
'0012.700'~STRIP(,0)     ->    '12.7'
```

(See “STRIP” on page 344 for information about the STRIP built-in function.)

SUBSTR



Returns the substring of the receiving 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 `receiving_string~LENGTH`, only pad characters are returned.

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

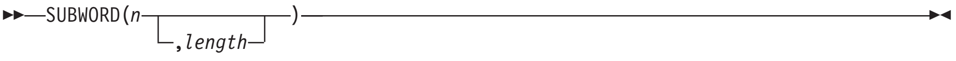
Examples:

```
'abc'~SUBSTR(2)          ->    'bc'
'abc'~SUBSTR(2,4)        ->    'bc  '
'abc'~SUBSTR(2,6,'.')    ->    'bc....'
```

Note: In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting substrings, in particular if you need to extract more than one substring from a string. See also“LEFT” on page 226 and “RIGHT” on page 229.

(See “SUBSTR (Substring)” on page 344 for information about the SUBSTR built-in function.)

SUBWORD



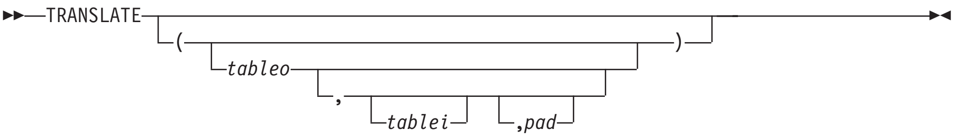
Returns the substring of the receiving string that starts at the *n*th word and is up to *length* blank-delimited words. The *n* must be a positive whole number. If you omit *length*, it defaults to the number of remaining words in the receiving string. The returned string never has leading or trailing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

Examples:

```
'Now is the  time'~SUBWORD(2,2)    ->    'is the'
'Now is the  time'~SUBWORD(3)      ->    'the  time'
'Now is the  time'~SUBWORD(5)      ->    ''
```

(See “SUBWORD” on page 345 for information about the SUBWORD built-in function.)

TRANSLATE



Returns the receiving string with each character translated to another character or unchanged. You can also use this method to reorder the characters in the receiving string.

The output table is *tableo* and the input translation table is *tablei*. TRANSLATE searches *tablei* for each character in the receiving string. If the character is found, 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 the receiving string is used. The result string is always of the same length as the receiving string.

The tables can be of any length. If you specify translation table and omit *pad*, the receiving string is 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.

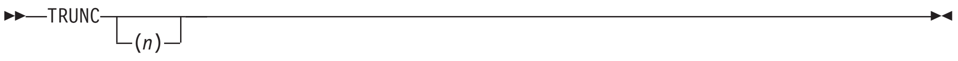
Examples:

```
'abcdef'~TRANSLATE                       ->    'ABCDEF'
'abcdef'~TRANSLATE('12','ec')            ->    'ab2d1f'
'abcdef'~TRANSLATE('12','abcd','.')      ->    '12..ef'
'APQRV'~TRANSLATE(,'PR')                 ->    'A Q V'
'APQRV'~TRANSLATE(XRANGE('00'X,'Q'))     ->    'APQ  '
'4123'~TRANSLATE('abcd','1234')          ->    'dabc'
```

Note: The last example shows how to use the TRANSLATE method to reorder the characters in a string. In the example, the last character of any 4-character string specified as the first argument would be moved to the beginning of the string.

(See "TRANSLATE" on page 349 for information about the TRANSLATE built-in function.)

TRUNC



Returns the integer part the receiving string, which is a 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 receiving string is first rounded according to standard REXX rules, as though the operation `receiving_string+0` had been carried out. This number is then truncated to *n* decimal places or trailing zeros are added if needed to reach the specified length. The result is never in exponential form. If there are no nonzero digits in the result, any minus sign is removed.

Examples:

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

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

(See "TRUNC (Truncate)" on page 350 for information about the TRUNC built-in function.)

VERIFY

```
>>──VERIFY(reference─┬──────────────────────────┬─)──────────────────────────><
                     └─,─┬────────┬─┬────────┬──┘
                         └─option─┘ └─,start─┘
```

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

The *option* can be either `Nomatch` (the default) or `Match`. (You need to specify only the first capitalized and highlighted letter; the language processor ignores all characters following the first character, which can be in uppercase or lowercase.)

If you specify `Match`, the method returns the position of the first character in the receiving 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 the receiving string. You can override this by specifying a different *start* point, which must be a positive whole number.

If the receiving string is null, the method returns `0`, regardless of the value of the *option*. Similarly, if *start* is greater than `receiving_string~LENGTH`, the method returns `0`. If *reference* is null, the method returns `0` if you specify `Match`. Otherwise, the method returns the *start* value.

Examples:

```
'123'~VERIFY('1234567890')              ->    0
'1Z3'~VERIFY('1234567890')              ->    2
'AB4T'~VERIFY('1234567890')             ->    1
'AB4T'~VERIFY('1234567890','M')         ->    3
'AB4T'~VERIFY('1234567890','N')         ->    1
'1P3Q4'~VERIFY('1234567890',,3)         ->    4
'123'~VERIFY('',N,2)                    ->    2
'ABCDE'~VERIFY('',,3)                   ->    3
'AB3CD5'~VERIFY('1234567890','M',4)     ->    6
```

(See “VERIFY” on page 353 for information about the VERIFY built-in function.)

WORD

```
►►──WORD(n)──────────────────────────────────────────────────────────►◄
```

Returns the *n*th blank-delimited word in the receiving string or the null string if the receiving string has fewer than *n* words. The *n* must be a positive whole number. This method is exactly equivalent to `receiving_string~SUBWORD(n,1)`.

Examples:

```
'Now is the time'~WORD(3)    ->    'the'
'Now is the time'~WORD(5)    ->    ''
```

(See “WORD” on page 354 for information about the WORD built-in function.)

WORDINDEX

```
►►──WORDINDEX(n)─────────────────────────────────────────────────────►◄
```

Returns the position of the first character in the *n*th blank-delimited word in the receiving string. It returns `0` if the receiving string has fewer than *n* words. The *n* must be a positive whole number.

Examples:

```
'Now is the time'~WORDINDEX(3)    ->    8
'Now is the time'~WORDINDEX(6)    ->    0
```

(See “WORDINDEX” on page 354 for information about the WORDINDEX built-in function.)

WORDLENGTH

```
►►──WORDLENGTH(n)────────────────────────────────────────────────────►◄
```

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

Examples:

```
'Now is the time'~WORDLENGTH(2)       ->    2
'Now comes the time'~WORDLENGTH(2)    ->    5
'Now is the time'~WORDLENGTH(6)       ->    0
```

(See “WORDLENGTH” on page 355 for information about the WORDLENGTH built-in function.)

WORDPOS

```
>>──WORDPOS(phrase─┬────────┬─)──────────────────────────────────────────><
                   └─,start─┘
```

Returns the word number of the first word of *phrase* found in the receiving string, or `0` if *phrase* contains no words or if *phrase* is not found. Several blanks between words in either *phrase* or the receiving 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 the receiving string. You can override this by specifying *start* (which must be positive), the word at which the search is to be started.

Examples:

```
'now is the time'~WORDPOS('the')              ->  3
'now is the time'~WORDPOS('The')              ->  0
'now is the time'~WORDPOS('is the')           ->  2
'now is the time'~WORDPOS('is   the')         ->  2
'now is   the time'~WORDPOS('is   time ')     ->  0
'To be or not to be'~WORDPOS('be')            ->  2
'To be or not to be'~WORDPOS('be',3)          ->  6
```

(See “WORDPOS (Word Position)” on page 355 for information about the WORDPOS built-in function.)

WORDS

```
>>──WORDS────────────────────────────────────────────────────────────────><
```

Returns the number of blank-delimited words in the receiving string.

Examples:

```
'Now is the time'~WORDS      ->    4
' '~WORDS                    ->    0
```

(See “WORDS” on page 355 for information about the WORDS built-in function.)

X2B

```
►►──X2B──────────────────────────────────────────────────────────►◄
```

Returns a string, in character format, that represents the receiving string, which is a string of hexadecimal characters converted to binary. The receiving string can be of any length. Each hexadecimal character is converted to a string of 4 binary digits. The receiving string can optionally include blanks (at byte boundaries only, not leading or trailing) to improve readability; they are ignored.

The returned string has a length that is a multiple of four, and does not include any blanks.

If the receiving string is null, the method returns a null string.

Examples:

```
'C3'~X2B       ->  '11000011'
'7'~X2B        ->  '0111'
'1 C1'~X2B     ->  '000111000001'
```

You can combine X2B with the methods D2X and C2X to convert numbers or character strings into binary form.

Examples:

```
'C3'x~C2X~X2B  ->  '11000011'
'129'~D2X~X2B  ->  '10000001'
'12'~D2X~X2B   ->  '1100'
```

(See “X2B (Hexadecimal to Binary)” on page 356 for information about the X2B built-in function.)

X2C

```
►►──X2C──────────────────────────────────────────────────────────►◄
```

Returns a string, in character format, that represents the receiving string, which is a hexadecimal string converted to character. The returned string is half as many bytes as the receiving string. The receiving string can be 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 the receiving string (at byte boundaries only, not leading or trailing) to improve readability; they are ignored.

If the receiving string is null, the method returns a null string.

Examples:

```
'4865 6c6c 6f'~X2C ->  'Hello'     /*  ASCII            */
'3732 73'~X2C      ->  '72s'       /*  ASCII            */
```

(See "X2C (Hexadecimal to Character)" on page 357 for information about the X2C built-in function.)

X2D

```
>>──X2D──┬─────┬──────────────────────────────────────────────────────────────><
         └─(n)─┘
```

Returns the decimal representation of the receiving string, which 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 the receiving string (at byte boundaries only, not leading or trailing) to improve readability; they are ignored.

If the receiving string is null, the method returns `0`.

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

Examples:

```
'0E'~X2D        ->    14
'81'~X2D        ->    129
'F81'~X2D       ->    3969
'FF81'~X2D      ->    65409
'46 30'X~X2D    ->    240          /*  ASCII   */
'66 30'X~X2D    ->    240          /*  ASCII   */
```

If you specify *n*, the receiving 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 both cases it is converted to a whole number, which can be negative. If *n* is `0`, the method returns `0`.

If necessary, the receiving string is padded on the left with `0` characters (note, not "sign-extended"), or truncated on the left to *n* characters.

Examples:

```
'81'~X2D(2)      ->    -127
'81'~X2D(4)      ->    129
'F081'~X2D(4)    ->    -3967
'F081'~X2D(3)    ->    129
'F081'~X2D(2)    ->    -127
'F081'~X2D(1)    ->    1
'0031'~X2D(0)    ->    0
```

(See “X2D (Hexadecimal to Decimal)” on page 357 for information about the X2D built-in function.)

The Supplier Class

You can use a supplier object to enumerate the items a collection contained at the time of the supplier's creation. The following methods return a supplier object:
- The SUPPLIER methods of the Array, Bag, Directory, List, Queue, Relation, Set, Table, and Stream classes
- The METHODS method of the Class class

The Supplier class is a subclass of the Object class.

Methods the Supplier class defines:

NEW (Class method. Overrides Object class method.)
AVAILABLE
INDEX
ITEM
NEXT

Methods inherited from the Object class:

Operator methods: =, ==, \=, ><, <>, \==
CLASS
COPY
DEFAULTNAME
HASMETHOD
INIT
OBJECTNAME
OBJECTNAME=
REQUEST
RUN
SETMETHOD
START
STRING
UNSETMETHOD

Note: The Supplier class also has available class methods that its metaclass, the Class class, defines.

NEW (Class Method)

```
►►──NEW(values,indexes)──────────────────────────────────────────────────►◄
```

Returns a new supplier object. The *values* argument must be an array of values over which the supplier iterates. The *indexes* argument is an array of index values with a one-to-one correspondence to the objects contained in the values array. The created supplier iterates over the arrays, returning elements of the values array in response to ITEM messages, and elements of the indexes array in response to INDEX messages. The supplier iterates for the number of items contained in the values array, returning the NIL object for any nonexistent items in either array.

AVAILABLE

```
►►──AVAILABLE────────────────────────────────────────────────────────────►◄
```

Returns 1 (true) if an item is available from the supplier (that is, if the ITEM method would return a value). It returns 0 (false) if the collection is empty or the supplier has already enumerated the entire collection.

INDEX

```
►►──INDEX────────────────────────────────────────────────────────────────►◄
```

Returns the index of the current item in the collection. If no item is available, that is, if AVAILABLE would return false, the language processor raises an error.

ITEM

```
►►──ITEM─────────────────────────────────────────────────────────────────►◄
```

Returns the current item in the collection. If no item is available, that is, if AVAILABLE would return false, the language processor raises an error.

NEXT

```
►►──NEXT──────────────────────────────────────────────────────────────►◄
```

Moves to the next item in the collection. By repeatedly sending NEXT to the supplier (as long as AVAILABLE returns true), you can enumerate all items in the collection. If no item is available, that is, if AVAILABLE would return false, the language processor raises an error.

Examples

```
desserts=.array~of(apples, peaches, pumpkins, 3.14159) /* Creates array */
say "The desserts we have are:"
baker=desserts~supplier            /* Creates supplier object named BAKER */
do while baker~available           /* Array suppliers are sequenced       */
  if baker~index=4
  then say baker~item "is pi, not pie!!!"
  else say baker~item
  baker~next
end

/* Produces: */
/* The desserts we have are: */
/* APPLES                    */
/* PEACHES                   */
/* PUMPKINS                  */
/* 3.14159 is pi, not pie!!! */
```

The WindowsProgramManager Class

Object REXX provides a class for interaction with the Windows Program Manager. You can use this class to create program groups and shortcuts to access your programs. This class is specifically for Windows systems and may not be available on other systems.

The WindowsProgramManager class is defined in the file WINSYSTM.CLS. Use a `::requires` statement to activate its function:

```
::requires "winsystm.cls"
```

A sample program `DESKTOP.REX` is provided in the `OBJREXX\SAMPLES` directory.

Methods of the WindowsProgramManager class are:

Table 2. Methods Available to the WindowsProgramManager Class

Method...	...on page
AddDeskTopIcon	242
AddGroup	244

Table 2. Methods Available to the WindowsProgramManager Class (continued)

Method...	...on page
AddItem	244
AddShortCut	243
DeleteGroup	245
DeleteItem	245
Init	245
ShowGroup	246

AddDesktopIcon

```
>>--AddDesktopIcon--(--name--,--program--+---------------------------------------+------------->
                                         |                            .-0------.  |
                                         '-,--+----------+--,--+--------+-----'
                                              '-iconfile-'     '-iconnr-'

>--+------------------------------------------------------------------------------------------------+--)--><
   |                .-"PERSONAL"-.                                             .-"NORMAL"----.  |
   '-,--+---------+-,-+------------+--,--+-----------+--,--+--------+--,--+-------------+--'
        '-workdir-'   '-"COMMON"---'     '-arguments-'     '-hotkey-'     +-"MAXIMIZED"-+
                                                                          '-"MINIMIZED"-'
```

Adds a shortcut to the Windows desktop. A sample program `DESKICON.REX` is provided in the `OBJREXX\SAMPLES` directory.

Arguments:

The arguments are:

name The name of the shortcut, displayed below the icon.

program
The program file launched by the shortcut.

iconfile
The name of the icon used for the shortcut. If not specified, the icon of *program* is used.

iconnr The number of the icon within the *iconfile*. The default is 0.

workdir
The working directory of the shortcut.

location
Either of the following locations:

"PERSONAL" The shortcut is personal and displayed only on the desktop of the user.

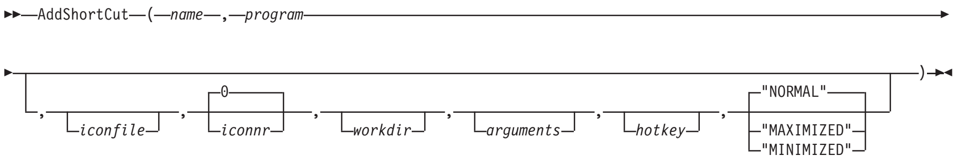
"COMMON" The shortcut is common to all users and displayed on the desktop of all users.

arguments
The arguments passed to the program that the shortcut refers to.

hotkey
The virtual key to be used as a hotkey to open the shortcut. For a list of the key names, see “Symbolic Names for Virtual Keys” on page 246.

run Specifies one of the options listed in the syntax diagram. The default is "NORMAL".

AddShortCut



Creates a shortcut within the specified folder.

Arguments:
The arguments are:

name The full name of the shortcut.

program
The program file launched by the shortcut.

iconfile
The name of the icon used for the shortcut. If not specified, the icon of *program* is used.

iconnr The number of the icon within the *iconfile*. The default is 0.

workdir
The working directory of the shortcut.

arguments
The arguments passed to the program that the shortcut refers to.

hotkey
The virtual key to be used as a hotkey to open the shortcut. For a list of the key names, see “Symbolic Names for Virtual Keys” on page 246.

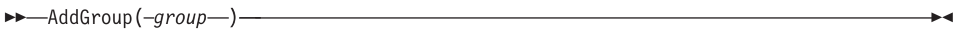
run Specifies one of the options listed in the syntax diagram. The default is "NORMAL".

Example:

The following example creates a shortcut named "My NotePad" to the Notepad editor wihin the directory c:\temp:

```
pm = .WindowsProgramManager~new
if pm~InitCode \= 0 then exit
pm~AddShortCut("c:\temp\My Notepad","%SystemRoot%\system32\notepad.exe")
::requires "winsystm.cls"
```

AddGroup



Adds a program group to the Programs group of the desktop. If the group already exists, it is opened. The *group* argument specifies the name of the program group to be added. Example:

```
AddGroup("Object REXX Redbook")
```

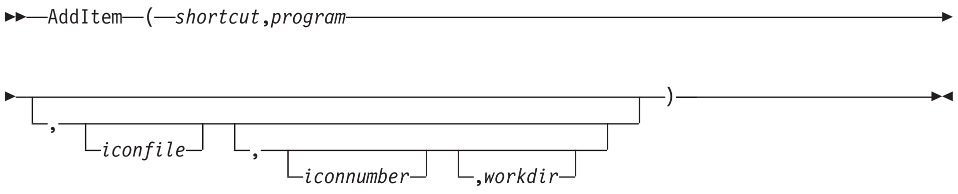
Note: The name that you specify for the *group* argument must not contain any brackets or parenthesis. Otherwise, this method fails.

Return value:

0 The method was successful.

1 The method failed.

AddItem



Adds a shortcut to a program group. The shortcut is placed into the last group used with either AddGroup or ShowGroup. Example:

```
AddItem('OODialog Samples',,
'rexx oodialog\samples\sample.rex',,
'oodialog\samples\oodialog.ico')
```

Note: The name that you specify for the *group* argument must not contain characters that are not valid, such as brackets or parenthesis. Otherwise, this method fails. Some characters are changed, for example / to _.

Return value:

0 The method was successful.

1 The method failed.

DeleteGroup

```
►►─DeleteGroup(─group─)──────────────────────────────────────────────►◄
```

Deletes a program group from the desktop. The *group* argument specifies the name of the program group to be deleted.

Return value:

0 The method was successful.

1 The method failed.

DeleteItem

```
►►─DeleteItem(shortcut)──────────────────────────────────────────────►◄
```

Deletes a shortcut from a program group.

Return value:

0 The method was successful.

1 The method failed.

Init

```
►►─Init──────────────────────────────────────────────────────────────►◄
```

Creates an instance of the WindowsProgramManager class and loads the required function package.

ShowGroup

```
►►──ShowGroup(─group,─┬─────┬─)──────────────────────────────────────►◄
                      ├─MIN─┤
                      └─MAX─┘
```

Opens a program group. The *group* argument specifies the name of the program group to be opened. If MIN or MAX is specified, the program group is opened minimized or maximized.

Return value:

0	The method was successful.
1	The method failed.

Symbolic Names for Virtual Keys

Table 3 shows the symbolic names and the keyboard equivalents for the virtual keys used by Object REXX.

Table 3. Symbolic Names for Virtual Keys

Symbolic Name	**Mouse or Keyboard Equivalent**
LBUTTON	Left mouse button
RBUTTON	Right mouse button
CANCEL	Control-break processing
MBUTTON	Middle mouse button (three-button mouse)
BACK	BACKSPACE key
TAB	TAB key
CLEAR	CLEAR key
RETURN	ENTER key
SHIFT	SHIFT key
CONTROL	CRTL key
MENU	ALT key
PAUSE	PAUSE key
CAPITAL	CAPS LOCK key
ESCAPE	ESC key
SPACE	SPACEBAR
PRIOR	PAGE UP key
NEXT	PAGE DOWN key
END	END key

Table 3. Symbolic Names for Virtual Keys (continued)

Symbolic Name	Mouse or Keyboard Equivalent
HOME	HOME key
LEFT	LEFT ARROW key
UP	UP ARROW key
RIGHT	RIGHT ARROW key
DOWN	DOWN ARROW key
SELECT	SELECT key
EXECUTE	EXECUTE key
SNAPSHOT	PRINT SCREEN key
INSERT	INS key
DELETE	DEL key
HELP	HELP key
0	0 key
1	1 key
2	2 key
3	3 key
4	4 key
5	5 key
6	6 key
7	7 key
8	8 key
9	9 key
A	A key
B	B key
C	C key
D	D key
E	E key
F	F key
G	G key
H	H key
I	I key
J	J key
K	K key

Table 3. Symbolic Names for Virtual Keys (continued)

Symbolic Name	Mouse or Keyboard Equivalent
L	L key
M	M key
N	N key
O	O key
Q	Q key
R	R key
S	S key
T	T key
U	U key
V	V key
W	W key
X	X key
Y	Y key
Z	Z key
NUMPAD0	Numeric keypad 0 key
NUMPAD1	Numeric keypad 1 key
NUMPAD2	Numeric keypad 2 key
NUMPAD3	Numeric keypad 3 key
NUMPAD4	Numeric keypad 4 key
NUMPAD5	Numeric keypad 5 key
NUMPAD6	Numeric keypad 6 key
NUMPAD7	Numeric keypad 7 key
NUMPAD8	Numeric keypad 8 key
NUMPAD9	Numeric keypad 9 key
MULTIPLY	Multiply key
ADD	Add key
SEPARATOR	Separator key
SUBTRACT	Subtract key
DECIMAL	Decimal key
DIVIDE	Divide key
F1	F1 key
F2	F2 key

Table 3. Symbolic Names for Virtual Keys (continued)

Symbolic Name	Mouse or Keyboard Equivalent
F3	F3 key
F4	F4 key
F5	F5 key
F6	F6 key
F7	F7 key
F8	F8 key
F9	F9 key
F10	F10 key
F11	F11 key
F12	F12 key
F13	F13 key
F14	F14 key
F15	F15 key
F16	F16 key
F17	F17 key
F18	F18 key
F19	F19 key
F20	F20 key
F21	F21 key
F22	F22 key
F23	F23 key
F24	F24 key
NUMLOCK	NUM LOCK key
SCROLL	SCROLL LOCK key

The WindowsRegistry Class

Object REXX provides a class for interaction with the WindowsRegistry. You can use this class to query the registry and modify, add, and delete entries. This class is specifically for Windows systems and may not be available on other systems.

The WindowsRegistry class is not a built-in class; it is defined in the file WINSYSTM.CLS.

Use a `::requires` statement to activate its function:

```
::requires "winsystm.cls"
```

A sample program, `REGISTRY.REX`, is provided in the `OBJREXX\SAMPLES` directory.

Methods the WindowsRegistry Class Defines

CLASSES_ROOT
CLASSES_ROOT=
CLOSE
CREATE
CURRENT_KEY
CURRENT_KEY=
CURRENT_USER
CURRENT_USER=
DELETE
DELETEVALUE
FLUSH
GETVALUE
INIT
LIST
LISTVALUES
LOAD
LOCAL_MACHINE
LOCAL_MACHINE=
OPEN
QUERY
REPLACE
RESTORE
SAVE
SETVALUE
UNLOAD
USERS
USERS=

CLASSES_ROOT

```
►►──CLASSES_ROOT───────────────────────────────────────────────────────►◄
```

Returns the handle of the root key HKEY_CLASSES_ROOT.

CLASSES_ROOT=

```
►►──CLASSES_ROOT=──────────────────────────────────────────────────────►◄
```

This method is used by INIT to set the attribute CLASSES_ROOT to HKEY_CLASSES_ROOT. Do not modify this attribute.

CLOSE

```
►►──CLOSE(──┬────────────┬──)────────────────────────────────────────────►◄
            └─key_handle─┘
```

Closes a previously opened key specified by its handle. Example:

```
rg~close(objectrexxkey)
```

It can take several seconds before all data is written to disk. You can use FLUSH to empty the cache.

If *key_handle* is omitted, CURRENT_KEY is closed.

CONNECT

```
►►──CONNECT(─key,computer──)─────────────────────────────────────────────►◄
```

Opens a key on a remote computer. This is supported only for HKEY_LOCAL_MACHINE and HKEY_USERS.

CREATE

```
►►──CREATE(──┬────────┬──,subkey)────────────────────────────────────────►◄
             └─parent─┘
```

Adds a new named subkey to the registry and returns its handle. The parent key handle *parent* can be a root key or a key retrieved using OPEN. If the *parent* key is omitted, CURRENT_KEY is used. Example:

```
newkey = rg~create(rg~local_machine,'MyOwnKey')
```

CURRENT_KEY

```
►►──CURRENT_KEY──────────────────────────────────────────────────────────►◄
```

Returns the handle of the current key. The current key is set by INIT, CREATE, and OPEN. It is used as a default value if the key is omitted in other methods.

CURRENT_KEY=

```
►►──CURRENT_KEY=──────────────────────────────────────────────────────►◄
```

Sets the handle of the current key.

CURRENT_USER

```
►►──CURRENT_USER──────────────────────────────────────────────────────►◄
```

Returns the handle of the root key HKEY_CURRENT_USER.

CURRENT_USER=

```
►►──CURRENT_USER=─────────────────────────────────────────────────────►◄
```

This method is used by INIT to set the attribute CURRENT_USER to HKEY_CURRENT_USER. Do not modify this attribute.

DELETE

```
►►──DELETE(──┬────────────┬──,subkeyname)─────────────────────────────►◄
             └─key_handle─┘
```

Deletes a given named subkey of an open key specified by its handle and all its subkeys and values. If *key_handle* is omitted, CURRENT_KEY is used.

DELETEVALUE

```
►►──DELETEVALUE(──┬────────────┬──┬────────┬──)───────────────────────►◄
                  └─key_handle─┘  └─,value─┘
```

Deletes the named value for a given key. If *key_handle* is omitted, CURRENT_KEY is used. If *value* is blank or omitted, the default value is deleted.

FLUSH

```
►►──FLUSH(──┬────────────┬──)─────────────────────────────────────────►◄
            └─key_handle─┘
```

Forces the system to write the cache buffer of a given key to disk. If *key_handle* is omitted, CURRENT_KEY is flushed.

GETVALUE

```
>>──GETVALUE(──┬────────────┬──┬────────┬──)─────────────────────────────────><
               └─key_handle─┘  └─,value─┘
```

Retrieves the data and type for a named value of a given key. The result is a compound variable with suffixes data and type. If *key_handle* is omitted, CURRENT_KEY is used. If named *value* is blank or omitted, the default value is retrieved. Example:

```
myval. = rg~GETVALUE(,'filesystem')       /* current key */
say "Type is" myval.type
if myval.type = 'NORMAL' then say "Value is" myval.data
myval. = rg~GETVALUE(mykey)
say "my default value is:" myval.data
myval. = rg~GETVALUE(mykey,'')
say "my default value is:" myval.data
```

Possible types: NORMAL, EXPAND, MULTI, NUMBER, BINARY, NONE, OTHER.

INIT

```
>>──INIT─────────────────────────────────────────────────────────────────────><
```

Creates an instance of the WindowsRegistry class and loads the required external function package. The current key is set to HKEY_LOCAL_MACHINE.

LIST

```
>>──LIST(──┬────────────┬──,stem.)──────────────────────────────────────────><
           └─key_handle─┘
```

Retrieves the list of subkeys for a given key in a stem variable. The name of the stem variable must include the period. The keys are returned as stem.1, stem.2, and so on. Example:

```
rg~LIST(objectrexxkey,orexxkeys.)
do i over orexxkeys.
say orexxkeys.i
end
```

LISTVALUES

```
►►──LISTVALUES(──┬────────────┬──,variable.)─────────────────────────────►◄
                 └─key_handle─┘
```

Retrieves all value entries of a given key into a compound variable. The name of the variable must include the period. The suffixes of the compound variable are numbered starting with 1, and for each number the three values are the name (var.i.name), the data (var.i.data), and the type (var.i.type). The type is NORMAL for alphabetic values, EXPAND for expandable strings such as a path, NONE for no specified type, MULTI for multiple strings, NUMBER for a 4–byte value, and BINARY for any data format.

If *key_handle* is omitted, the values of CURRENT_KEY are listed.

Example:

```
qstem. = rg~QUERY(objectrexxkey)
rg~LISTVALUES(objectrexxkey,lv.)
do i=1 to qstem.values
say "name of value:" lv.i.name "(type="lv.i.type")"
if lv.i.type = 'NORMAL' then
say "data of value:" lv.i.data
end
```

LOAD

```
►►──LOAD(──┬────────────┬──,subkeyname, filename)──────────────────────►◄
           └─key_handle─┘
```

Load creates a named subkey under the open key *key_handle* and loads registry data from the file *filename* (created by SAVE) and stores the data under the newly created subkey.

key_handle can only be HKEY_USERS or HKEY_LOCAL_MACHINE. Registry information is stored in the form of a hive – a discrete body of keys, subkeys, and values that is rooted at the top of the registry hierarchy. A hive is backed by a single file.

If *key_handle* is omitted, the subkey is created under HKEY_LOCAL_MACHINE.

Use UNLOAD to delete the subkey and to unlock the registry data file *filename*.

LOCAL_MACHINE

```
►►──LOCAL_MACHINE─────────────────────────────────────────────────────►◄
```

Returns the handle of the root key HKEY_LOCAL_MACHINE.

LOCAL_MACHINE=

```
►►──LOCAL_MACHINE=────────────────────────────────────────────────────►◄
```

This method is used by INIT to set the attribute LOCAL_MACHINE to HKEY_LOCAL_MACHINE. Do not modify this attribute.

OPEN

```
►►──OPEN(─┬───────────────┬─,subkey─┬──────────────┬─)────────────────►◄
          └─parent_handle─┘         └─,┤ access ├──┘
```

access:

```
   ┌─ALL─┐
├──┴─────┴──────────────────────────────────────────────────────────────►

   ┌──────────────────────────────────────────────────────────┐
   ▼                                                          │
►───┬───────┬──┬──────┬──┬───────┬──┬─────────┬──┬────────┬──┬──────┬─┴───────────┤
    └─WRITE─┘  └─READ─┘  └─QUERY─┘  └─EXECUTE─┘  └─NOTIFY─┘  └─LINK─┘
```

Opens a named subkey and return its handle. See “CREATE” on page 251 for more information about *parent_handle*. Possible values for *access* are:

ALL	Default
WRITE	Create subkeys, set values
READ	Query subkeys and values
QUERY	Values
EXECUTE	Key access, no subkey access
NOTIFY	Change notification
LINK	Create symbolic links

More than one value can be specified separated by blanks.

Notice that on Windows NT some keys require certain access rights and do not allow to open the key with all but only with certain access values.

QUERY

```
►►──QUERY──(─┬────────────┬─)──────────────────────────────────────►◄
             └─key_handle─┘
```

Retrieves information about a given key in a compound variable. The values returned are *class* (class name), *subkeys* (number of subkeys) *values* (number of value entries), *date* and *time* of last modification. If *key_handle* is omitted, CURRENT_KEY is queried. Example:

```
myquery. = rg~QUERY(objectrexxkey)
say "class="myquery.class "at" myquery.date
say "subkeys="myquery.subkeys "values="myquery.values
```

REPLACE

```
►►──REPLACE(─┬────────────┬─,─┬─────────────┬─,newfilename,oldfilename──)───────►◄
             └─key_handle─┘   └─subkeyname──┘
```

Replaces the backup file of a key or subkey with a new file. Key must be an immediate descendant of HKEY_LOCAL_MACHINE or HKEY_USERS. If *key_handle* is omitted, the backup file of CURRENT_KEY is replaced. The values in the new file become active when the system is restarted. If *subkeyname* is omitted, the key and all its subkeys will be replaced.

RESTORE

```
►►──RESTORE(─┬────────────┬─,filename─┬──────────────┬─)──────────────────►◄
             └─key_handle─┘           └─,'VOLATILE'──┘
```

Restores a key from a file. If *key_handle* is omitted, CURRENT_KEY is restored. Example:

```
rg~RESTORE(objectrexxkey,'\objrexx\orexx')
```

The VOLATILE keyword creates a new memory-only set of registry information that is valid only until the system is restarted.

SAVE

```
►►──SAVE(─┬────────────┬─,filename)────────────────────────────────────►◄
          └─key_handle─┘
```

Saves the entries of a given key into a file. If *key_handle* is omitted, CURRENT_KEY is saved. Example:

```
rg~SAVE(objectrexxkey,'\objrexx\orexx')
```

On a FAT system, do not use a file extension in *filename*.

SETVALUE

```
►►──SETVALUE(──┬────────────┬──,──┬──────┬──,value──┬──────────┬──)──────────────────►◄
               └─key_handle─┘     └─name─┘          ├─,NORMAL──┤
                                                    ├─,EXPAND──┤
                                                    ├─,MULTI───┤
                                                    ├─,NUMBER──┤
                                                    ├─,BINARY──┤
                                                    └─,NONE────┘
```

Sets a named value of a given key. If *name* is blank or omitted, the default value is set. Examples:

```
rg~SETVALUE(objectrexxkey,,'My default','NORMAL')
rg~SETVALUE(objectrexxkey,'Product_Name','Object REXX')
rg~SETVALUE(objectrexxkey,'VERSION','1.0')
```

UNLOAD

```
►►──UNLOAD(──┬────────────┬──,subkey)────────────────────────────────────────────────►◄
             └─key_handle─┘
```

Removes a named subkey (created with LOAD) and its dependents from the registry, but does not modify the file containing the registry information. If *key_handle* is omitted, the subkey under CURRENT_KEY is unloaded. Unload also unlocks the registry information file.

USERS

```
►►──USERS────────────────────────────────────────────────────────────────────────────►◄
```

Returns the handle of the root key HKEY_USERS.

USERS=

```
►►──USERS=───────────────────────────────────────────────────────────────────────────►◄
```

This method is used by INIT to set the attribute USERS to HKEY_USERS. Do not modify this attribute.

The WindowsEventLog Class

Object REXX provides a class for interaction with the Windows NT event log. You can use this class to read, write, and clear event-log records. This class is specifically for Windows NT systems and might not be available on other systems.

The WindowsEventLog class is not a built-in class; it is defined in the file `WINSYSTM.CLS`. Use a `::requires` statement to activate its function:

```
::requires "winsystm.cls"
```

A sample program `EventLog.REX` is provided in the `OBJREXX\SAMPLES` directory.

Methods the WindowsEventLog Class Defines
- INIT
- OPEN
- CLOSE
- READ
- WRITE
- CLEAR
- GETNUMBER

INIT

```
►►──INIT──────────────────────────────────────────────────────────────────►◄
```

Creates an instance of the WindowsEventLog class and loads the required function package.

OPEN

```
►►──OPEN──┬──────────────────────────────────┬─────────────────────────────►◄
          └─(──┬──────────────────────────┬──)─┘
               └─server──┬─────────────┬──┘
                         └─,──source───┘
```

Opens the specified event log.

Arguments:

The arguments are:

server The UNC (universal naming convention) name of the server on which the event log is to be opened. If this argument is not specified, the log is opened at the local machine.

source The name of the source of the event log. It must be a subkey of a log file entry under the EventLog key in the registry. If, for example, the registry looks as follows:

```
HKEY_LOCAL_MACHINE
  System
    CurrentControlSet
      Services
      EventLog
        Application
          WinApp1
        Security
          WinApp2
        System
```

"Application", "Security", "System", "WinApp1", and "WinApp2" are valid sources. If the source is not found, the "Application Log" is used.

Example:

The following example opens the Application log on the local machine:

```
rc = event_log1~OPEN
rc = event_log~OPEN(,"Application")
```

The following example opens the System log on SERVER01:

```
rc = event_log~OPEN("\\SERVER01","System")
```

CLOSE

```
►►──CLOSE──────────────────────────────────────────────────────────►◄
```

Closes an open event log.

READ

```
►►──READ──┬──────────────────────────────────────────────────────┬──►
          │    ┌─FORWARDS──┐                                     │
          └─(──┴─BACKWARDS─┴─,──┬────────┬─,──┬────────┬─,───────┘
                                └─server─┘    └─source─┘

►──┬─────────────────────────────┬─────────────────────────────────►◄
   └─┬───────┬─,─┬─────┬─)───────┘
     └─start─┘   └─num─┘
```

Reads event log records. If the event log was not opened with the OPEN method, the event log specified by the *server* and *source* is opened and closed after processing.

Arguments:
The arguments are:

server The UNC (universal naming convention) name of the server on which the event log is to be opened. If this argument is not specified, the log is opened at the local machine.

This argument is only used if the event log was not opened before.

source The name of the source of the event log. It must be a subkey of a log file entry under the EventLog key in the registry. If, for example, the registry looks as follows:

```
HKEY_LOCAL_MACHINE
  System
    CurrentControlSet
      Services
      EventLog
        Application
          WinApp1
        Security
          WinApp2
        System
```

"Application", "Security", "System", "WinApp1", and "WinApp2" are valid sources. If the source is not found, the "Application Log" is used.

This argument is only used if the event log was not opened before.

start The record number of the event log record to be started. The oldest record is always the first record regardless of the direction specified.

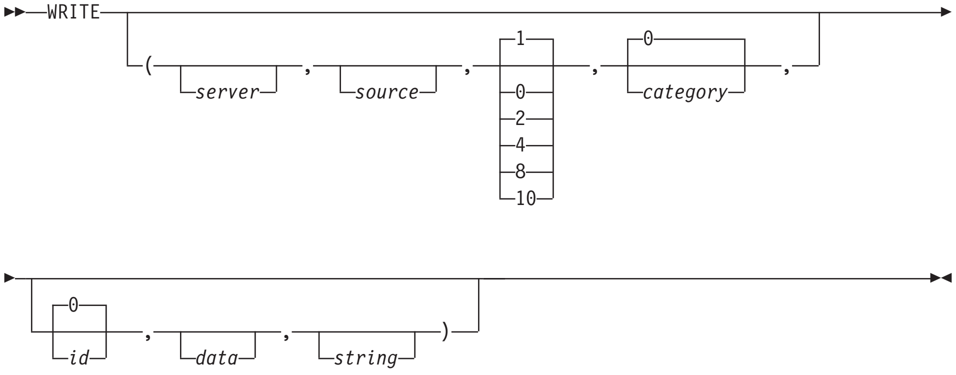
num The number of the event log record to be read.

Example:

```
evl = .WindowsEventLog~new

if evl~InitCode \= 0 then exit

say " reading complete System log forwards without opening it before "

events = evl~Read("FORWARDS", ,"System")

if events \= .nil then
  call DisplayRecords
```

```
else
  say "==> Error reading complete System event log"

evl~deinstall

exit 0   /* leave program */

DisplayRecords:

say evl~Events~items "records read"

do i=1 to evl~Events~items
  say "============================================================="
  temp = evl~Events[i]
  parse var temp type date time "'" sourcename"'" id,
  userid computer "'" string "'" "'" data "'"
  say 'Type : 'type
  say 'Date : 'date
  say 'Time : 'time
  say 'Source : 'sourcename
  say 'ID : 'id
  say 'UserId : 'userid
  say 'Computer : 'computer
  say 'Detail : 'string
  say 'Data  : 'data
end
return

::requires "winsystm.cls"
```

WRITE



Reads event log records. If the event log was not opened with the Open method, the event log specified by the *server* and *source* is opened and closed after processing.

Arguments:

The arguments are:

server The UNC (universal naming convention) name of the server on which the event log is to be opened. If this argument is not specified, the log is opened at the local machine.

source The name of the source of the event log. It must be a subkey of a log file entry under the EventLog key in the registry. If, for example, the registry looks as follows:

```
HKEY_LOCAL_MACHINE
  System
    CurrentControlSet
      Services
      EventLog
        Application
          WinApp1
        Security
          WinApp2
        System
```

"Application", "Security", "System", "WinApp1", and "WinApp2" are valid sources. If the source is not found, the "Application Log" is used.

type The type of the events to be logged. It can be one of the numbers listed in the syntax diagram:

0	SUCCESS
1	ERROR, which is the default
2	WARNING
4	INFORMATION
8	AUDIT SUCCESS
10	AUDIT FAILURE

category
The event category. This is source-specific information that can have any value. The default is 0.

id The event identifier specifying the message that, together with the event, is an entry in the message file associated with the event source. The default is 0.

data The binary data. This is source-specific information and can be omitted.

string The strings merged into the message.

Example:

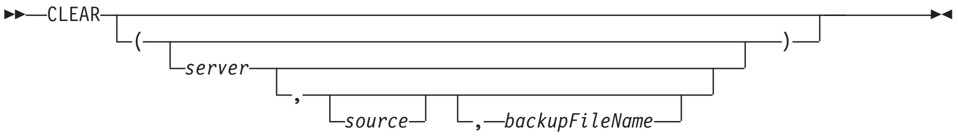
The following example writes the strings and the data to the system log. "MyApplication" must be a subkey of a log file entry under the EventLog key in the registry. If the source name cannot be found, event logging uses the Application log file.

```
HKEY_LOCAL_MACHINE
  System
    CurrentControlSet
      Services
        EventLog
        Application
        Security
          System
            MyApplication
```

```
evl~Write(,"MyApplication", , , ,"1A 1B 1C 0000 00"x,,
"First String", "Second String")
```

See the `EventLog.REX` for more examples.

CLEAR



Clears the specified event log and, optionally, saves a copy of the current log file as a backup file. If the event log was not opened with the Open method, the event log specified by the *server* and *source* is opened. The event log is closed after it is cleared.

Arguments:

The arguments are:

server The UNC (universal naming convention) name of the server on which the event log is to be opened. If this argument is not specified, the log is opened at the local machine.

This argument is only used if the event log was not opened before.

source The name of the source of the event log. It must be a subkey of a log file entry under the EventLog key in the registry. If, for example, the registry looks as follows:

```
HKEY_LOCAL_MACHINE
  System
    CurrentControlSet
      Services
```

```
EventLog
  Application
    WinApp1
  Security
    WinApp2
  System
```

"Application", "Security", "System", "WinApp1", and "WinApp2" are valid sources. If the source is not found, the "Application Log" is used.

This argument is only used if the event log was not opened before.

backupFileName
The name of a file to which the copy of the current event log file is to be written. If this file already exists, the function fails. The file can reside on a remote server.

If you omit this argument, the current event log file is not backed up but cleared.

Example:
The following example creates a backup of the Application event log of the local machine and clears it:

```
event_log~CLEAR(, ,"e:\evlbackup\application.evt")
```

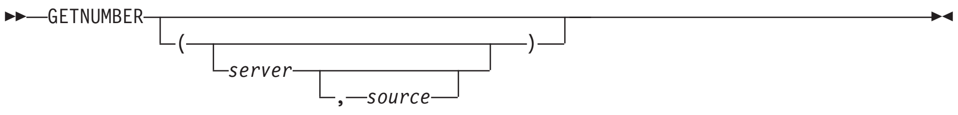
The following example creates a backup of the System event log on the server \\SERVER01 and clears it:

```
event_log~CLEAR("\\SERVER01","System","e:\evlbackup\system_server01.evt")
```

The following example clears the Application log on the local machine without a backup:

```
event_log~CLEAR
```

GETNUMBER



Retrieves the number of records in the specified event log. If the event log was not opened with the Open method, the event log specified by the *server* and *source* is opened and closed after processing.

Arguments:
The arguments are:

server The UNC (universal naming convention) name of the server on which the event log is to be opened. If this argument is not specified, the log is opened at the local machine.

This argument is only used if the event log was not opened before.

source The name of the source of the event log. It must be a subkey of a log file entry under the EventLog key in the registry. If, for example, the registry looks as follows:

```
HKEY_LOCAL_MACHINE
  System
    CurrentControlSet
      Services
      EventLog
        Application
          WinApp1
        Security
          WinApp2
        System
```

"Application", "Security", "System", "WinApp1", and "WinApp2" are valid sources. If the source is not found, the "Application Log" is used.

This argument is only used if the event log was not opened before.

Example:

The following example returns the number of event log records in the Application log of the local machine:

```
num = event_log~GETNUMBER
```

The WindowsManager Class

The WindowsManager class provides methods to query, manipulate, and interact with windows on your desktop. This class is specifically for Windows NT and Windows 2000 systems and might not be available on other systems.

The WindowsManager class is not a built-in class; it is defined in the file `WINSYSTM.CLS`. Use a `::requires` statement to activate its function:

```
::requires "winsystm.cls"
```

Methods the WindowsManager Class Defines

FIND
FOREGROUNDWINDOW
WINDOWATPOSITION
CONSOLETITLE

CONSOLETITLE=
SENDTEXTTOWINDOW
PUSHBUTTONINWINDOW
PROCESSWINDOWCOMMAND

FIND

```
►►──FIND──(──title──)────────────────────────────────────────────►◄
```

Searches for a top-level window (not a child window) on your desktop with the specified *title*.

If this window already exists, an instance of the WindowObject class is returned. Otherwise, .NIL is returned.

FOREGROUNDWINDOW

```
►►──FOREGROUNDWINDOW─────────────────────────────────────────────►◄
```

Returns an instance of the WindowObject class that is associated with the current foreground window.

WINDOWATPOSITION

```
►►──WINDOWATPOSITION──(──x──,──y──)──────────────────────────────►◄
```

Returns an instance of the WindowObject class that is associated with the window at the specified position (*x*,*y*). The coordinates are specified in screen pixels. This method does not retrieve hidden or disabled windows. If you are interested in a particular child window, use method "CHILDATPOSITION" on page 273.

CONSOLETITLE

```
►►──CONSOLETITLE─────────────────────────────────────────────────►◄
```

Returns the title of the current console.

This method does not apply within the Object REXX Workbench or if you start your application with REXXHIDE.

CONSOLETITLE=

```
►►──CONSOLETITLE=title──────────────────────────────────────────────►◄
```

Sets the title of the current console.

This method does not apply within the Object REXX Workbench or if you start your application with REXXHIDE.

SENDTEXTTOWINDOW

```
►►──SENDTEXTTOWINDOW──(──title──,──text──)──────────────────────────►◄
```

Sends a case-sensitive *text* to the window with the specified *title*..

PUSHBUTTONINWINDOW

```
►►──PUSHBUTTONINWINDOW──(──title──,──text──)────────────────────────►◄
```

Selects the button with label *text* in the window with the specified *title*. If the button’s label contains a mnemonic (underscored letter), you must specify an ampersand (&) in front of it. You can also use this method to select radio buttons and to check or uncheck check boxes.

Example:

```
winmgr~PushButtonInWindow("Testwindow","List &Employees")
```

PROCESSMENUCOMMAND

```
►►──PROCESSMENUCOMMAND──(──┬───────┬──,──┬───────┬──,───────────────►
                           └─title─┘     └─popup─┘

   ┌─,───────┐
   ▼         │
►───submenu──┴──,──menuItem──)─────────────────────────────────────►◄
```

Selects an item of the menu or submenu of the specified window *title*. You can specify as many submenus as necessary to get to the required item.

Example:

The following example starts the IBM Resource Workshop when the Object REXX Workbench is running by selecting menu item Resource Workshop of the OODialog submenu in the Tools popup menu:

```
winmgr~ProcessMenuCommand("Object REXX for Windows NT/95 - Output", "T&ools",,
"&OODialog", "&Resource Workshop")
```

The WindowObject Class

The WindowObject class provides methods to query, manipulate, and interact with a particular window or one of its child windows.

Methods the WindowObject Class Defines

- ASSOCWINDOW
- CHILDATPOSITION
- COORDINATES
- DISABLE
- ENABLE
- ENUMERATECHILDREN
- FINDCHILD
- FIRST
- FIRSTCHILD
- FOCUSITEM
- FOCUSNEXTITEM
- FOCUSPREVIOUSITEM
- HANDLE
- HIDE
- ID
- ISMENU
- LAST
- MAXIMIZE
- MENU
- MINIMIZE
- MOVETO
- NEXT
- OWNER
- PREVIOUS
- PROCESSMENUCOMMAND
- PUSHBUTTON
- RESIZE

- RESTORE
- SENDCHAR
- SENDCOMMAND
- SENDKEY
- SENDKEYDOWN
- SENDKEYUP
- SENDMENUCOMMAND
- SENDMESSAGE
- SENDMOUSECLICK
- SENDSYSCOMMAND
- SENDTEXT
- STATE
- SYSTEMMENU
- TITLE
- TITLE=
- TOFOREGROUND
- WCLASS

ASSOCWINDOW

```
►►──ASSOCWINDOW──(──handle──)──────────────────────────────────────►◄
```

Assigns a new window *handle* to the WindowObject instance.

HANDLE

```
►►──HANDLE─────────────────────────────────────────────────────────►◄
```

Returns the handle of the associated window.

TITLE

```
►►──TITLE──────────────────────────────────────────────────────────►◄
```

Returns the title of the window.

TITLE=

```
►►──TITLE=newTitle──────────────────────────────────────────────────►◄
```

Sets a new title for the window.

WCLASS

```
►►──WCLASS──────────────────────────────────────────────────────────►◄
```

Returns the class of the window associated with the WindowObject instance.

ID

```
►►──ID──────────────────────────────────────────────────────────────►◄
```

Returns the numeric ID of the window.

COORDINATES

```
►►──COORDINATES─────────────────────────────────────────────────────►◄
```

Returns the upper left and the lower right corner positions of the window in the format "left,top,right,bottom".

STATE

```
►►──STATE───────────────────────────────────────────────────────────►◄
```

Returns information about the window state. The returned state can contain one or more of the following constants:

- "Enables" or "Disabled"
- "Visible" or "Invisible"
- "Zoomed" or "Minimized"
- "Foreground"

RESTORE

```
►►──RESTORE─────────────────────────────────────────────────────────►◄
```

Activates and displays the associated window. If the window is minimized or maximized, it is restored to its original size and position.

HIDE

```
►►──HIDE──────────────────────────────────────────────────────────────►◄
```

Hides the associated window and activates another window.

MINIMIZE

```
►►──MINIMIZE──────────────────────────────────────────────────────────►◄
```

Minimizes the associated window and activates the next higher-level window.

MAXIMIZE

```
►►──MAXIMIZE──────────────────────────────────────────────────────────►◄
```

Maximizes the associated window.

RESIZE

```
►►──RESIZE──(──width──,──height──)────────────────────────────────────►◄
```

Resizes the associated window to the specified width and height. The width and height are specified in screen coordinates.

ENABLE

```
►►──ENABLE────────────────────────────────────────────────────────────►◄
```

Enables the associated window if it was disabled.

DISABLE

```
►►──DISABLE───────────────────────────────────────────────────────────►◄
```

Disables the associated window.

MOVETO

```
►►──MOVETO──(──x──,──y──)──────────────────────────────────────────►◄
```

Moves the associated window to the specified position (*x*,*y*). Specify the new position in screen pixels.

TOFOREGROUND

```
►►──TOFOREGROUND───────────────────────────────────────────────────►◄
```

Makes the associated window the foreground window.

FOCUSNEXTITEM

```
►►──FOCUSNEXTITEM──────────────────────────────────────────────────►◄
```

Sets the input focus to the next child window of the associated window.

FOCUSPREVIOUSITEM

```
►►──FOCUSPREVIOUSITEM──────────────────────────────────────────────►◄
```

Sets the input focus to the previous child window of the associated window.

FOCUSITEM

```
►►──FOCUSITEM──(──wndObject──)─────────────────────────────────────►◄
```

Sets the input focus to the child window associated with the specified WindowObject instance *wndObject*.

Example:

The following example sets the input focus to the last child window:

```
dlg = wndmgr~Find("TestDialog")
   if dlg \= .Nil then do
       fChild = dlg~FirstChild
       lChild = fChild~Last
       dlg~FocusItem(lChild)
   end
```

FINDCHILD

```
►►──FINDCHILD──(──label──)─────────────────────────────────────────►◄
```

Returns an instance of the WindowObject class associated with the child window with the specified *label*. If the associated window does not own such a window, the .NIL object is returned.

CHILDATPOSITION

```
►►──CHILDATPOSITION──(──x──,──y──)─────────────────────────────────►◄
```

Returns an instance of the WindowObject class associated with the child window at the specified client position (*x*,*y*). The coordinates that are relative to the upper left corner of the associated window must be specified in screen pixels. To retrieve top-level windows, use method "WINDOWATPOSITION" on page 266.

NEXT

```
►►──NEXT───────────────────────────────────────────────────────────►◄
```

Returns an instance of the WindowObject class associated with the next window of the same level as the associated window. If the associated window is the last window of a level, the .NIL object is returned.

PREVIOUS

```
►►──PREVIOUS───────────────────────────────────────────────────────►◄
```

Returns an instance of the WindowObject class associated with the previous window of the same level as the associated window. If the associated window is the first window of a level, the .NIL object is returned.

FIRST

```
►►──FIRST──────────────────────────────────────────────────────────►◄
```

Returns an instance of the WindowObject class associated with the first window of the same level as the associated window.

LAST

```
▸▸──LAST──────────────────────────────────────────────────────────▸◂
```

Returns an instance of the WindowObject class associated with the last window of the same level as the associated window.

OWNER

```
▸▸──OWNER─────────────────────────────────────────────────────────▸◂
```

Returns an instance of the WindowObject class associated with the window that owns the associated window (parent). If the associated window is a top-level window, the .NIL object is returned.

FIRSTCHILD

```
▸▸──FIRSTCHILD────────────────────────────────────────────────────▸◂
```

Returns an instance of the WindowObject class associated with the first child window of the associated window. If no child window exists, the .NIL object is returned.

ENUMERATECHILDREN

```
▸▸──ENUMERATECHILDREN─────────────────────────────────────────────▸◂
```

Returns a stem that stores information about the child windows of the associated window. "Stem.0" contains the number of child windows. The returned stem contains as many records as child windows. The first record is stored at "Stem.1" continued by increments of 1. Each record contains the following entries, where each entry starts with an exclamation mark (!):

!Handle	The handle of the window.
!Title	
!Class	The window class.
!State	
!Coordinates	
!Children	1 if the window has child windows, 0 if is has none.

!Id

Example:

```
wo = winmgr~Find("TestDialog")
  enum. = wo~EnumerateChildren
  do i = 1 to enum.0   /* number of children */
     say "---"
     say "Handle:" enum.i.!Handle
     say "Title:" enum.i.!Title
     say "Class:" enum.i.!Class
     say "Id:" enum.i.!Id
     say "Children:" enum.i.!Children
     say "State:" enum.i.!State
     say "Rect:" enum.i.!Coordinates
  end
```

SENDMESSAGE

```
>>──SENDMESSAGE──(──message──,──wParam──,──lParam──)───────────────────><
```

Sends a message to the associated window.

SENDCOMMAND

```
>>──SENDCOMMAND──(──command──)─────────────────────────────────────────><
```

Sends a WM_COMMAND message to the associated window. WM_COMMAND is sent, for example, when a button is pressed, where *command* is the button ID.

SENDMENUCOMMAND

```
>>──SENDMENUCOMMAND──(──id──)──────────────────────────────────────────><
```

Selects the menu item *id* of the associated window. Method “IDOF” on page 281 returns the ID of a menu item.

SENDMOUSECLICK

```
                       ┌"LEFT"───┐      ┌"DBLCLK"─┐
>>──SENDMOUSECLICK──(──┼"RIGHT"──┼───,──┼"UP"─────┼───"──,──x──,──y──────────>
                       └"MIDDLE"─┘      └"DOWN"───┘
►─┬──────────────────────────────────┬─)──────────────────────────────────────────►◄
  │           ┌──────────────┐       │
  │           ▼              │       │
  └─,─"────┬─LEFTDOWN───┬────┴─"─────┘
           ├─RIGHTDOWN──┤
           ├─MIDDLEDOWN─┤
           ├─SHIFT──────┤
           └─CONTROL────┘
```

Simulates a mouse click event in the associated window.

Arguments:

The arguments are:

which Specifies which mouse button is simulated. LEFT is the default.

kind Selects the simulated mouse action. DBLCLK is the default.

x,y Specifies the coordinates of the mouse click event, in screen coordinates, relative to the upper left corner of the window.

ext Can be one or more of the following strings:

LEFTDOWN
Simulates the pressed left mouse button.

RIGHTDOWN
Simulates the pressed right mouse button.

MIDDLEDOWN
Simulates the pressed middle mouse button.

SHIFT
Simulates the pressed Shift key.

CONTROL
Simulates the pressed Control key.

SENDSYSCOMMAND

```
►►──SENDSYSCOMMAND──(──"──┬─SIZE────────┬──"──)──────────────────────────────►◄
                          ├─MOVE────────┤
                          ├─MINIMIZE────┤
                          ├─MAXIMIZE────┤
                          ├─NEXTWINDOW──┤
                          ├─PREVWINDOW──┤
                          ├─CLOSE───────┤
                          ├─VSCROLL─────┤
                          ├─HSCROLL─────┤
                          ├─ARRANGE─────┤
                          ├─RESTORE─────┤
                          ├─TASKLIST────┤
                          ├─SCREENSAVE──┤
                          └─CONTEXTHELP─┘
```

Sends a WM_SYSCOMMAND message to the associated window. These messages are normally sent when the user selects a command in the Window menu.

Argument:

The only argument is:

command
: One of the commands listed in the syntax diagram:

SIZE	Puts the window in size mode.
MOVE	Puts the window in move mode.
MINIMIZE	Minimizes the window.
MAXIMIZE	Maximizes the window.
NEXTWINDOW	Moves to the next window.
PREVWINDOW	Moves to the previous window.
CLOSE	Closes the window.
VSCROLL	Scrolls vertically.
HSCROLL	Scrolls horizontally.

ARRANGE
Arranges the window.

RESTORE
Restores the window to its normal position and size.

TASKLIST
Activates the Start menu.

SCREENSAVE
Executes the screen-saver application specified in the [boot] section of the SYSTEM.INI file.

CONTEXTHELP
Changes the cursor to a question mark with a pointer. If the user then clicks on a control in the dialog box, the control receives a WM_HELP message.

PUSHBUTTON

```
►►──PUSHBUTTON──(──label──)──────────────────────────────────────────►◄
```

Selects the button with the specified *label* within the associated window and sends the corresponding WM_COMMAND message. If the button's label contains a mnemonic (underscored letter), you must specify an ampersand (&) in front of it. You can also use this method to select radio buttons and check or uncheck check boxes.

SENDKEY

```
►►──SENDKEY──(──keyName─┬───────────────────────┬─)──────────────────►◄
                        └─,─┬─────┬─┬────────┬──┘
                            └─alt─┘ └─,──ext─┘
```

Sends all messages (CHAR, KEYDOWN, and KEYUP) that would be sent by pressing a specific key on the keyboard. Character keys (a to z) are not case-sensitive.

If the *alt* argument is 1, the Alt key flag is set, which is equal to pressing the specified key together with the Alt key.

The *Ext* argument must be 1 if the key is an extended key, such as a right Ctrl or a right Shift.

For a list of key names, refer to “Symbolic Names for Virtual Keys” on page 246.

SENDCHAR

```
►►──SENDCHAR──(──character──┬────────┬──)──────────────────────────►◄
                            └─,──alt─┘
```

Sends a WM_CHAR message to the associated window. If the *alt* argument is 1, a pressed Alt key is simulated.

SENDKEYDOWN

```
►►──SENDKEYDOWN──(──keyName──┬────────┬──)─────────────────────────►◄
                             └─,──ext─┘
```

Sends a WM_KEYDOWN message to the associated window. The *ext* argument must be 1 if the key is an extended key, such as a right Ctrl or a right Shift.

For a list of key names, refer to “Symbolic Names for Virtual Keys” on page 246.

SENDKEYUP

```
►►──SENDKEYUP──(──keyName──┬────────┬──)───────────────────────────►◄
                           └─,──ext─┘
```

Sends a WM_KEYUP message to the associated window. The *ext* argument must be 1 if the key is an extended key, such as a right Ctrl or a right Shift.

For a list of key names, refer to “Symbolic Names for Virtual Keys” on page 246.

SENDTEXT

```
►►──SENDTEXT──(──text──)───────────────────────────────────────────►◄
```

Sends a (case-sensitive) text to the associated window by sending a sequence of WM_CHAR, WM_KEYDOWN, and WM_KEYUP messages.

MENU

```
►►──MENU───────────────────────────────────────────────────────────►◄
```

Returns an instance of the MenuObject class that refers to the menu of the associated window.

SYSTEMMENU

```
>>--SYSTEMMENU------------------------------------------------------------><
```

Returns an instance of the MenuObject class that refers to the system menu of the associated window.

ISMENU

```
>>--ISMENU----------------------------------------------------------------><
```

Returns 1 if the associated window is a menu, otherwise 0.

PROCESSMENUCOMMAND

```
                                  .-,-------.
                                  V         |
>>--PROCESSMENUCOMMAND--(------------,----submenu----,--menuItem--)------><
                         '-menu-'
```

Selects an item of the menu or submenu of the associated window. You can specify as many submenus as necessary to get to the required item.

Example:

The following example starts the IBM Resource Workshop when the Object REXX Workbench is running by selecting menu item Resource Workshop of the OODialog submenu in the Tools popup menu provided that *winobj* is associated with the Object REXX Workbench:

```
winobj~ProcessMenuCommand("T&ools", "&OODialog", "&Resource Workshop")
```

The MenuObject Class

The MenuObject class provides methods to query, manipulate, and interact with the menu or submenu of a window.

Methods the MenuObject Class Defines

- FINDITEM
- FINDSUBMENU

- IDOF
- ISMENU
- ITEMS
- PROCESSITEM
- SUBMENU
- TEXTOF(id)
- TEXTOF(position)

ISMENU

```
►►—ISMENU——————————————————————————————————————————————————————————►◄
```

Returns 1 if the associated window is a menu, otherwise 0.

ITEMS

```
►►—ITEMS———————————————————————————————————————————————————————————►◄
```

Returns the number of menu items contained in the associated menu.

IDOF

```
►►—IDOF—(—position—)———————————————————————————————————————————————►◄
```

Returns the ID of the menu item at the specified *position*, starting with 0.

TEXTOF(position)

```
►►—TEXTOF—(—position—)—————————————————————————————————————————————►◄
```

Returns the text of the menu item at the specified *position*, starting with 0. A mnemonic (underscored letter) is represented by a leading ampersand (&). If the menu item contains an accelerator, it is separated by a tab.

TEXTOF(id)

```
►►—TEXTOF—(—id—)———————————————————————————————————————————————————►◄
```

Returns the text of menu item *id*. A mnemonic is represented by a leading ampersand (&). If the menu item contains an accelerator, it is separated by a tab.

SUBMENU

```
►►──SUBMENU──(──position──)──────────────────────────────────────►◄
```

Returns an instance of the MenuObject class that is associated with the submenu at the specified *position*, starting with 0. If no submenu exists at this position, the .NIL object is returned.

Example:

```
sub = menu~Submenu(5)
    if sub \= .Nil then do
         say "Items:" sub~items
    end
```

FINDSUBMENU

```
►►──FINDSUBMENU──(──label──)─────────────────────────────────────►◄
```

Returns an instance of the MenuObject class that is associated with the submenu with the specified *label*. If the associated menu does not contain such a submenu, the .NIL object is returned.

FINDITEM

```
►►──FINDITEM──(──label──)────────────────────────────────────────►◄
```

Returns the ID of the menu item *label*. If the specified label does not include an accelerator, the comparison excludes the accelerators of the menu items. If no menu item is found that matches the specified label, 0 is returned.

Example:

```
f = menu~FindItem("&Tools" || "9"x || "Ctrl+T")
    if f \= 0 then menu~ProcessItem(f)
```

PROCESSITEM

```
►►──PROCESSITEM──(──id──)────────────────────────────────────────►◄
```

Selects the menu item *id*. This causes a WM_COMMAND to be sent to the window owning the menu.

The WindowsClipboard Class

The WindowsClipboard class provides methods to access the data in the Windows clipboard.

Methods the WindowsClipboard Class Defines
COPY
PASTE
EMPTY
ISDATAAVAILABLE

COPY

```
▶▶──COPY──(──text──)──────────────────────────────────────────────▶◀
```

Empties the clipboard and copies the specified text to it.

PASTE

```
▶▶──PASTE─────────────────────────────────────────────────────────▶◀
```

Retrieves the text data stored on the clipboard.

EMPTY

```
▶▶──EMPTY──────────────────────────────────────────────────────────▶◀
```

Empties the clipboard.

ISDATAAVAILABLE

```
▶▶──ISDATAAVAILABLE────────────────────────────────────────────────▶◀
```

Returns 1 if the text data is available on the clipboard. If no data is available, 0 is returned.

The OLEObject Class

This class provides support for OLE automation. OLE (Object Linking and Embedding) is an implementation of COM (Component Object Model). OLE automation makes it possible for one application to manipulate objects implemented in another application, or to expose objects so they can be manipulated.

An automation client is an application that can manipulate exposed objects belonging to another application. An automation server is an application that exposes the objects. The OLEObject class enables REXX to be an OLE automation client. Note that the OLE acronym has now been replaced by ActiveX.

Applications can provide OLE objects, and OLE objects that support automation can be used by a REXX script to remotely control the object through the supplied methods. This lets you write a REXX script that, for example, starts a Web browser, navigates to a certain page, and changes the display mode of the browser.

Every application that supports OLE places a unique identifier in the registry. This identifier is called the class ID (CLSID) of the OLE object. It consists of several hexadecimal numbers separated by the minus symbol.

Example: CLSID of Microsoft Internet Explorer (Version 5.00.2014.0216):

```
"{0002DF01-0000-0000-C000-000000000046}"
```

The CLSID number can prove inconvenient when you want to create or access a certain object, so a corresponding easy-to-remember entry is provided in the registry, and this entry is mapped to the CLSID. This entry is called the ProgID (the program ID), and is a string containing words separated by periods.

Example: ProgID of Microsoft Internet Explorer:
`"InternetExplorer.Application"`

To find the ProgID of an application, you can use the sample script `OLEINFO.REX` or the Microsoft OLEViewer, or you can consult the documentation of the application or search the registry manually.

The OLEObject class is not a built-in class; it is defined in the file `OREXXOLE.CLS`. This means you have to use a `::requires` statement to activate its functionality, as in:

```
::requires "OREXXOLE.CLS"
```

Several sample programs are provided in the Object REXX installation directory under Samples\OLE.

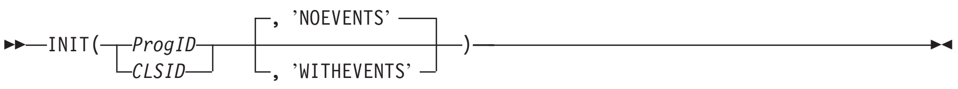
- The APPS directory contains 13 examples of how to use REXX to remote-control other applications.
- The OLEINFO directory is a sample REXX application that can be used to browse through the information an OLE object provides.
- In the ADSI directory there are eight examples of how to use the Active Directory Services Interface with the REXX OLE interface.
- The METHINFO directory contains a very basic example of how to access the information an OLE object provides.
- Finally, the WMI directory contains five examples of how to work with the Windows Management Instrumentation.

Methods available to the OLEObject class:

INIT
GETCONSTANT
GETKNOWNEVENTS
GETKNOWNMETHODS
GETOBJECT(Class method)
GETOUTPARAMETERS
UNKNOWN

Note: The REXX OLE object acts as a proxy to the real OLE object. The OLE object has its own methods, depending on its individual implementation; its methods are accessed transparently through the method mechanism "UNKNOWN" on page 290.

INIT



Instantiates an OLE object of the given ProgID or CLSID. If the creation fails, an error will be raised (see list of OLE specific errors on page 540).

The optional parameter 'events' defines whether events are to be used or not. Allowed values for events are `'NOEVENTS'` (the default) and `'WITHEVENTS'`.

Example:

```
myOLEObject = .OLEObject~new("InternetExplorer.Application")
```

GETCONSTANT

```
>>--GETCONSTANT(--+--------------+--)-----------------------------------><
                  +-ConstantName-+
```

Retrieves the value of a constant that is associated with this OLE object. If no constant of that name exists, the .NIL object will be returned. You can also omit the name of the constant; this returns a stem with all known constants and their values. In this case the constant names will be prefixed with a "!" symbol.

Example 1:

```
myExcel = .OLEObject~new("Excel.Application")
say "xlCenter has the value" myExcel~GetConstant("xlCenter")
myExcel~quit
exit

::requires "orexxole.cls"
```

Possible output:

```
xlCenter has the value -4108
```

Example 2:

```
myExcel = .OLEObject~new("Excel.Application")
constants. = myExcel~GetConstant
myExcel~quit

do i over constants.
  say i"="constants.i
end
```

Possible output:

```
!XLFORMULA=5
!XLMOVE=2
!XLTEXTMAC=19
...
```

GETKNOWNEVENTS

```
>>--GETKNOWNEVENTS------------------------------------------------------><
```

Returns a stem with information on the events that the OLE object can create. It collects this information from the type library of the object. A type library provides the names, types, and arguments of the provided methods.

The stem provides the following information:

stem.0	The number of events.
stem.n.!NAME	Name of n-th event.
stem.n.!DOC	Description of n-th event (if available).
stem.n.!PARAMS.0	Number of parameters for n-th event.
stem.n.!PARAMS.i.!NAME	Name of i-th parameter of n-th event.
stem.n.!PARAMS.i.!TYPE	Type of i-th parameter of n-th event.
stem.n.!PARAMS.i.!FLAGS	Flags of i-th parameter of n-th event; can be ″in″, ″out″, ″opt″, or any combination of these.

If no information is available, the .NIL object is returned and this OLE object does not have any events.

Example script:

```
myIE = .OLEObject~new("InternetExplorer.Application","NOEVENTS")
events. = myIE~GetKnownEvents

if events. == .nil then
  say "Sorry, this object does not have any events."
else do
  say "The following events may occur:"
  do i = 1 to events.0
    say events.i.!NAME
  end
end

exit

::requires "OREXXOLE.CLS"
```

Sample output:

```
The following events may occur:
ONTHEATERMODE
ONFULLSCREEN
ONSTATUSBAR
...
```

For an example of how to use events, see examples OLE\APPS\SAMP12.REX and OLE\APPS\SAMP13.REX in the SAMPLES directory.

GETKNOWNMETHODS

```
►►──GETKNOWNMETHODS──────────────────────────────────────────────────►◄
```

Returns a stem with information on the methods that the OLE object supplies. It collects this information from the type library of the object. A type library provides the names, types, and arguments of the provided methods. Parts of the supplied information have only informational character as you cannot use them directly.

The stem provides the following information:

stem.0	The number of methods.
stem.!LIBNAME	Name of the type library that describes this object.
stem.!LIBDOC	A help string describing the type library. Only set when the string is available.
stem.n.!NAME	The name of the n-th method.
stem.n.!DOC	A help string for the n-th method. If this information is not supplied in the type library this value will not be set.
stem.n.!INVKIND	A number that represents the invocation kind of the method: 1 = normal method call, 2 = `property get`, 4 = `property put`. A normal method call is used with brackets; for a `property get` only the name is to be specified; and a `property set` uses the ″=″ symbol, as in these examples: `object~methodCall(a,b,c)` `object~propertyPut="Hello" say object~propertyGet`
stem.n.!RETTYPE	The return type of the n-th method. The return type will be automatically converted to a REXX object (see ″Type Conversion″ in the description of the UNKNOWN method of the OLEObject class).
stem.n.!MEMID	The MemberID of the n-th method. This is only used internally to call the method.
stem.n.!PARAMS.0	The number of parameters of the n-th method.
stem.n.!PARAMS.i.!NAME	The name of the i-th parameter of the n-th method.
stem.n.!PARAMS.i.!TYPE	The type of the i-th parameter of the n-th method.
stem.n.!PARAMS.i.!FLAGS	The flags of the i-th parameter of the n-th method; can be ″in″, ″out″, ″opt″, or any combination of these (for example: ″[in, opt]″).

If no information is available, the .NIL object is returned.

Note: An object might provide additional methods that cannot be retrieved for display but that can be invoked. In these cases, consult the documentation of those objects.

There are mechanisms to 'hide' methods from the user, because these methods can only be used internally. It might happen that these are not hidden properly and will get displayed. Be careful with methods like:

AddRef
GetTypeInfoCount
GetTypeInfo
GetIDsOfNames
QueryInterface
Release

Example script:

```
myOLEObject = .OLEObject~new("InternetExplorer.Application")
methods. = myOLEObject~GetKnownMethods

if methods. == .nil then
  say "Sorry, no information on the methods available!"
else do
  say "The following methods are available to this OLE object:"
  do i = 1 to methods.0
    say methods.i.!NAME
  end
end

exit

::requires "OREXXOLE.CLS"
```

Sample output:

```
The following methods are available to this OLE object:
GoBack
GoForward
GoHome
...
```

GETOBJECT

```
►►──GETOBJECT(Moniker──┬────────┬──)──────────────────────────────────►◄
                       └─,class─┘
```

This is a class method that allows you to obtain an OLE object through the use of a so-called moniker or nickname (a string). A moniker is used to find out which object has to be created or, if it is already running, addressed. The moniker itself tells OLE which type of object is required. The optional parameter class can be used to specify a subclass of OLEObject, and can be used to obtain an OLE object that supports events (the 'WITHEVENTS' option will be used in this case). This method is similar to the Init method where you have to specify a ProgID or CLSID.

Example:

```
/* create a Word.Document by opening a certain file */
myOLEObject = .OLEObject~GetObject("C:\DOCS\HELLOWORLD.DOC")
```

GETOUTPARAMETERS

```
>>--GETOUTPARAMETERS--------------------------------------------------><
```

Returns an array containing the results of the single out parameters of the OLE object, or the .NIL object if it does not have any. Out parameters are arguments to the OLE object that are filled in by the OLE object. As this is not possible in REXX due to data encapsulation, the results are placed in the array mentioned above.

Example:

Consider an OLE object method with the following signature:

```
aMethod([in] A, [in] B, [out] sumAB)
```

The resulting out parameter of the method invocation will be placed in the out array at position one; the ″normal″ return value gets processed as usual. In this case the method will return the .NIL object:

```
resultTest = myOLEObject~aMethod(1, 2, .NIL)
say "Invocation result  :" resultTest
say "Result in out array:" myOLEObject~GetOutParameters~at(1)
```

The output of this sample script will be:

```
The NIL object
3
```

Out parameters are placed in the out array in order from left to right. If the above OLE method looked like this:

```
aMethod([in] A, [in] B, [out] sumAB, [out] productAB),
```

then the out array would contain the sum of A and B at position one, and the product at position two.

UNKNOWN

```
►►—UNKNOWN(messagename,messageargs)——————————————————————————►◄
```

The UNKNOWN message is the central mechanism through which methods of the OLE object are called.

For further information, see “Defining an UNKNOWN Method” on page 102.

You can invoke the methods of the real OLE object by simply stating their names to the REXX (proxy) OLE object like this:

```
myOLEObject~OLEMethodName
```

This calls the method "OLEMethodName" of the real OLE object for any message that does not exist for the REXX OLE object by dispatching the call to the real OLE object.

If an OLE object offers a method with a name that is identical to one that is defined for OLEObject, you must call UNKNOWN directly, like this:

```
msgs = .array~of("Hello","World")
val = myOLEObject~Unknown("Unknown",msgs)
```

This invokes the method "UNKNOWN" of the OLE object with two arguments, "Hello" and "World".

Parameters for the OLE object are used in the usual way, with the exception of so-called "out" parameters. Out parameters will be filled in by the OLE object itself. As this is not possible in REXX due to data encapsulation, a special method, GETOUTPARAMETERS, has to be used. Specify the .NIL object for any Out parameters when invoking this method.

Type Conversion

Unlike REXX, OLE uses strict typing of data. Conversion to and from these types is done automatically, if conversion is possible. OLE types are called *variants*, because they are stored in one structure that gets flagged with the type it represents. The following is a list of common types that OLE uses and the REXX objects that they are converted into.

VARIANT type	REXX object
VT_EMPTY	.NIL
VT_NULL	.NIL
VT_VOID	.NIL
VT_I1	REXX string (a whole number)
VT_I2	REXX string (a whole number)

VARIANT type	REXX object
VT_I4	REXX string (a whole number)
VT_I8	REXX string (a whole number)
VT_UI1	REXX string (a whole, positive number)
VT_UI2	REXX string (a whole, positive number)
VT_UI4	REXX string (a whole, positive number)
VT_UI8	REXX string (a whole, positive number)
VT_R4	REXX string (a real number)
VT_R8	REXX string (a real number)
VT_CY	REXX string (a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right)
VT_DATE	REXX string
VT_BSTR	REXX string
VT_DISPATCH REXX	OLEObject
VT_BOOL	.TRUE or .FALSE *
VT_VARIANT	Any REXX object that can be represented as a VARIANT
VT_PTR	see VT_VARIANT
VT_SAFEARRAY	REXX Array

* When you pass .TRUE or .FALSE to an OLE object, these get passed as 1 or 0, respectively.

Chapter 7. Other Objects

In addition to the class objects described in the previous chapter, REXX also provides the following objects:

- The Environment object
- The NIL object
- The Local environment object
- The Error object
- The Input object
- The Output object

The Environment Object

The *Environment object* is a directory of public objects that are always accessible. To access the entries of the Environment object, you can use environment symbols.An environment symbol starts with a period and has at least one other character, which cannot be a digit. For example, the term:

```
.method   /* Same as .METHOD */
```

refers to the Method class.

Note: All environment objects that REXX provides are single symbols. Users are recommended to use compound symbols when creating environment objects.

(See “Environment Symbols” on page 35 for details about environment symbols.) REXX provides the following public objects:

.ALARM	The Alarm class. See “The Alarm Class” on page 159.
.ARRAY	The Array class. See “The Array Class” on page 118.
.BAG	The Bag class. See “The Bag Class” on page 124.
.CLASS	The Class class. See “The Class Class” on page 161.
.DIRECTORY	The Directory class. See “The Directory Class” on page 127.
.ENVIRONMENT	The Environment directory.
.FALSE	The FALSE object (the value `0`).
.LIST	The List class. See “The List Class” on page 133.
.LOCAL	The Local environment directory. See “The Local Environment Object (.LOCAL)” on page 294.

.MESSAGE	The Message class. See "The Message Class" on page 170.
.METHOD	The Method class. See "The Method Class" on page 175.
.MONITOR	The Monitor class. See "The Monitor Class" on page 177.
.NIL	The NIL object. See "The NIL Object".
.OBJECT	The Object class. See "The Object Class" on page 179.
.QUEUE	The Queue class. See "The Queue Class" on page 138.
.RELATION	The Relation class. See "The Relation Class" on page 141.
.SET	The Set class. See "The Set Class" on page 147.
.STEM	The Stem class. See "The Stem Class" on page 186.
.STREAM	The Stream class. See "The Stream Class" on page 188.
.STRING	The String class. See "The String Class" on page 206.
.SUPPLIER	The Supplier class. See "The Supplier Class" on page 239.
.TABLE	The Table class. See "The Table Class" on page 150.
.TRUE	The TRUE object (the value 1).

The NIL Object

The *NIL object* is a special object that does not contain data. It usually represents the absence of an object, as a null string represents a string with no characters. It has only the methods of the Object class. Note that you use the .NIL object (rather than the null string ("")) to test for the absence of data in an array entry:

```
if .nil = board[row,col]   /* .NIL rather than "" */
then ...
```

The Local Environment Object (.LOCAL)

The *Local environment object* is a directory of process-specific objects that are always accessible. You can access objects in the Local environment object in the same way as objects in the Environment object. REXX provides the following objects in the Local environment object:

.ERROR	The Error object (default error stream). See "The Error Object" on page 295. This is the object to which REXX error messages and trace output are written.
.INPUT	The Input object (default input stream). See "The Input Object" on page 295.
.OUTPUT	The Output object (default output stream). See "The Output Object" on page 296.

Objects in the Environment object and objects in the Local environment object are available only to programs running within the same process.

Because both of these environment objects are directory objects, you can place objects into, or retrieve objects from, these environments by using any of the directory messages ([],[]=, PUT, AT, SETENTRY, ENTRY, or SETMETHOD). To avoid potential name clashes with built-in objects and public objects that REXX provides, each object that your programs add to these environments should have a period in its index.

Examples:

```
/* .LOCAL example--places something in the Local environment directory */
.local~my.alarm = theAlarm
/* To retrieve it                                                      */
say .local~my.alarm
```

```
/* Another .LOCAL example */
.environment['MYAPP.PASSWORD'] = 'topsecret'
.environment['MYAPP.UID'] = 200
                                       /* Create a local directory for */
                                       /*  my stuff.                   */
.local['MYAPP.LOCAL'] = .directory~new
/* Add log file for my local directory                                 */
.myapp.local['LOG'] = .stream~new('C:\MYAPP.LOG')
say .myapp.password                    /* Displays "topsecret"         */
say .myapp.uid                         /* Displays "200"               */
/* Write a line to the log file */
.myapp.local~log~lineout('Logon at 'time()' on 'date())

/* Redirect SAY lines into a file:  */
.local['OUTPUT'] = .stream~new('C:\SAY_REDIRECT.TXT')
say "This goes into a file, and not onto the screen!"
```

The Error Object

This monitor object (see “The Monitor Class” on page 177) holds the trace stream object. You can redirect the trace output in the same way as with the output object in the Monitor class example.

The Input Object

This monitor object (see “The Monitor Class” on page 177) holds the default input stream object (see “Chapter 16. Input and Output Streams” on page 463). This input stream is the source for the PARSE LINEIN instruction, the LINEIN method of the Stream class, and, if you specify no stream name, the LINEIN built-in function. It is also the source for the PULL and PARSE PULL instructions if the external data queue is empty.

The Output Object

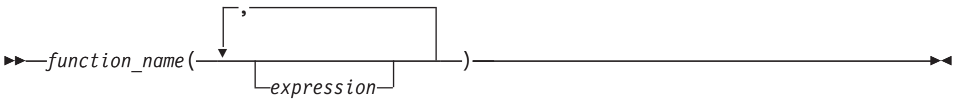
This monitor object (see “The Monitor Class” on page 177) holds the default output stream object (see “Chapter 16. Input and Output Streams” on page 463). This is the destination for output from the SAY instruction, the LINEOUT method (.OUTPUT~LINEOUT), and, if you specify no stream name, the LINEOUT built-in function. You can replace this object in the environment to direct such output elsewhere (for example, to a transcript window).

Chapter 8. Functions

A *function* is an internal, built-in, or external routine that returns a single result object. (A *subroutine* is a function that is an internal, built-in, or external routine that might return a result and is called with the CALL instruction.)

Syntax

A *function call* is a term in an expression calling a routine that carries out some procedures and returns an object. This object 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 following notation:



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

There can be any number of expressions, separated by commas, between the parentheses. These expressions are called the *arguments* to the function. Each argument expression can include further function calls.

Note that the left parenthesis must be adjacent to the name of the function, with no blank in between. (A blank operator would be assumed at this point instead.) Only a comment can appear between the name and the left parenthesis.

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

For example, the function SUBSTR is built into the language processor 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 can have a variable number of arguments.You need to specify only those required. For example, `SUBSTR('ABCDEF',4)` would return `DEF`.

Functions and Subroutines

Functions and subroutines are called in the same way. 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 for a later 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, status information, such as TRACE and NUMERIC settings, is saved too. See the CALL instruction ("CALL" on page 45) for details.

If you call an internal routine as a function, you *must* specify an expression in any RETURN instruction so that the routine can return. This is not necessary if it is called as a subroutine.

Example:

```
/* Recursive internal function execution... */
arg x
say x'! =' factorial(x)
exit
factorial: procedure   /* Calculate factorial by    */
  arg n                /*   recursive invocation.   */
  if n=0 then return 1
  return  factorial(n-1) * n
```

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

Built-in
These functions are always available and are defined in "Built-in Functions" on page 302.

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,

PARSE ARG, or USE ARG instruction or the ARG built-in function can retrieve these argument strings. When called as a function, a program must return data to the caller.

Notes:

1. Calling an external REXX program as a function is similar to calling an internal routine. For an external routine, however, the caller's variables are hidden. To leave the called REXX program, you can use either EXIT or RETURN. In either case, you must specify an expression.
2. You can use the INTERPRET instruction to process a function with a variable function name. However, avoid this if possible because it reduces the clarity of the program.

Search Order

Functions are searched in the following sequence: internal routines, built-in functions, external functions.

The name of internal routines must not be specified as a literal string, that is, in quotation marks, whereas the name of built-in functions or external routines must be specified in quotation marks. Be aware of this when you want to extend the capabilities of an existing internal function, for example, and call it as a built-in function or external routine under the same name as the existing internal function. In this case, you must specify the name in quotation marks.

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. File names can be in uppercase, lowercase, or mixed case. The operating system uses a case-insensitive search for files. When calling a REXX subroutine, the case of the name does not matter.

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

The search order for external functions is as follows:

1. Functions defined on ::ROUTINE directives within the program.

2. Public functions defined on ::ROUTINE directives of programs referenced with ::REQUIRES.
3. Functions that have been loaded into the macrospace for preorder execution. (See the *Object REXX for Windows: Programming Guide* for details.)
4. Functions that are part of a function package. (See the *Object REXX for Windows: Programming Guide* for details.)
5. REXX functions in the current directory, with the current extension.
6. REXX functions along environment PATH, with the current extension.
7. REXX functions in the current directory, with the default extension (.REX or .CMD).
8. REXX functions along environment PATH, with the default extension (.REX or .CMD).
9. Functions that have been loaded into the macrospace for postorder execution.

The full search pattern for functions and routines is shown in Figure 12 on page 301.

Errors during Execution

If an external or built-in function detects an error, 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 might then be possible. If the error is not trapped, the program is ended.

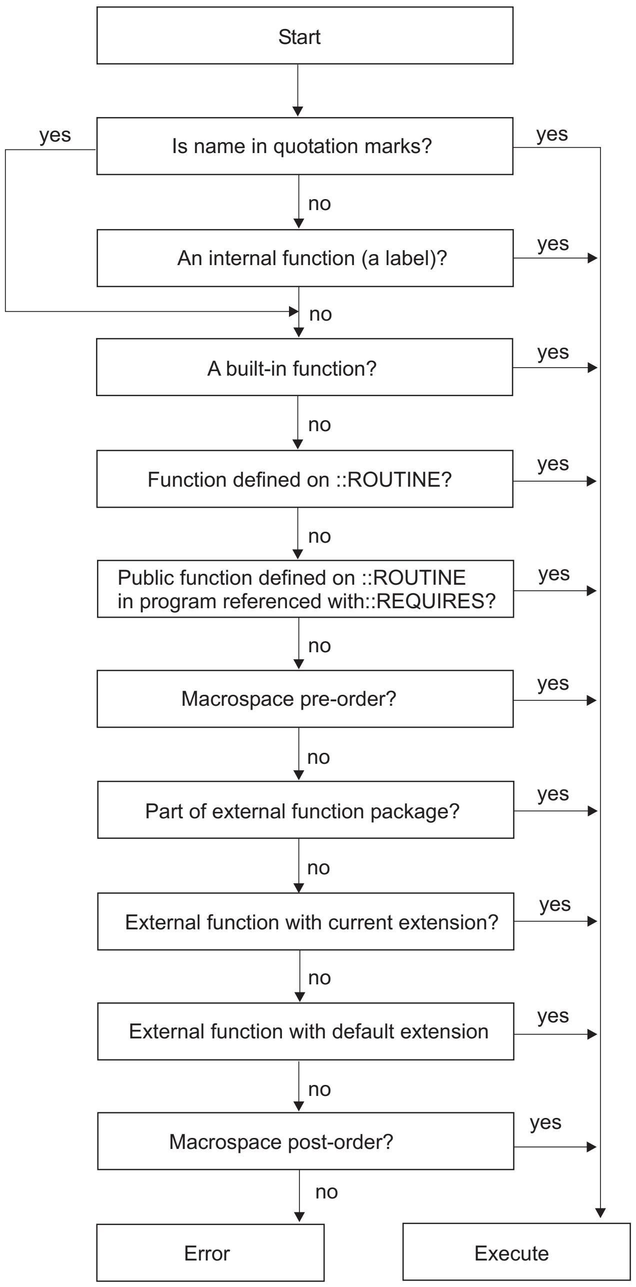

Start
Is name in quotation marks?
yes
yes
no
An internal function (a label)?
yes
no
A built-in function?
yes
no
Function defined on ::ROUTINE?
yes
no
Public function defined on ::ROUTINE in program referenced with::REQUIRES?
yes
no
Macrospace pre-order?
yes
no
Part of external function package?
yes
no
External function with current extension?
yes
no
External function with default extension
yes
no
Macrospace post-order?
yes
no
Error
Execute

Figure 12. Function and Routine Resolution and Execution

Return Values

A function usually returns a value that is substituted for the function call when the expression is evaluated.

How the value returned by a function (or any REXX routine) is handled depends on whether it is called by a function call or as a subroutine with the CALL instruction.

A routine called as a subroutine: If the routine returns a value, that value is stored in the special variable named RESULT. Otherwise, the RESULT variable is dropped, and its value is the string `RESULT`.

A routine called as a function: If the function returns a value, that value is substituted in the expression at the position where the function was called. Otherwise, the language processor stops with an error message.

Here are some examples of how to call a REXX procedure:

```
call Beep 500, 100         /* Example 1: a subroutine call */
```

The built-in function BEEP is called as a REXX subroutine. The return value from BEEP is placed in the REXX special variable `RESULT`.

```
bc = Beep(500, 100)        /* Example 2: a function call   */
```

BEEP is called as a REXX function. The return value from the function is substituted for the function call. The clause itself is an assignment instruction; the return value from the BEEP function is placed in the variable `bc`.

```
Beep(500, 100)             /* Example 3: result passed as  */
                           /*            a command         */
```

The BEEP function is processed and its return value is substituted in the expression for the function call, like in the preceding example. In this case, however, the clause as a whole evaluates to a single expression. Therefore, the evaluated expression is passed to the current default environment as a command.

Note: Many other languages, such as C, throw away the return value of a function if it is not assigned to a variable. In REXX, however, a value returned like in the third example is passed on to the current environment or subcommand handler. If that environment is the default, the operating system performs a disk search for what seems to be a command.

Built-in Functions

REXX provides a set of built-in functions, including character manipulation, conversion, and information functions. The following are general notes on the built-in functions:

- The parentheses in a function are always needed, even if no arguments are required. The first parenthesis must follow the name of the function with no space in between.
- The built-in functions internally work 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 (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.
- Any argument named as a *string* can 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.
- If the last argument is optional, you can always include a comma to indicate that 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. If 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 “LEFT” on page 328.
- If a function has an *option* that you can select by specifying the first character of a string, that character can be in uppercase or lowercase.
- Many of the built-in functions send messages the String class defines (see “The String Class” on page 206). For the functions ABBREV, ABS, BITAND, BITOR, BITXOR, B2X, CENTER, CENTRE, CHANGESTR, COMPARE, COPIES, COUNTSTR, C2D, C2X, DATATYPE, DELSTR, DELWORD, D2C, D2X, FORMAT, LEFT, LENGTH, MAX, MIN, REVERSE, RIGHT, SIGN, SPACE, STRIP, SUBSTR, SUBWORD, TRANSLATE, TRUNC, VERIFY, WORD, WORDINDEX, WORDLENGTH, WORDS, X2B, X2C, and X2D, the first argument to the built-in function is used as the receiver object for the message sent, and the remaining arguments are used in the same order as the message arguments. For example, SUBSTR(‘ABCDE’,3,2) is equivalent to ‘ABCDE’~SUBSTR(3,2).

  For the functions INSERT, LASTPOS, OVERLAY, POS, and WORDPOS, the second argument to the built-in functions is used as the receiver object for the message sent, and the other arguments are used in the same order as the message arguments. For example, `POS('a','Haystack',3)` is equivalent to `'Haystack'~POS('a',3)`.
- The language processor evaluates all built-in function arguments to produce character strings.

ABBREV (Abbreviation)

```
>>──ABBREV(information,info──┬─────────┬──)──────────────────────────────────><
                             └─,length─┘
```

Returns 1 if *info* is equal to the leading characters of *information* and the length of *info* is not less than *length*. It 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 submitted. Trailing blanks are removed from the result.

Here is an example:

```
ADDRESS()    ->    'CMD'        /* default under Windows */
```

ARG (Argument)

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

Returns one or more arguments, or information about the arguments to a program, internal routine, or method.

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. *n* must be a positive whole number.

If you specify *option*, the value returned depends on the value of *option*. The following are valid *option*s. (Only the capitalized letter is needed; all characters following it are ignored.)

Array	returns a single-index array containing the arguments, starting with the *n*th argument. The array indexes correspond to the argument positions, so that the *n*th argument is at index 1, the following argument at index 2, and so on. If any arguments are omitted, their corresponding indexes are absent.
Exists	returns 1 if the *n*th argument exists; that is, if it was explicitly specified when the routine was called. Otherwise, it returns `0`.
Normal	returns the *n*th argument, if it exists, or a null string.
Omitted	returns 1 if the *n*th argument was omitted; that is, if it was *not* explicitly specified when the routine was called. Otherwise, it returns `0`.

Here are some examples:

```
/*  following "Call name;" (no arguments) */
ARG()          ->    0
ARG(1)         ->    ''
ARG(2)         ->    ''
ARG(1,'e')     ->    0
ARG(1,'O')     ->    1
ARG(1,'a')     ->    .array~of()

/*  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
ARG(1,'A')     ->    .array~of(a,,b)
ARG(3,'a')     ->    .array~of(b)
```

Notes:

1. The number of argument strings is the largest number *n* for which `ARG(n,'e')` returns 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 in the command.
3. Programs called by the RexxStart entry point can have several argument strings. (See the *Object REXX for Windows: Programming Guide* for information about RexxStart.)
4. You can access the argument objects of a program with the USE instruction. See "USE" on page 84 for more information.
5. You can retrieve and directly parse the argument strings of a program or internal routine with the ARG or PARSE ARG instructions.

BEEP

```
►►──BEEP(frequency,duration)──────────────────────────────────────►◄
```

Sounds the speaker at *frequency* (Hertz) for *duration* (milliseconds). The *frequency* can be any whole number in the range 37 to 32767 Hertz. The *duration* can be any number in the range 1 to 60000 milliseconds.

This routine is most useful when called as a subroutine. A null string is returned.

Note: Both parameters (frequency, duration) are ignored on Windows 95. On computers with multimedia support the function plays the default sound event. On computers without soundcard, the function plays the standard system beep (if activated).

Here is an example for Windows NT:

```
/* C scale */
note.1 = 262    /* middle C */
note.2 = 294    /*    D     */
note.3 = 330    /*    E     */
note.4 = 349    /*    F     */
note.5 = 392    /*    G     */
note.6 = 440    /*    A     */
note.7 = 494    /*    B     */
note.8 = 523    /*    C     */

do i=1 to 8
call beep note.i,250    /* hold each note for */
/* one-quarter second */
end
```

BITAND (Bit by Bit AND)

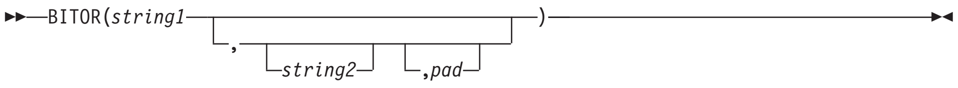
```
►►──BITAND(string1──┬──────────────────────────┬──)──────────────────────────►◄
                    └─,──┬─────────┬──┬──────┬──┘
                         └─string2─┘  └─,pad─┘
```

Returns a string composed of the two input strings logically ANDed, 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',,'DF'x)          ->    'PQRS'      /* ASCII   */
```

BITOR (Bit by Bit OR)

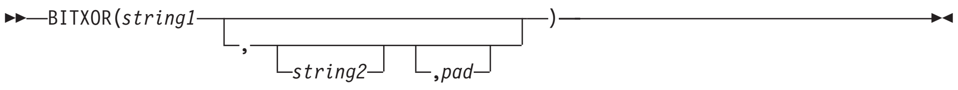


Returns a string composed of the two input strings logically inclusive-ORed, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the OR operation stops when the shorter 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',,'20'x)          ->    'pqrs' /* ASCII   */
```

BITXOR (Bit by Bit Exclusive OR)



Returns a string composed of the two input strings logically eXclusive-ORed, 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,' ')      ->  'E53302'x  /* ASCII  */
```

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 4-digit boundaries only, not leading or trailing) to improve readability; they are ignored.

The returned string uses uppercase alphabetical characters 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 4, then up to three `0` digits are added on the left before the conversion to make a total that is a multiple of 4.

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 (or CENTRE)

```
►►──┬─CENTER(─┬──string,length──┬──────┬──)──────────────────────────►◄
    └─CENTRE(─┘                 └─,pad─┘
```

Returns a string of length *length* with *string* centered in it and 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 is 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.

CHANGESTR

```
►►──CHANGESTR(needle,haystack,newneedle)──────────────────────────────►◄
```

Returns a copy of *haystack* in which *newneedle* replaces all occurrences of *needle*. The following defines the effect:

```
result=''
$tempx=1;
do forever
$tempy=pos(needle,haystack,$tempx)
if $tempy=0 then leave
result=result||substr(haystack,$tempx,$tempy-$tempx)||newneedle
$tempx=$tempy+length(needle)
end
result=result||substr(haystack,$tempx)
```

Here are some examples:

```
CHANGESTR('1','101100','')     ->    '000'
CHANGESTR('1','101100','X')    ->    'X0XX00'
```

CHARIN (Character Input)

```
►►──CHARIN(──┬──────┬──┬──────────────────────────┬──)──────────────►◄
             └─name─┘  └─,──┬───────┬──┬─────────┬─┘
                            └─start─┘  └─,length─┘
```

Returns a string of up to *length* characters read from the character input stream *name*. (To understand the input and output functions, see “Chapter 16. Input and Output Streams” on page 463.) If you omit *name*, characters are read from STDIN, which is the 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. You can give a *start* value to specify an explicit read position. This read position must be positive and within the bounds of the stream, and must not be specified for a transient stream. A value of 1 for *start* refers to the first character in 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.

In a transient stream, if there are fewer than *length* characters available, the execution of the program generally stops until sufficient characters become available. If, however, it is impossible for those characters to become available because of an error or another problem, the NOTREADY condition is raised (see "Errors during Input and Output" on page 472) 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'
```

Notes:

1. CHARIN returns all characters that appear in the stream, including control characters such as line feed, carriage return, and end of file.
2. When CHARIN reads from the keyboard, program execution stops until you press the Enter key.

CHAROUT (Character Output)

```
►►──CHAROUT(──┬──────┬──┬───────────────────────────┬──)──────────────────────►◄
              └─name─┘  └─,──┬────────┬──┬────────┬─┘
                             └─string─┘  └─,start─┘
```

Returns the count of characters remaining after attempting to write *string* to the character output stream *name*. (To understand the input and output functions, see "Chapter 16. Input and Output Streams" on page 463.) If you omit *name*, characters in *string* are written to STDOUT (generally the display), which is the default output stream. The *string* can be a null string, in which case no characters are written to the stream, and `0` is always returned.

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 append characters to the end of the stream.

You can give a *start* value to specify an explicit write position for a persistent stream. This write position must be a positive whole number. A value of 1 for *start* refers to the first character in the stream.

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*, the stream is closed and 0 is returned.

Execution of the program usually stops until the output operation is 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 might not have been printed. 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 472) and CHAROUT returns with the number of characters that could not be written (the residual count).

Here are some examples:

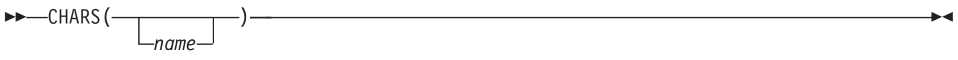
```
CHAROUT(myfile,'Hi')     ->   0   /* typically        */
CHAROUT(myfile,'Hi',5)   ->   0   /* typically        */
CHAROUT(myfile,,6)       ->   0   /* now at char 6    */
CHAROUT(myfile)          ->   0   /* at end of stream */
CHAROUT(,'Hi')           ->   0   /* typically        */
CHAROUT(,'Hello')        ->   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,'Hi',6
Call CHAROUT myfile
```

CHARS (Characters Remaining)



Returns the total number of characters remaining in the character input stream *name*. The count includes any line separator characters, if these are defined for the stream. In the case of persistent streams, it is the count of characters from the current read position. (See “Chapter 16. Input and Output Streams” on page 463 for a discussion of REXX input and output.) If you omit *name*, the number of characters available in the default input stream (STDIN) is returned.

The total number of characters remaining cannot be determined for some streams (for example, STDIN). For these streams, the CHARS function returns `1` to indicate that data is present, or `0` if no data is present. For windows devices, CHARS always returns `1`.

Here are some examples:

```
CHARS(myfile)     ->  42   /* perhaps */
CHARS(nonfile)    ->  0
CHARS()           ->  1    /* perhaps */
```

COMPARE

```
►►──COMPARE(string1,string2──┬──────┬──)──────────────────────────────►◄
                             └─,pad─┘
```

Returns `0` if the strings *string1* and *string2* are identical. Otherwise, it 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

```
►►──CONDITION(──┬────────┬──)─────────────────────────────────────────►◄
                └─option─┘
```

Returns the condition information associated with the current trapped condition. (See “Chapter 12. Conditions and Condition Traps” on page 429 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

- Any condition-specific information associated with the current trapped condition
- The instruction processed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition

In addition, you can request a condition object containing all of the preceding information.

To select the information to be returned, use the following *option*s. (Only the capitalized letter is needed; all characters following it are ignored.)

Additional
returns any additional object information associated with the current trapped condition. See "Additional Object Information" on page 436 for a list of possible values. If no additional object information is available or no condition has been trapped, the language processor returns the NIL object.

Condition name
returns the name of the current trapped condition. For user conditions, the returned string is a concatenation of the word `USER` and the user condition name, separated by a blank.

Description
returns any descriptive string associated with the current trapped condition. See "Descriptive Strings" on page 435 for the list of possible values. If no description is available or no condition has been trapped, it 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*. If no condition has been trapped, it returns a null string.

Object
returns an object that contains all the information about the current trapped condition. See "Chapter 12. Conditions and Condition Traps" on page 429 for more information. If no condition has been trapped, it returns the NIL object.

Status
returns the status of the current trapped condition. This can change during processing, and is one of the following:
`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, a null string is returned.

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 can 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)    ->    ''
```

COUNTSTR

```
►►──COUNTSTR(needle,haystack)─────────────────────────────────────►◄
```

Returns a count of the occurrences of *needle* in *haystack* that do not overlap. The following defines the effect:

```
result=0
$tempx=pos(needle,haystack)
do while $temp > 0
result=result+1
$temp=pos(needle,haystack,$temp+length(needle))
end
```

Here are some examples:

```
COUNTSTR('1','101101')       ->    4
COUNTSTR('KK','J0KKK0')      ->    1
```

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 specify *n*, it is the length of the returned result. If you do not specify *n*, *string* is processed as an unsigned binary number.

If *string* is null, `0` is returned.

Here are some examples:

```
C2D('09'X)      ->       9
C2D('81'X)      ->     129
C2D('FF81'X)    ->   65409
C2D('')         ->       0
C2D('a')        ->      97     /*  ASCII   */
```

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 if the leftmost bit is on. In both cases, it is converted to a whole number, which can be negative. The *string* is padded on the left with '00'x characters (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
```

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. On an ASCII system, C2X(1) returns 31 because the ASCII representation of the character 1 is '31'X.

The string returned uses uppercase alphabetical characters for the values A–F and does not include blanks. The *string* can be of any length. If *string* is null, a null string is returned.

Here are some examples:

```
C2X('0123'X)    ->    '0123'   /* '30313233'X     in ASCII */
C2X('ZD8')      ->    '5A4438' /* '354134343338'X in ASCII */
```

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*, it returns 1 if *string* matches the type. Otherwise, it returns 0. If *string* is null, the function returns 0 (except when the *type* is X or B, for which DATATYPE returns 1 for a null string). The following are valid *type*s. (Only the capitalized letter, or the number of the last type listed, 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 character 0 or 1, or a blank. Blanks can appear only between groups of 4 binary characters. It also returns 1 if *string* is a null string, which is a valid binary string.
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 DATATYPE(string) returns NUM.
Symbol	returns 1 if *string* is a valid symbol, that is, if SYMBOL(*string*) does not return BAD. (See "Symbols" on page 14.) Note that both uppercase and lowercase alphabetics are permitted.
Uppercase	returns 1 if *string* contains only characters from the range A–Z.
Variable	returns 1 if *string* could appear on the left-hand side of an assignment without causing a SYNTAX condition.

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). It also returns 1 if *string* is a null string, which is a valid hexadecimal string.

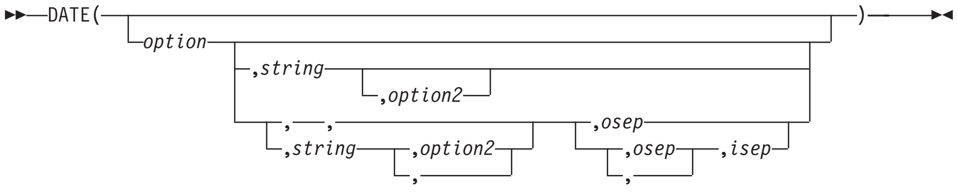
9 digits
: returns 1 if `DATATYPE(string,'W')` returns 1 when NUMERIC DIGITS is set to `9`.

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



Returns, by default, the local date in the format: *dd mon yyyy* (day month year—for example, `13 Nov 1998`), with no leading zero or blank on the day. The first three characters of the English name of the month are used.

You can use the following *option*s to obtain specific formats. (Only the capitalized letter is needed; all characters following it are ignored.)

Base
: returns 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.

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.

Days returns the number of days, including the current day, that have passed this year in the format *ddd* (no leading zeros or blanks).

European returns the date in the format dd/mm/yy.

Language returns the date in an implementation- and language-dependent, or local, date format. The format is *dd month yyyy*. The name of the month is according to the national language installed on the system. If no local date format is available, the default format is returned.

Note: This format is intended to be used as a whole; REXX programs must not make any assumptions about the form or content of the returned string.

Month returns the full English name of the current month, for example, `August`.

Normal returns the date in the format *dd mon yyyy*. This is the default.

Ordered returns the date in the format *yy/mm/dd* (suitable for sorting, for example).

Standard returns the date in the format *yyyymmdd* (suitable for sorting, for example).

Usa returns the date in the format *mm/dd/yy*.

Weekday returns the English name for the day of the week, in mixed case, for example, `Tuesday`.

Here are some examples, assuming today is 13 November 1996:

```
DATE()         ->    '13 Nov 1996'
DATE('B')      ->    728609
DATE('D')      ->    317
DATE('E')      ->    '13/11/96'
DATE('L')      ->    '13 November 1996'
DATE('M')      ->    'November'
DATE('N')      ->    '13 Nov 1996'
DATE('O')      ->    '96/11/13'
DATE('S')      ->    '19961113'
DATE('U')      ->    '11/13/96'
DATE('W')      ->    'Monday'
```

Note: 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, several calls to any of the DATE or TIME functions, or both, in a single expression or clause are consistent with each other.

If you specify *string*, DATE returns the date corresponding to *string* in the format *option*. The *string* must be supplied in the format *option2*. The *option2* format must specify day, month, and year (that is, not 'D', 'L', 'M', or 'W'). The default for *option2* is 'N', so you need to specify *option2* if *string* is not in the **N**ormal format. Here are some examples:

```
DATE('O','13 Feb 1923')     ->    '23/02/13'
DATE('O','06/01/50','U')    ->    '50/06/01'
```

If you specify an output separator character *osep*, the days, month, and year returned are separated by this character. Any nonalphanumeric character or an empty string can be used. A separator character is only valid for the formats 'E', 'N', 'O', 'S', and 'U'. Here are some examples:

```
DATE('S','13 Feb 1996','N','-')     ->    '1996-02-13'
DATE('N','13 Feb 1996','N','')      ->    '13Feb1996'
DATE('N','13 Feb 1996','N','-')     ->    '13-Feb-1996'
DATE('O','06/01/50','U','')         ->    '500601'
DATE('E','02/13/96','U','.')        ->    '13.02.96'
DATE('N',,,'_')                     ->    '26_Mar_1998'  (today)
```

In this way, formats can be created that are derived from their respective default format, which is the format associated with *option* using its default separator character. The default separator character for each of these formats is:

```
Option       Default separator

European     '/'
Normal       ' '
Ordered      '/'
Standard     ''  (empty string)
Usa          '/'
```

If you specify a *string* containing a separator that is different from the default separator character of *option2*, you must also specify *isep* to indicate which separator character is valid for *string*. Basically, any date format that can be generated with any valid separator character can be used as input date *string* as long as its format has the generalized form specified by *option2* and its separator character matches the character specified by *isep*.

Here are some examples:

```
DATE('S','1996-11-13','S',,'','-')     ->    '19961113'
DATE('S','13-Nov-1996','N','','-')     ->    '19961113'
DATE('O','06*01*50','U','','*')        ->    '500601'
DATE('U','13.Feb.1996','N',,'.')       ->    '02/13/96'
```

You can determine the number of days between two dates; for example:

```
say date('B','12/25/96','U')-date('B') " shopping days till Christmas!"
```

If *string* does not include the century but *option* defines that the century be returned as part of the date, the century is determined depending on whether the year to be returned is within the past 50 years or the next 49 years. Assume, for example, that you specify 10/15/43 for *string* and today's date is 10/27/1998. In this case, 1943 would be 55 years ago and 2043 would be 45 years in the future. So, 10/15/2043 would be the returned date.

Note: This rule is suitable for dates that are close to today's date. However, when working with birth dates, it is recommended that you explicitly provide the century.

DELSTR (Delete String)

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

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. *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)

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

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. *n* must be a positive whole number. If *n* is greater than the number of words in *string*, the function returns *string* unchanged. The string deleted includes any blanks following the final word involved but none of the blanks preceding the first word involved.

Here are some examples:

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

DIGITS

```
►►──DIGITS()─────────────────────────────────────────────────────────►◄
```

Returns the current setting of NUMERIC DIGITS. See “NUMERIC” on page 62 for more information.

Here is an example:

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

DIRECTORY

```
►►──DIRECTORY(──┬──────────────┬──)──────────────────────────────────►◄
                └─newdirectory─┘
```

Returns the current directory, changing it to *newdirectory* if an argument is supplied and the named directory exists. If *newdirectory* is not specified, the name of the current directory is returned. Otherwise, an attempt is made to change to the specified *newdirectory*. If successful, the name of the *newdirectory* is returned; if an error occurred, null is returned.

For example, the following program fragment saves the current directory and switches to a new directory; it performs an operation there, and then returns to the former directory.

```
/* get current directory      */
curdir = directory()
/* go play a game             */
newdir = directory("/usr/bin")
if newdir = "/usr/games" then
  do
  fortune   /* tell a fortune */
/* return to former directory */
```

```
  call directory curdir
end
else
  say 'Can't find /usr/games'
```

D2C (Decimal to Character)

```
►►──D2C(wholenumber──┬────┬──)──────────────────────────────────────────►◄
                     └─,n─┘
```

Returns a string, in character format, that is the ASCII representation of the decimal number. If you specify *n*, it is the length of the final result in characters; leading blanks are added to the output character. *n* must be a positive whole number or zero.

Wholenumber must not have more digits than the current setting of NUMERIC DIGITS.

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(65)      ->   'A'      /* '41'x is an ASCII 'A'    */
D2C(65,1)    ->   'A'
D2C(65,2)    ->   ' A'
D2C(65,5)    ->   '    A'
D2C(109)     ->   'm'      /* '6D'x  is an ASCII 'm'   */
D2C(-109,1)  ->   'ô'      /* '93'x is an ASCII 'ô'    */
D2C(76,2)    ->   ' L'     /* '4C'x  is an ASCII ' L'  */
D2C(-180,2)  ->   ' L'
```

Implementation maximum: The output string must not have more than 250 significant characters, although it can be longer if it contains 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.

Wholenumber must not have more digits than the current setting of NUMERIC DIGITS.

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 *n* characters, it is truncated on the left. *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 must not have more than 500 significant hexadecimal characters, although it can be longer if it contains leading sign characters (0 and F).

ERRORTEXT

```
►►──ERRORTEXT(n)──────────────────────────────────────────────────────►◄
```

Returns the REXX error message associated with error number *n*. *n* must be in the range 0–99. It returns the null string if *n* is in the allowed range but is not a defined REXX error number. See “Appendix C. Error Numbers and Messages” on page 527 for a complete description of error numbers and messages.

Here are some examples:

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

FILESPEC

```
►►──FILESPEC(option,filespec)─────────────────────────────────────────►◄
```

Returns a selected element of *filespec*, a given file specification, identified by one of the following strings for *option*:

Drive The drive letter of the given *filespec*.

Path The directory path of the given *filespec*.

Name The file name of the given *filespec*.

If the requested string is not found, then FILESPEC returns a null string ("").

Note: Only the initial letter of *option* is needed.

Here are some examples:

```
thisfile = "C:\WINDOWS\UTIL\SYSTEM.INI"
say FILESPEC("drive",thisfile)     /* says "C:"              */
say FILESPEC("path",thisfile)      /* says "\WINDOWS\UTIL\"  */
say FILESPEC("name",thisfile)      /* says "SYSTEM.INI"      */
part = "name"
say FILESPEC(part,thisfile)        /* says "SYSTEM.INI"      */
```

FORM

```
►►──FORM()──────────────────────────────────────────────────────────►◄
```

Returns the current setting of NUMERIC FORM. See “NUMERIC” on page 62 for more information.

Here is an example:

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

FORMAT

```
►►──FORMAT(number───────────────────────────────────────────────────►

►──┬────────────────────────────────────────────────────────┬──)────►◄
   └─,──┬────────┬──┬─────────────────────────────────────┬─┘
        └─before─┘  └─,──┬───────┬──┬───────────────────┬─┘
                         └─after─┘  └─,──┬──────┬──┬───────┬─┘
                                         └─expp─┘  └─,expt─┘
```

Returns *number*, rounded and formatted.

The *number* is first rounded according to standard REXX rules, 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 described in the following.

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 them, 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. *expp* sets the number of places for the exponent part; the default is to use as many as needed (which can be zero). *expt* specifies when the exponential expression is used. The default is the current setting of NUMERIC DIGITS.

If *expp* is `0`, the number is not in exponential notation. 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, the exponential notation is used. If *expt* is `0`, the 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, the number is not an exponential expression.

Here are some examples:

```
FORMAT('12345.73',,,2,2)    ->    '1.234573E+04'
FORMAT('12345.73',,3,,0)    ->    '1.235E+4'
FORMAT('1.234573',,3,,0)    ->    '1.235'
FORMAT('12345.73',,,3,6)    ->    '12345.73'
FORMAT('1234567e5',,3,0)    ->    '123456700000.000'
```

FUZZ

```
►►──FUZZ()─────────────────────────────────────────────────────────►◄
```

Returns the current setting of NUMERIC FUZZ. See “NUMERIC” on page 62 for more information.

Here is an example:

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

INSERT

```
►►──INSERT(new,target──┬───────────────────────────────────────┬──)───────►◄
                       └─,──┬───┬──┬─────────────────────────┬─┘
                            └─n─┘  └─,──┬────────┬──┬──────┬─┘
                                        └─length─┘  └─,pad─┘
```

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 insertion 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, the string *new* is padded at the beginning. 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'
```

LASTPOS (Last Position)

```
►►──LASTPOS(needle,haystack──┬────────┬──)──────────────────────────►◄
                             └─,start─┘
```

Returns the position of the last occurrence of one string, *needle*, in another, *haystack*. (See also “POS (Position)” on page 333.) It returns `0` if *needle* is a null string or 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

```
>>──LEFT(string,length──┬──────┬──)──────────────────────────────────────><
                        └─,pad─┘
```

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:

```
>>──SUBSTR(string,1,length──┬──────┬──)──────────────────────────────────><
                            └─,pad─┘
```

Here are some examples:

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

LENGTH

```
>>──LENGTH(string)───────────────────────────────────────────────────────><
```

Returns the length of *string*.

Here are some examples:

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

LINEIN (Line Input)

```
>>──LINEIN(──┬──────┬──┬───────────────────────────┬──)──────────────────><
             └─name─┘  └─,──┬──────┬──┬────────┬───┘
                            └─line─┘  └─,count─┘
```

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 16. Input and Output Streams" on page 463.) If you omit *name*, the line is read from the default input stream, STDIN. 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 moved to the beginning of the next line. The read position can be set to the beginning of the stream by giving *line* a value of 1.

If you give a *count* of `0`, then no characters are read and a 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 another problem, the NOTREADY condition is raised (see "Errors during Input and Output" on page 472) and LINEIN returns whatever characters are available.

Here are some examples:

```
LINEIN()                            /* Reads one line from the    */
                                    /* default input stream;      */
                                    /* usually this is an entry   */
                                    /* typed at the keyboard      */
myfile = 'ANYFILE.TXT'
LINEIN(myfile)     -> 'Current line' /* Reads one line from        */
                                    /* ANYFILE.TXT, beginning     */
                                    /* at the current read        */
                                    /* position. (If first call,  */
                                    /* file is opened and the     */
                                    /* first line is read.)       */

LINEIN(myfile,1,1) ->  'first line'  /* Opens and reads the first  */
                                    /* line of ANYFILE.TXT (if    */
                                    /* the file is already open,  */
                                    /* reads first line); sets    */
                                    /* read position on the       */
                                    /* second line.               */

LINEIN(myfile,1,0) ->  ''            /* No read; opens ANYFILE.TXT */
                                    /* (if file is already open,  */
                                    /* sets the read position to  */
                                    /* the first line).           */

LINEIN(myfile,,0)  ->  ''            /* No read; opens ANYFILE.TXT */
                                    /* (no action if the file is  */
                                    /* already open).             */

LINEIN("QUEUE:")   ->  'Queue line'  /* Read a line from the queue. */
                                    /* If the queue is empty, the  */
                                    /* program waits until a line  */
                                    /* is put on the queue.        */
```

Note: If you want to read complete lines from the default input stream, as in a dialog with a user, use the PULL or PARSE PULL instruction.

The PARSE LINEIN instruction is also useful in certain cases. (See page 64.)

LINEOUT (Line Output)

```
►►──LINEOUT(──┬──────┬──┬────────────────────────────┬──)────────────────────────►◄
              └─name─┘  └─,──┬────────┬──┬───────┬───┘
                             └─string─┘  └─,line─┘
```

Returns 0 if successful in writing *string* to the character output stream *name*, or 1 if an error occurs while writing the line. (To understand the input and output functions, see "Chapter 16. Input and Output Streams" on page 463.) If you omit *string* but include *line*, only the write position is repositioned. If *string* is a null string, LINEOUT repositions the write position (if you include *line*) and does a carriage return. Otherwise, the stream is closed. LINEOUT adds a line-feed and a carriage-return character to the end of *string*.

If you omit *name*, the line is written to the default output stream STDOUT (usually the display).

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 can 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 set to the beginning of the line following the one just written. When the stream is first opened, the write position is at the end of the stream, so that calls to LINEOUT 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 unless it is a binary 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. Note that, unlike CHAROUT, you cannot specify a position beyond the end of the stream for non-binary streams.

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*, the stream is closed. Again, the function returns 0.

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 might not have been printed. 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 472), and LINEOUT returns a result of 1, that is, the residual count of lines written.

Here are some examples:

```
LINEOUT(,'Display this')          /* Writes string to the default   */
                                  /* output stream (usually, the    */
                                  /* display); returns 0 if         */
                                  /* successful                     */

myfile = 'ANYFILE.TXT'
LINEOUT(myfile,'A new line')      /* Opens the file ANYFILE.TXT and */
                                  /* appends the string to the end. */
                                  /* If the file is already open,   */
                                  /* the string is written at the   */
                                  /* current write position.        */
                                  /* Returns 0 if successful.       */

LINEOUT(myfile,'A new start',1)   /* Opens the file (if not already */
                                  /* open); overwrites first line   */
                                  /* with a new line.               */
                                  /* Returns 0 if successful.       */

LINEOUT(myfile,,1)                /* Opens the file (if not already */
                                  /* open). No write; sets write    */
                                  /* position at first character.   */

LINEOUT(myfile)                   /* Closes ANYFILE.TXT             */
```

LINEOUT is often most useful when called as a subroutine. The return value is then available in the variable RESULT. For example:

```
Call LINEOUT 'A:rexx.bat','Shell',1
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)

```
►►──LINES(──┬──────┬──)──────────────────────────────────────────►◄
            └─name─┘
```

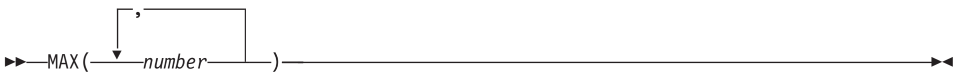
Returns 1 if any data remains between the current read position and the end of the character input stream *name*. It returns 0 if no data remains. In effect, LINES reports whether a read action that CHARIN (see “CHARIN (Character Input)” on page 310) or LINEIN (see “LINEIN (Line Input)” on page 328) performs will succeed. (To understand the input and output functions, see “Chapter 16. Input and Output Streams” on page 463.)

Here are some examples:

```
LINES(myfile)    ->    0    /* at end of the file   */
LINES()          ->    1    /* data remains in the  */
                            /* default input stream */
                            /* STDIN:               */
```

Note: The CHARS function returns the number of characters in a persistent stream or the presence of data in a transient stream.

MAX (Maximum)

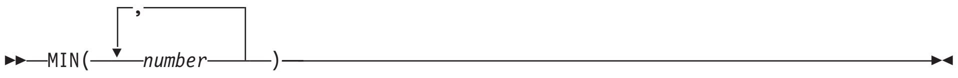


Returns the largest number of the list specified, formatted according to the current NUMERIC settings. You can specify any number of *number*s.

Here are some examples:

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

MIN (Minimum)



Returns the smallest number of the list specified, formatted according to the current NUMERIC settings. You can specify any number of *number*s.

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

OVERLAY

```
>>──OVERLAY(new,target──┬────────────────────────────────────────┬──)──────────><
                        └─,──┬───┬──┬──────────────────────────┬─┘
                             └─n─┘  └─,──┬────────┬──┬──────┬──┘
                                         └─length─┘  └─,pad─┘
```

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 must 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, the string *new* is padded at the beginning. 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)

```
>>──POS(needle,haystack──┬────────┬──)─────────────────────────────────><
                         └─,start─┘
```

Returns the position of one string, *needle*, in another, *haystack*. (See also "LASTPOS (Last Position)" on page 327.) It returns `0` if *needle* is a null string or 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

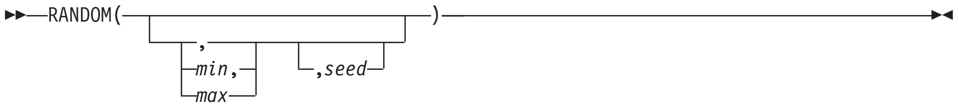
```
>>──QUEUED()───────────────────────────────────────────────────────────><
```

Returns the number of lines remaining in the external data queue when the function is called. (See "Chapter 16. Input and Output Streams" on page 463 for a discussion of REXX input and output.)

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,max*, then *max* minus *min* cannot exceed 100000. *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. This *seed* must be a positive whole number from 0 to 999999999.

Here are some examples:

```
RANDOM()          ->    305
RANDOM(5,8)       ->      7
RANDOM(2)         ->      0  /*  0  to  2    */
RANDOM(,,1983)    ->    123  /* reproducible */
```

Notes:

1. To obtain a predictable sequence of quasi-random numbers, use RANDOM a number of times, but specify a *seed* only the first time. For example, to simulate 40 throws of a 6-sided, unbiased die:

   ```
   sequence = RANDOM(1,6,12345)  /* any number would */
                                 /* do for a seed    */
   do 39
   sequence = sequence RANDOM(1,6)
   end
   say sequence
   ```

   The numbers are generated mathematically, using the initial *seed*, so that as far as possible they appear to be random. Running the program again produces the same sequence; using a different initial *seed* almost certainly produces a different sequence. If you do not supply a *seed*, the first time RANDOM is called, an arbitrary seed is used. Hence, your program usually gives different results each time it is run.
2. The random number generator is global for an entire program; the current seed is not saved across internal routine calls.

REVERSE

```
►►──REVERSE(string)──────────────────────────────────────────────►◄
```

Returns *string* reversed.

Here are some examples:

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

RIGHT

```
►►──RIGHT(string,length──┬──────┬──)─────────────────────────────►◄
                         └─,pad─┘
```

Returns a string of length *length* containing the rightmost *length* characters of *string*. The string returned is padded with *pad* character, 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, as though the operation `number+0` had been carried out. It returns `-1` if *number* is less than `0`, `0` if it is `0`, and `1` if it is greater than `0`.

Here are some examples:

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

SOURCELINE

```
>>──SOURCELINE(──┬───┬──)─────────────────────────────────────────><
                 └─n─┘
```

Returns the line number of the final line in the program if you omit *n*. If you specify *n*, returns the *n*th line in the program if available at the time of execution. Otherwise, it returns a null string. If specified, *n* must be a positive whole number and must not exceed the number that a call to SOURCELINE with no arguments returns.

Here are some examples:

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

SPACE

```
>>──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'
```

STREAM

```
>>──STREAM(name──┬─────────────────────────────────────────┬──)────><
                 │    ┌─State──────────────────────────┐   │
                 └─,──┼────────────────────────────────┼───┘
                      ├─Command──,──stream_command─────┤
                      └─Description────────────────────┘
```

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 16. Input and Output Streams” on page 463.) 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 second argument can be one of the following strings that describe the action to be carried out. (Only the capitalized 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 string that is returned depends on the command performed and can be a null string. The possible input strings for the *stream_command* argument are described later.

Description
: returns any descriptive string associated with the current state of the specified stream. It is identical to the State operation, except that the returned string is followed by a colon and, if available, additional information about the ERROR or NOTREADY states.

State
: returns a string that indicates the current state of the specified stream. This is the default operation.

  The returned strings are as described in “STATE” on page 205.

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

Stream Commands: The following stream commands are used to:

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

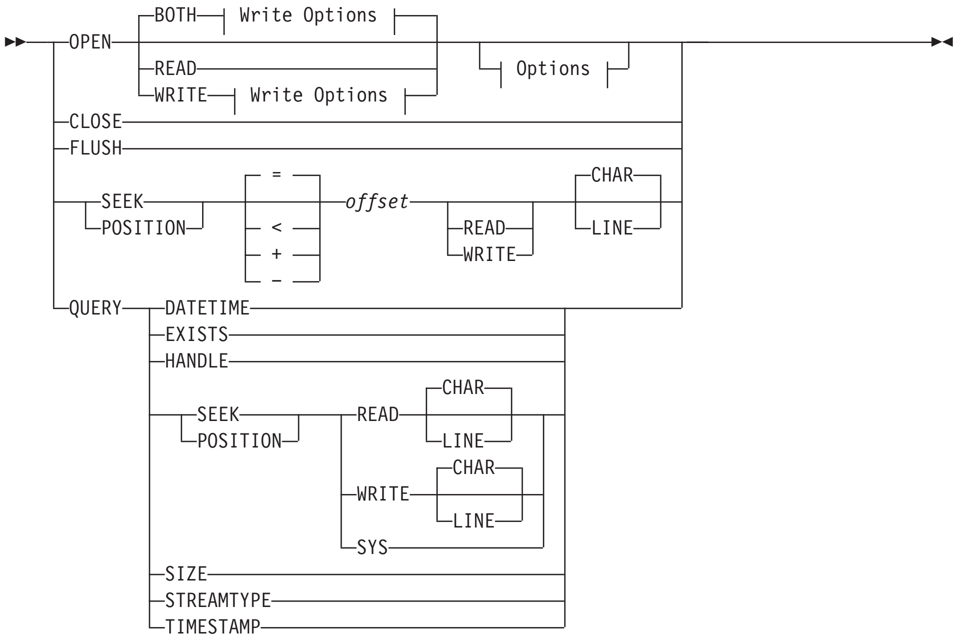
The *streamcommand* argument must be used when—and only when—you select the operation C (command). The syntax is:

```
>>──STREAM(name,'C',streamcommand)──────────────────────────────────><
```

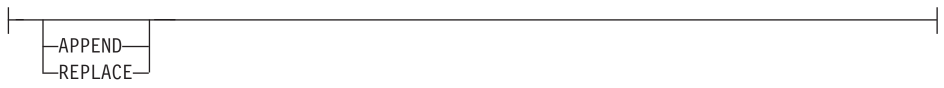
In this form, the STREAM function itself returns a string corresponding to the given *streamcommand* if the command is successful. If the command is unsuccessful, STREAM returns an error message string in the same form as the D (Description) operation supplies.

For most error conditions, the additional information is in the form of a numeric return code. This return code is the value of *ERRNO* that is set whenever one of the file system primitives returns with a -1.

Command Strings: The argument *streamcommand* can be any expression that the language processor evaluates to a command string that corresponds to the following diagram:



Write Options:



Options:

```
                      ┌──────────────────────────────┐
                      ▼                              │
├──┬────────────┬──────┬─NOBUFFER──────────────────┬─┴──────────────────────┤
   ├─SHARED─────┤      └─BINARY──┬────────────────┬┘
   ├─SHAREREAD──┤                └─RECLENGTH─length┘
   └─SHAREWRITE─┘
```

OPEN opens the named stream. The default for OPEN is to open the stream for both reading and writing data, for example, `'OPEN BOTH'`.

The STREAM function itself returns a description string similar to the one that the `D` option provides, for example, 'READY:' if the named stream is successfully opened, or 'ERROR:2' if the named stream is not found.

The following is a description of the options for OPEN:

READY opens the stream for reading only.

WRITE opens the stream for writing only.

BOTH opens the stream for both reading and writing. (This is the default.) Separate read and write pointers are maintained.

APPEND positions the write pointer at the end of the stream. The write pointer cannot be moved anywhere within the extent of the file as it existed when the file was opened.

REPLACE sets the write pointer to the beginning of the stream and truncates the file. In other words, this option deletes all data that was in the stream when opened.

SHARED Enables another process to work with the stream in a shared mode. This mode must be compatible with the shared mode (SHARED, SHAREREAD, or SHAREWRITE) used by the process that opened the stream.

SHAREREAD Enables another process to read the stream in a shared mode.

SHAREWRITE
Enables another process to write the stream in a shared mode.

NOBUFFER turns off buffering of the stream. Thus, all data written to the stream is flushed immediately to the operating system for writing. This option can severely affect output performance. Therefore, use it only when data integrity is a concern, or to force interleaved output to a stream to appear in the exact order in which it was written.

BINARY
causes the stream to be opened in binary mode. This means that line end characters are ignored and treated as another byte of data. This is intended to force file operations that are compatible with other REXX language processors that run on record-based systems, or to process binary data using the line operations.

Note: Specifying the BINARY option for a stream that does not exist but is opened for writing also requires the RECLENGTH option to be specified. Omitting the RECLENGTH option in this case raises an error condition.

RECLENGTH *length*
allows the specification of an exact length for each line in a stream. This allows line operations on binary-mode streams to operate on individual fixed-length records. Without this option, line operations on binary-mode files operate on the entire file (for example, as if the `RECLENGTH` option were specified with a length equal to that of the file). *length* must be 1 or greater.

Examples:

```
stream(strout,'c','open')
stream(strout,'c','open write')
stream(strinp,'c','open read')
stream(strinp,'c','open read shared')
```

CLOSE
closes the named stream. The STREAM function itself returns `READY:` if the named stream is successfully closed, or an appropriate error message. If an attempt is made to close an unopened file, STREAM returns a null string ("").

Example:

```
stream('STRM.TXT','c','close')
```

FLUSH
forces any data currently buffered for writing to be written to this stream.

SEEK *offset* sets the read or write position within a persistent stream. If the stream is opened for both reading and writing and no SEEK option is specified, both the read and write positions are set.

Note: See "Chapter 16. Input and Output Streams" on page 463 for a discussion of read and write positions in a persistent stream.

To use this command, the named stream must first be opened with the OPEN stream command or implicitly with an input or output operation. One of the following characters can precede the *offset* number:

= explicitly specifies the *offset* from the beginning of the stream. This is the default if no prefix is supplied. `Line Offset=1` means the beginning of stream.

< specifies *offset* from the end of the stream.

+ specifies *offset* forward from the current read or write position.

- specifies *offset* backward from the current read or write position.

The STREAM function itself returns the new position in the stream if the read or write position is successfully located or an appropriate error message otherwise.

The following is a description of the options for SEEK:

READ specifies that the read position is to be set by this command.

WRITE specifies that the write position is to be set by this command.

CHAR specifies that the positioning is to be done in terms of characters. This is the default.

LINE specifies that the positioning is to be done in terms of lines. For non-binary streams, this is an operation that can take a long time to complete, because, in most cases, the file must be scanned from the top to count line-end characters. However, for binary streams with a specified record length, this results in a simple multiplication of the new resulting line number by the record length, and then a simple character positioning. See “Line versus Character Positioning” on page 469 for a detailed discussion of this issue.

Note: If you do line positioning in a file open only for writing, you receive an error message.

Examples:

```
stream(name,'c','seek =2 read')
stream(name,'c','seek +15 read')
stream(name,'c','seek -7 write line')
fromend  = 125
stream(name,'c','seek <'fromend read)
```

POSITION is a synonym for SEEK.

QUERY Stream Commands: Used with these stream commands, the STREAM function returns specific information about a stream. Except for QUERY HANDLE and QUERY POSITION, the language processor returns the query information even if the stream is not open. The language processor returns the null string for nonexistent streams.

QUERY DATETIME
returns the date and time stamps of a stream in US format. This is included for compatibility with OS/2®.

```
stream('..\file.txt','c','query datetime')
```

A sample output might be:

```
11-12-98 03:29:12
```

QUERY EXISTS
returns the full path specification of the named stream, if it exists, or a null string.

```
stream('..\file.txt','c','query exists')
```

A sample output might be:

```
c:\data\file.txt
```

QUERY HANDLE
returns the handle associated with the open stream.

```
stream('..\file.txt','c','query handle')
```

A sample output might be:

```
3
```

QUERY POSITION
returns the current read or write position for the stream, as qualified by the following options:

READ
returns the current read position.

WRITE
returns the current write position.

Note: If the stream is open for both reading and writing, the default is to return the read position. Otherwise, it returns the appropriate position by default.

CHAR
returns the position in terms of characters. This is the default.

LINE
returns the position in terms of lines. For non-binary streams, this operation can take a long time to complete, because the language processor starts tracking the current line number if not already doing so. Thus, it might require a scan of the stream from the top to count line-end characters. See "Line versus Character Positioning" on page 469 for a detailed discussion of this issue.

```
stream('myfile','c','query position write')
```

A sample output might be:

```
247
```

SYS
returns the operating-system stream position in terms of characters.

QUERY SIZE
returns the size, in bytes, of a persistent stream.

```
stream('..\file.txt','c','query size')
```

A sample output might be:

```
1305
```

QUERY STREAMTYPE
returns a string indicating whether the stream is `PERSISTENT`, `TRANSIENT`, or `UNKNOWN`.

QUERY TIMESTAMP
returns the date and time stamps of a stream in an international format. This is the preferred method of getting the date and time because it provides the full 4-digit year.

```
stream('..\file.txt','c','query timestamp')
```

A sample output might be:

```
1998-11-12 03:29:12
```

STRIP

```
>>--STRIP(string--+----------------------------+--)------------------><
                  '-,--+--------+--+-------+--'
                       '-option-'  '-,char-'
```

Returns *string* with leading characters, trailing characters, or both, removed, based on the *option* you specify. The following are valid *option*s. (Only the capitalized 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)

```
>>--SUBSTR(string,n--+----------------------------+--)---------------><
                     '-,--+--------+--+------+---'
                          '-length-'  '-,pad-'
```

Returns the substring of *string* that begins at the *n*th character and is of length *length*, padded with *pad* if necessary. *n* must be a positive whole number. If *n* is greater than `LENGTH(string)`, 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 "LEFT" on page 328 and "RIGHT" on page 335.

SUBWORD

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

Returns the substring of *string* that starts at the *n*th word, and is up to *length* blank-delimited words. *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

```
►►──SYMBOL(name)──────────────────────────────────────────────────────►◄
```

Returns the state of the symbol named by *name*. It returns `BAD` if *name* is not a valid REXX symbol. It returns `VAR` if it is the name of a variable, that is, a symbol that has been assigned a value. Otherwise, it 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──┬───────────────────┬─┘
                     └─,string──┬──────────┬┘
                                └─,option2─┘
```

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 letter is needed; all characters following it are ignored.)

Civil returns the time in Civil format hh:mmxx. The hours can 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 and microseconds since the elapsed-time clock (described later) was started or reset. The returned number has no leading zeros or blanks, and the setting of NUMERIC DIGITS does not affect it. The number has always four trailing zeros in the decimal portion.

The language processor calculates elapsed time by subtracting the time at which the elapsed-time clock was started or reset from the current time. It is possible to change the system time clock while the system is running. This means that the calculated elapsed time value might not be a true elapsed time. If the time is changed so that the system time is earlier than when the REXX elapsed-time clock was started (so that the elapsed time would appear negative), the language processor raises an error and disables the elapsed-time clock. To restart the elapsed-time clock, trap the error through SIGNAL ON SYNTAX.

The clock can also be changed by programs on the system. Many LAN-attached programs synchronize the system time clock with the system time clock of the server during startup. This causes the REXX elapsed time function to be unreliable during LAN initialization.

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 (where uuuuuu are microseconds).

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. The hours can have the values `00` through `23`, and minutes and seconds, `00` through `59`. There are always two digits. Any fractions of seconds are ignored (times are never rounded). This is the default.

Reset
returns sssssssss.uuuuuu, the number of seconds and microseconds since the elapsed-time clock (described later) was started or reset and also resets the elapsed-time clock to zero. The returned number has no leading zeros or blanks, and the setting of NUMERIC DIGITS does not affect it. The number always has four trailing zeros in the decimal portion.

See the **Elapsed** option for more information on resetting the system time clock.

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')    ->   '16:54:22.120000'   /* 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 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 means that an internal routine inherits the time clock that 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')    ->    1.020000   /* or thereabouts */
                 /* pause of one second here */
time('R')    ->    2.030000   /* or thereabouts */
                 /* pause of one second here */
time('R')    ->    1.050000   /* or thereabouts */
```

Note: The elapsed-time clock is synchronized with the other calls to TIME and DATE, so several calls to the elapsed-time clock in a single clause always return the same result. For this reason, the interval between two usual TIME/DATE results can be calculated exactly using the elapsed-time clock.

If you specify *string*, TIME returns the time corresponding to *string* in the format *option*. The *string* must be supplied in the format *option2*. The default for *option2* is 'N'. So you need to specify *option2* only if *string* is not in the **N**ormal format. *option2* must specify the current time, for example, not 'E' or 'R'. Here are some examples:

```
time('C','11:27:21')    ->    11:27am
time('N','11:27am','C') ->    11:27:00
```

You can determine the difference between two times; for example:

```
If TIME('M','5:00pm','C')-TIME('M')<=0
then say "Time to go home"
else say "Keep working"
```

The TIME returned is the earliest time consistent with *string*. For example, if the result requires components that are not specified in the source format, then those components of the result are zero. If the source has components that the result does not need, then those components of the source are ignored.

Implementation maximum: If the number of seconds in the elapsed time exceeds nine digits (equivalent to over 31.6 years), an error results.

TRACE

```
►►──TRACE(──┬────────┬──)─────────────────────────────────────────►◄
            └─option─┘
```

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

If you specify *option*, it selects the trace setting. It must be the valid prefix `?`, one of the alphabetic character options associated with the TRACE instruction (that is, starting with `A`, `C`, `E`, `F`, `I`, `L`, `N`, `O`, or `R`), or both. (See the TRACE instruction in “Alphabetic Character (Word) Options” on page 80 for full details.)

Unlike the TRACE instruction, the TRACE function alters the trace action even if interactive debugging 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 debugging */
```

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, 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 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('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. The last character of any four-character string specified as the second argument is 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 rounded according to standard REXX rules, as though the operation `number+0` had been carried out. Then it is truncated to *n* decimal places or trailing zeros are added to reach the specified length. The result is never in exponential form. If there are no nonzero digits in the result, any minus sign is removed.

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.

VALUE

```
►►──VALUE(name─┬─────────────────────────────────┬─)───────────────────►◄
               └─,─┬──────────┬──┬───────────┬───┘
                   └─newvalue─┘  └─,selector─┘
```

Returns the value of the symbol that *name* (often constructed dynamically) represents and optionally assigns a new value to it. By default, VALUE refers to the current REXX-variables environment, but other, external collections of variables can be selected. If you use the function to refer to REXX variables, *name* must be a valid REXX symbol. (You can confirm this by using the

SYMBOL function.) Lowercase characters in *name* are translated to uppercase for the local environment. For the global environment lowercase characters are not translated because the global environment supports mixed-case identifiers. Substitution in a compound name (see “Compound Symbols” on page 33) occurs if possible.

If you specify *newvalue*, 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'
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          */
```

To use VALUE to manipulate environment variables, *selector* must be the string “ENVIRONMENT” or an expression that evaluates to “ENVIRONMENT”. In this case, the variable *name* need not be a valid REXX symbol. Environment variables set by VALUE are not kept after program termination.

Restriction: The values assigned to the variables must not contain any character that is a hexadecimal zero ('00'X). For example:

```
Call VALUE 'MYVAR', 'FIRST' || '00'X || 'SECOND',
'ENVIRONMENT'
```

sets MYVAR to "FIRST", truncating '00'x and 'SECOND'.

Here are some more examples:

```
/* Given that an external variable FRED has a value of 4        */
share = 'ENVIRONMENT'
say VALUE('fred',7,share)      /* says '4' and assigns          */
                               /* FRED a new value of 7         */

say VALUE('fred',,share)       /* says '7'                      */

/* Accessing and changing Windows environment entries given that */
                               /* PATH=C:\EDIT\DOCS;            */
env = 'ENVIRONMENT'
new = 'C:\EDIT\DOCS;'
say value('PATH',new,env)   /* says 'C:\WINDOWS' (perhaps)      */
```

```
                            /* and sets PATH = 'C:\EDIT\DOCS;'   */

say value('PATH',,env)      /* says 'C:\EDIT\DOCS;'              */
```

To delete an environment variable use the .NIL object as the *newvalue*. To delete the environment variable 'MYVAR' specify: value('MYVAR', .NIL, 'ENVIRONMENT'). If you specify an empty string as the *newvalue* like in value('MYVAR', '', `'ENVIRONMENT'`) the value of the external environment variable is set to an empty string which on Windows is not the same as deleting the environment variable.

Note: In contrary to OS/2, the Windows environment is unchanged after program termination.

You can use the VALUE function to return a value to the global environment directory. To do so, omit *newvalue* and specify *selector* as the null string. The language processor sends the message *name* (without arguments) to the current environment object. The environment returns the object identified by *name*. If there is no such object, it returns, by default, the string *name* with an added initial period (an environment symbol—see “Environment Symbols” on page 35).

Here are some examples:

```
/* Assume the environment name MYNAME identifies the string "Simon"    */
name = value('MYNAME',,'')  /* Sends MYNAME message to the environment */
name = .myname              /* Same as previous instruction            */
say 'Hello,' name           /* Produces: "Hello, Simon"                */
/* Assume the environment name NONAME does not exist.                  */
name = value('NONAME',,'')  /* Sends NONAME message to the environment */
say 'Hello,' name           /* Produces:  "Hello, .NONAME"             */
```

You can use the VALUE function to change a value in the REXX environment directory. Include a *newvalue* and specify *selector* as the null string. The language processor sends the message *name* (with = appended) and the single argument *newvalue* to the current environment object. After receiving this message, the environment identifies the object *newvalue* by the name *name*.

Here is an example:

```
name = value('MYNAME','David','') /* Sends "MYNAME=("David") message  */
/* to the environment.                                                */
/* You could also use:                                                */
/*   call value 'MYNAME','David',''                                   */
say 'Hello,' .myname              /* Produces:  "Hello, David"        */
```

Notes:

1. If the VALUE function refers to an uninitialized REXX variable, the default value of the variable is always returned. The NOVALUE condition is not raised because a reference to an external collection of variables never raises NOVALUE.
2. The VALUE function is used when a variable contains the name of another variable, or when a name is constructed dynamically. If you specify *name* as a single literal string and omit *newvalue* and *selector*, the symbol is a constant and 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 trapped. See "Chapter 12. Conditions and Condition Traps" on page 429.

VAR

```
►►──VAR(name)──────────────────────────────────────────────────────►◄
```

Returns 1 if *name* is the name of a variable, that is, a symbol that has been assigned a value), or `0`.

Here are some examples:

```
/* Following: DROP A.3; J=3  */
VAR('J')       -> 1
VAR(J)         -> 0  /* has tested "3"    */
VAR('a.j')     -> 0  /* has tested "A.3"  */
VAR(2)         -> 0  /* a constant symbol */
VAR('*')       -> 0  /* an invalid symbol */
```

VERIFY

```
►►──VERIFY(string,reference──┬──────────────────────┬──)──────────►◄
                             └─,─┬────────┬─┬───────┬┘
                                 └─option─┘ └─,start┘
```

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

The *option* can be either `Nomatch` (the default) or `Match`. (Only the capitalized and highlighted letter is needed. All characters following it are ignored, and it can be in uppercase or lowercase characters.) If you specify `Match`, the function returns the position of the first character in the *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 Match; 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

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

Returns the *n*th blank-delimited word in *string* or returns the null string if less than *n* words are in *string*. *n* must be a positive whole number. This function is equal to SUBWORD(*string*,*n*,1).

Here are some examples:

```
WORD('Now is the time',3)    ->    'the'
WORD('Now is the time',5)    ->    ''
```

WORDINDEX

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

Returns the position of the first character in the *n*th blank-delimited word in *string* or returns 0 if less than *n* words are in *string*. *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

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

Returns the length of the *n*th blank-delimited word in the *string* or returns `0` if less than *n* words are in the *string*. *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)

```
►►──WORDPOS(phrase,string─┬────────┬─)────────────────────────────►◄
                          └─,start─┘
```

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. Several 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

```
►►──WORDS(string)─────────────────────────────────────────────────►◄
```

Returns the number of blank-delimited words in *string*.

Here are some examples:

```
WORDS('Now is the time')    ->    4
WORDS(' ')                  ->    0
```

XRANGE (Hexadecimal Range)

```
►►──XRANGE(──┬───────┬──┬──────┬──)──────────────────────────────────►◄
             └─start─┘  └─,end─┘
```

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('FE'x,'02'x)  ->   'FEFF000102'x
XRANGE('i','j')      ->   'ij'                  /* ASCII  */
```

X2B (Hexadecimal to Binary)

```
►►──X2B(hexstring)───────────────────────────────────────────────────►◄
```

Returns a string, in character format, that represents *hexstring* converted to binary. The *hexstring* is a string of hexadecimal characters. It can be of any length. Each hexadecimal character is converted to a string of 4 binary digits. You can optionally include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to improve readability; they are ignored.

The returned string has a length that is a multiple of 4, and does not include any blanks.

If *hexstring* is null, the function returns a null string.

Here are some examples:

```
X2B('C3')        ->  '11000011'
X2B('7')         ->  '0111'
X2B('1 C1')      ->  '000111000001'
```

You can combine X2B with the functions D2X and C2X to convert numbers or character strings into binary form.

Here are some examples:

```
X2B(C2X('C3'x))  ->  '11000011'
X2B(D2X('129'))  ->  '10000001'
X2B(D2X('12'))   ->  '1100'
```

X2C (Hexadecimal to Character)

```
►►──X2C(hexstring)──────────────────────────────────────────────────►◄
```

Returns a string, in character format, that represents *hexstring* converted to character. The returned string has half as many bytes as the original *hexstring*. *hexstring* can be of any length. If necessary, it is padded with a leading zero to make an even number of hexadecimal digits.

You can optionally include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to improve readability; they are ignored.

If *hexstring* is null, the function returns a null string.

Here are some examples:

```
X2C('4865 6c6c 6f') ->  'Hello'     /*  ASCII             */
X2C('3732 73')      ->  '72s'       /*  ASCII             */
```

X2D (Hexadecimal to Decimal)

```
►►──X2D(hexstring──┬──────┬──)──────────────────────────────────────►◄
                   └─,n───┘
```

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 occurs. 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*, the *hexstring* is processed as an unsigned binary number.

Here are some examples:

```
X2D('0E')       ->    14
X2D('81')       ->    129
X2D('F81')      ->    3969
X2D('FF81')     ->    65409
X2D('46 30'X)   ->    240         /*  ASCII   */
X2D('66 30'X)   ->    240         /*  ASCII   */
```

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 both cases it is converted to a whole number, which can be negative. If *n* is `0`, the function returns `0`.

If necessary, *hexstring* is padded on the left with `0` characters (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
```

Windows Application Programming Interface Functions

You can use the following built-in REXX functions in a REXX program to register, drop, or query external function packages and to create and manipulate external data queues. See the *Object REXX for Windows: Programming Guide* for a full discussion of the external-function interfaces.

RXFUNCADD

```
►►──RXFUNCADD(name,module──┬────────────┬──)────────────────────────►◄
                           └─,procedure─┘
```

Registers the function *name*, making it available to REXX procedures. A return value 0 signifies successful registration. A return value 1 signifies that the function is already registered.

```
rxfuncadd('SysCls','rexxutil', 'SysCls') -> 0 /* if not already registered */
                                         -> 1 /* if already registered */
```

Note: A return code of 0 signifies only that the function is successfully registered, but not that the function or module is actually available.

RXFUNCDROP

```
►►──RXFUNCDROP(name)──────────────────────────────────────────────────►◄
```

Removes (deregisters) the function *name* from the list of available functions. A zero return value signifies successful removal.

```
rxfuncdrop('SysLoadFuncs')         -> 0 /* if successfully removed   */
```

RXFUNCQUERY

```
►►──RXFUNCQUERY(name)──────────────────────────────────────────────────────►◄
```

Queries the list of available functions for the function *name*. It returns a value of `0` if the function is registered, and a value of `1` if it is not.

```
rxfuncquery('SysLoadFuncs')       -> 0 /* if registered            */
```

RXQUEUE

```
►►──RXQUEUE(──┬─"Create"──┬──────────────┬──┬─)────────────────────────────►◄
              │           └─,queuename───┘  │
              ├─"Delete",queuename──────────┤
              ├─"Get"───────────────────────┤
              └─"Set",newqueuename──────────┘
```

Creates and deletes external data queues. It also sets and queries their names.

"Create" creates a queue with the name *queuename* if you specify *queuename* and if no queue of that name exists already. You must not use SESSION as a *queuename*. If you specify no *queuename*, then the language processor provides a name. The name of the queue is returned in either case.

The maximum length of *queuename* can be 1024 characters.

Many queues can exist at the same time, and most systems have sufficient resources available to support several hundred queues at a time. If a queue with the specified name exists already, a queue is still created with a name assigned by the language processor. The assigned name is then returned to you.

"Delete" deletes the named queue. It returns 0 if successful or a nonzero number if an error occurs. Possible return values are:

0	Queue has been deleted.
5	Not a valid queue name or tried to delete queue named 'SESSION'.
9	Specified queue does not exist.
10	Queue is busy; wait is active.
12	A memory failure has occurred.
1002	Failure in memory manager.

"Get" returns the name of the queue currently in use.

"Set"	sets the name of the current queue to *newqueuename* and returns the previously active queue name.

The first parameter determines the function. Only the first character of the first parameter is significant. The parameter can be entered in any case. The syntax for a valid queue name is the same as for a valid REXX symbol.

The second parameter specified for Create, Set, and Delete must follow the same syntax rules as the REXX variable names. There is no connection, however, between queue names and variable names. A program can have a variable and a queue with the same name. The actual name of the queue is the uppercase value of the name requested.

Named queues prevent different REXX programs that are running in a single session from interfering with each other. They allow REXX programs running in different sessions to synchronize execution and pass data. LINEIN('QUEUE:') is especially useful because the calling program stops running until another program places a line on the queue.

```
/* default queue                                   */
rxqueue('Get')            -> 'SESSION'
/* assuming FRED does not already exist            */
rxqueue('Create', 'Fred)'  -> 'FRED'
/* assuming SESSION had been active                */
rxqueue('Set', 'Fred')     -> 'SESSION'
/* assuming FRED did not exist                     */
rxqueue('delete', 'Fred')  -> '0'
```

Chapter 9. REXX Utilities (RexxUtil)

RexxUtil is a Dynamic Link Library (DLL) package for Windows; the package contains REXX functions. These functions:
- Manipulate Windows operating system files and directories
- Manipulate Windows classes and objects
- Perform text screen input and output

To use a RexxUtil function, you must first register the function with the REXX RxFuncAdd function:

```
call RxFuncAdd 'SysCls', 'rexxutil', 'SysCls'
```

This example registers the SysCls function, which can now be used in your REXX programs.

The SysLoadFuncs RexxUtil function automatically loads the other RexxUtil functions. The following instructions in a REXX program register all of the RexxUtil functions:

```
call RxFuncAdd 'SysLoadFuncs', 'rexxutil', 'SysLoadFuncs'
call SysLoadFuncs
```

Once registered, the RexxUtil functions are available from all Windows operating system sessions.

The SysDropFuncs RexxUtil function lets you drop all RexxUtil functions. The following instruction in a REXX program deregisters all of the RexxUtil functions:

```
call SysDropFuncs
```

RxMessageBox

```
►►──RxMessageBox(text──┬──────────────────────────────────────────┬──)──────────►◄
                       └─,──┬───────┬──┬────────────────────────┬─┘
                            └─title─┘  └─,──┬────────┬──┬──────┬─┘
                                            └─button─┘  └─,icon─┘
```

Displays a Windows message box.

RxMessageBox returns the selected message box push button. Possible values are:

1	The OK push button was pressed
2	The CANCEL push button was pressed
3	The ABORT push button was pressed
4	The RETRY push button was pressed
5	The IGNORE push button was pressed
6	The YES push button was pressed
7	The NO push button was pressed

If a message box has a 'CANCEL' button, the function returns the 2 value if either the ESC key is pressed or the 'CANCEL' button is selected. If the message box has no 'CANCEL' button, pressing ESC has no effect.

Parameters:

text The message box text.

title The message box title. The default title is "Error!".

button The message box push button style. The allowed styles are:

'NONE'
No icon is displayed.

'OK' A single OK push button.

'OKCANCEL'
An OK push button and a CANCEL push button.

'RETRYCANCEL'
A RETRY push button and a CANCEL push button.

'ABORTRETRYIGNORE'
An ABORT push button, a RETRY push button and an IGNORE push button.

'YESNO'
A YES push button and a NO push button.

'YESNOCANCEL'
A YES push button, a NO push button and a CANCEL push button.

'QUERY'
A query icon is displayed.

'WARNING'
A warning icon is displayed.

'ERROR'
An error icon is displayed.

The default push button style is OK.

icon The message box icon style. The allowed styles are:

'HAND'
A hand icon is displayed.

'QUESTION'
A question mark icon is displayed.

'EXCLAMATION'
An exclamation point icon is displayed.

'ASTERISK'
An asterisk icon is displayed.

'INFORMATION'
An information icon is displayed.

'STOP'
A stop icon is displayed.

Example:

```
/* Give option to quit       */
if RxMessageBox("Shall we continue",, "YesNo", "Question") = 7
Then Exit                            /* quit option given, exit    */
```

RxWinExec

```
►►──RxWinExec(─cmdline──┬──────────┬──)────────────────────────────────────►◄
                        └─,cmdshow─┘
```

Runs the application as specified in *cmdline*.

Parameters:

cmdline
A string containing a file name and optional parameters for the application to be executed. If the name of the executable file in *cmdline* does not contain a directory path, RxWinExec searches for the executable file in this sequence:

1 The directory from which Object REXX was loaded.

2 The current directory.

3 The Windows system directory.

4 The Windows directory.

5 The directories listed in the PATH environment variable.

cmdshow
Specifies how a Windows-based application window is to be shown. For a non-Windows-based application, the PIF file, if any, for the application determines the window state.

SHOWNORMAL
Activates and displays a window.

SHOWNOACTIVATE
Displays the window while the current active window remains active.

SHOWMINNOACTIVE
Displays the window as a minimized window, the current active window remains active.

SHOWMINIMIZED
Activates the window and displays it as a minimized window.

SHOWMAXIMIZED
Activates the window and displays it as a maximized window.

HIDE Hides the window and activates another window.

MINIMIZE
Minimizes the specified window and activates the next top-level window in the Z order.

Return codes:

If the function succeeds, the return value is greater than 31; otherwise, it is one of the following:

0 The system is out of memory or resources.

2 The specified file was not found.

3 The specified path was not found.

11 The EXE file is invalid.

SysAddRexxMacro

```
►►──SysAddRexxMacro(name,file──┬────────┬──)──────────────────────────►◄
                               └─,order─┘
```

Adds a routine to the REXX macrospace. SysAddRexxMacro returns the RexxAddMacro return code.

Parameters:

name	The name of the function added to the macrospace.
file	The file containing the REXX program.
order	The macrospace search order. The order can be 'B' (Before) or 'A' (After).

SysBootDrive

```
►►──SysBootDrive()─────────────────────────────────────────────────►◄
```

Returns the drive used to boot Windows, for example, 'C:'.

SysClearRexxMacroSpace

```
►►──SysClearRexxMacroSpace()───────────────────────────────────────►◄
```

Clears the REXX macrospace. SysClearRexxMacroSpace returns the RexxClearMacroSpace return code.

SysCloseEventSem

```
►►──SysCloseEventSem(handle)───────────────────────────────────────►◄
```

Closes an event semaphore.

Parameter:

handle	A handle returned from a previous SysCreateEventSem or SysOpenEventSem call.

Return codes:

0	No errors.
6	Invalid handle.
102	Error semaphore busy.

SysCloseMutexSem

```
►►──SysCloseMutexSem(handle)──────────────────────────────────────────────►◄
```

Closes a mutex semaphore.

Parameter:

handle A handle returned from a previous SysCreateMutexSem call.

Return codes:

0 No errors.

6 Invalid handle.

102 Error semaphore busy.

SysCls

```
►►──SysCls()──────────────────────────────────────────────────────────────►◄
```

Clears the screen.

Example:

```
/* Code */
call SysCls
```

SysCreateEventSem

```
►►──SysCreateEventSem(──┬──────┬──┬──────────────┬──)─────────────────────►◄
                        └─name─┘  └─manual_reset─┘
```

Creates or opens an event semaphore. It returns an event semaphore handle that can be used with SysCloseEventSem, SysOpenEventSem, SysResetEventSem, SysPostEventSem, and SysWaitEventSem. SysCreateEventSem returns a null string ("") if the semaphore cannot be created or opened.

Parameters:

name The optional event semaphore name. If you omit *name*, SysCreateEventSem creates an unnamed, shared event semaphore. If you specify *name*, SysCreateEventSem opens the semaphore if the semaphore has already been created. A semaphore name can be MAX_PATH long, and can contain any character except the backslash (\) path-separator character. Semaphore names are case-sensitive.

manual_reset
: A flag to indicate that the event semaphore must be reset manually by SysResetEventSem. If this parameter is omitted, the event semaphore is reset automatically by SysWaitEventSem.

SysCreateMutexSem

```
►►──SysCreateMutexSem(──┬──────┬──)──────────────────────────────────►◄
                        └─name─┘
```

Creates or opens a mutex semaphore. Returns a mutex semaphore handle that can be used with SysCloseMutexSem, SysRequestMutexSem, and SysReleaseMutexSem. SysCreateMutexSem returns a null string ("") if the semaphore cannot be created or opened.

Parameter:

name The optional mutex semaphore name. If you omit *name*, SysCreateMutexSem creates an unnamed, shared mutex semaphore. If you specify *name*, SysCreateMutexSem opens the semaphore if the mutex has already been created. The semaphore names cannot be longer than 63 characters. Semaphore names are case-sensitive.

SysCurPos

```
►►──SysCurPos(──┬────────────┬──)────────────────────────────────────►◄
                └─row,column─┘
```

Returns the cursor position in the form `row col` and optionally moves the cursor to a new location.

Parameters:

row The row to move to.

col The column to move to.

Note: Position (0,0) is the upper left corner.

You can call SysCurPos without a column and row position to obtain the cursor position without moving the cursor.

Example:

```
/* Code */
call SysCls
parse value SysCurPos() with row col
say 'Cursor position is 'row', 'col


/* Output */
Cursor position is 0, 0
```

SysCurState

```
►►—SysCurState(state)——————————————————————————————————————————►◄
```

Hides or displays the cursor.

Parameter:

state The new cursor state. Allowed states are:

'ON'	Display the cursor
'OFF'	Hide the cursor

SysDriveInfo

```
►►—SysDriveInfo(drive)—————————————————————————————————————————►◄
```

Returns drive information in the form: `drive:  free total label`.

drive:	is the drive letter identifier.
free	is the drive unused space.
total	is the total size of the drive.
label	is the drive label.

If the drive is not accessible, then SysDriveInfo returns ''.

Parameter:

drive The drive of interest, 'C:'.

Example:

```
/* Code */
say 'Disk='SysDriveInfo('C:')
/* Output */
Disk=C: 33392640 83687424 TRIGGER_C
```

SysDriveMap

```
►►──SysDriveMap(──┬───────┬───┬──────┬──)──────────────────────────────────►◄
                  └─drive─┘   └─,opt─┘
```

Returns a string listing accessible drives (separated by blanks) in the form: `C: D: ....`

Parameters:

drive The first drive letter of the drive map. The default is 'C:'.

opt The drivemap option. This can be:

'USED'
: returns the drives that are accessible or in use, including all local and remote drives. This is the default.

'FREE'
: returns drives that are free or not in use.

'LOCAL'
: returns only local drives.

'REMOTE'
: returns only remote drives, such as redirected LAN resources or installable file system (IFS) attached drives.

'REMOVABLE'
: returns removable drives.

'CDROM'
: returns CD-ROM drives.

'RAMDISK'
: returns drives assigned from RAM.

Example:

```
/* Code */
say 'Used drives include:'
say SysDriveMap('C:', 'USED')
/* Output */
Used drives include:
C: D: E: F: W:
```

SysDropFuncs

```
►►──SysDropFuncs──────────────────────────────────────────────────────►◄
```

Drops all RexxUtil functions. After a REXX program calls SysDropFuncs, the RexxUtil functions are not available in any Windows operating system sessions.

SysDropRexxMacro

```
►►──SysDropRexxMacro(name)────────────────────────────────────────────►◄
```

Removes a routine from the REXX macrospace. SysDropRexxMacro returns the RexxDropMacro return code.

Parameter:

name The name of the function removed from the macrospace.

SysDumpVariables

```
►►──SysDumpVariables──┬──────────────┬────────────────────────────────►◄
                      └─(──name──)───┘
```

Dumps all variables in the current scope either to the specified file *filename* (new data is appended) or to STDOUT if you omit *filename*. The format of the data is, with one variable per line:

```
Name=MYVAR, Value="This is the content of MYVAR"
```

Parameter:

filename
The name of the file to which variables are appended. The dump is written to STDOUT if you omit this parameter.

Return codes:

0 Dump completed successfully.

–1 Dump failed.

Example:

```
Call SysDumpVariables "MyVars.Lst" /* append vars to file */
Call SysDumpVariables              /* list vars on STDOUT */
```

SysFileDelete

```
►►──SysFileDelete(file)──────────────────────────────────────────►◄
```

Deletes a file. SysFileDelete does not support wildcard file specifications.

Parameter:

file	The name of the file to be deleted.

Return codes:

0	File deleted successfully.
2	File not found.
3	Path not found.
5	Access denied or busy.
26	Not DOS disk.
32	Sharing violation.
36	Sharing buffer exceeded.
87	Does not exist.
206	File name exceeds range error.

Example:

```
/* Code */
parse arg InputFile OutputFile
call SysFileDelete OutputFile   /* unconditionally erase output file */
```

SysFileSearch

```
►►──SysFileSearch(target,file,stem──┬──────────┬──)──────────────►◄
                                    └─,options─┘
```

Finds all file lines containing the target string and returns the file lines in a REXX stem variable collection.

Parameters:

target	The target search string.
file	The searched file.
stem	The result stem variable name. SysFileSearch sets REXX variable *stem.0* to the number of lines returned and stores the individual lines in variables *stem.1* to *stem.n*.
options	Any combination of the following one-character options:

'C'	Conducts a case-sensitive search.
'N'	Returns the file line numbers.

The default is a case-insensitive search without line numbers.

Return codes:

0	Successful.
2	Not enough memory.
3	Error opening file.

Example:

```
/* Find DEVICE statements in CONFIG.SYS */
call SysFileSearch 'DEVICE', 'C:\CONFIG.SYS', 'file.'
do i=1 to file.0
say file.i
end

/* Output */
DEVICE=C:\SB16\DRV\CTSB16.SYS /UNIT=0 /BLASTER=A:240 I:5 D:1 H:5
DEVICE=C:\SB16\DRV\CTMMSYS.SYS
rem ****    DOS SCSI CDROM device drivers ***
DEVICE=C:\SCSI\ASPI8DOS.SYS /D
DEVICE=C:\SCSI\ASPICD.SYS /D:ASPICD0
rem **** IDE CDROM device drivers
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\SBCD\DRV\SBIDE.SYS /V /D:MSCD001  /P:1f0,14
DEVICE=C:\DOS\SETVER.EXE
DEVICE=C:\WINDOWS\SMARTDRV.EXE /DOUBLE_BUFFER
DEVICE=C:\WINDOWS\IFSHLP.SYS

/* Find DEVICE statements in CONFIG.SYS (along with */
/* line numbers)                                    */
call SysFileSearch 'DEVICE', 'C:\CONFIG.SYS', 'file.', 'N'
do i=1 to file.0
say file.i
end

/* Output */
1 DEVICE=C:\SB16\DRV\CTSB16.SYS /UNIT=0 /BLASTER=A:240 I:5 D:1
```

```
H:5
2 DEVICE=C:\SB16\DRV\CTMMSYS.SYS
4 rem ****    DOS SCSI CDROM device drivers ***
5 DEVICE=C:\SCSI\ASPI8DOS.SYS /D
6 DEVICE=C:\SCSI\ASPICD.SYS /D:ASPICD0
8 rem **** IDE CDROM device drivers
9 DEVICE=C:\DOS\HIMEM.SYS
10 DEVICE=C:\SBCD\DRV\SBIDE.SYS /V /D:MSCD001  /P:1f0,14
13 DEVICE=C:\DOS\SETVER.EXE
16 DEVICE=C:\WINDOWS\SMARTDRV.EXE /DOUBLE_BUFFER
17 DEVICE=C:\WINDOWS\IFSHLP.SYS
```

SysFileSystemType

```
►►──SysFileSystemType(drive)──────────────────────────────────────►◄
```

Returns the name of the file system used for a drive. If the drive is not accessible, it returns a null string ("").

Parameter:

drive The drive of interest, for example 'C:'.

Example:

```
/* Code */
say 'File System='SysFileSystemType('C:')
/* Output */
File System=NTFS
```

SysFileTree

```
►►──SysFileTree(filespec,stem─────────────────────────────────────►

►──┬──────────────────────────────────────────────┬──)──────────►◄
   └─,──┬─────────┬──┬──────────────────────────┬─┘
        └─options─┘  └─,──┬─────────┬──┬──────────────┬─┘
                          └─tattrib─┘  └─,──┬─────────┬─┘
                                            └─nattrib─┘
```

Finds all files that match a file specification. SysFileTree returns the file descriptions (date, time, size, attributes, and file specification) in a REXX stem variable collection.

Parameters:

filespec The search file specification.

stem The name of a stem variable to be used for storing results. SysFileTree sets REXX variable *stem.0* to the number of files and directories found and stores individual file descriptions into variables *stem.1* to *stem.n*.

options Any combination of the following:

'F' Search only for files.

'D' Search only for directories.

'B' Search for both files and directories. This is the default.

'S' Search subdirectories recursively.

'T' Return the time and date in the form YY/MM/DD/HH/MM.

'L' Return the time and date in the form YYYY-MM-DD HH:MM:SS.

'O' Return only the fully-qualified file name. The default is to return the date, time, size, attributes, and fully-qualified name for each file found.

tattrib The target attribute mask for file specification matches. Only files that match the target mask are returned. The default mask is '*****'. This returns all files regardless of the settings (clear or set) of the Archive, Directory, Hidden, Read-Only, and System attributes. The target mask attributes must appear in the order 'ADHRS'.

Target Mask Options

***** The file attribute may be any state.

+ The file attribute must be set.

- The file attribute must be cleared.

Target Mask Examples

'*+*'** Find all files with the Read-Only attribute set.

'++*'**
Find all files with the Read-Only and Archive attributes set.

'*++'**
Find all hidden subdirectories.

'---+-' Find all files with only the Read-Only attribute set.

nattrib The new attribute mask for setting the attributes of each matching file. The default mask is '*****'. This means not to change the Archive, Directory, Hidden, Read-Only, and System attributes. The target mask attributes must appear in the order 'ADHRS'.

New Attribute Mask Options

*	Do not change the file attribute.
+	Set the file attribute.
-	Clear the file attribute.

New Attribute Mask Examples

'*+*'** Set the Read-Only attribute on all files.

'-+*'** Set the Read-Only attribute and clear the Archive attribute of each file.

'+*+++'
Set all file attributes, except the directory attribute.

'-----' Clear all attributes on all files.

Note: You cannot set the directory attribute on non-directory files.

SysFileTree returns the file attribute settings after the new attribute mask has been applied.

Return codes:

0 Successful.

2 Not enough memory.

Examples:

```
/* Find all subdirectories on C: */
call SysFileTree 'c:\*.*', 'file', 'SD'

/* Find all Read-Only files */
call SysFileTree 'c:\*.*', 'file', 'S', '***+*'

/* Clear Archive and Read-Only attributes of files that have them set */
call SysFileTree 'c:\*.*', 'file', 'S', '+**+*', '-**-*'

/****<< Sample Code and Output Example.>>********/

/* Code */
call SysFileTree 'c:\win*.', 'file', 'B'
do i=1 to file.0
say file.i
end

/* Actual Output */
5:24:95   4:59p          0  -D---  C:\WINDOWS
```

SysFromUnicode

```
>>──SysFromUnicode──(──string──┬───────────┬──┬───────────────┬──────────────►
                                └─,codepage─┘  └─,mappingflags─┘

►──┬──────────────┬─,outstem─)──────────────────────────────────────────────►◄
   └─,defaultchar─┘
```

Maps a UNICODE character string to a character string. The new character string and additional information is returned in the outstem.

Parameters:

string A string containing the UNICODE characters to be mapped.

codepage
Specifies the code page used to perform the conversion. This parameter can be the value of any code page that is installed or available in the system. The default is the current original equipment manufacturer (OEM) code-page identifier for the system.

You can also specify one of the following values:

ACP	ANSI code page.
OEMCP	OEM code page.
SYMBOL	Windows 2000: symbol code page.
THREAD_ACP	Windows 2000: current thread's ANSI code page.
UTF7	Windows NT 4.0 and Windows 2000: translate using UTF-7.
UTF8	Windows NT 4.0 and Windows 2000: translate using UTF-8. When this is set, `mappingflags` must be set.

mappingflags
Specifies the handling of unmapped characters. The function performs more quickly when none of these flags is set.

The following flags can be used:

COMPOSITECHECK	Converts composite characters to precomposed characters.
SEPCHARS	Discards nonspacing characters during conversion.

DISCARDNS Generates separate characters during conversion. This is the default conversion behavior.

DEFAULTCHAR
Replaces non-convertible characters with the default character during conversion.

When `compositecheck` is specified, the function converts composite characters to precomposed characters. A composite character consists of a base character and a nonspacing character, each having different character values. A precomposed character has a single character value for a combination of a base and a nonspacing character. In the character è, the 'e' is the base character, and the 'grave' accent mark is the nonspacing character.

When `compositecheck` is specified, it can use the last three flags in this list (`discardns`, `sepchars`, and `defaultchar`) to customize the conversion to precomposed characters. These flags determine the function's behavior when there is no precomposed mapping for a combination of a base and a nonspace character in a Unicode character string. These last three flags can be used only if the `compositecheck` flag is set. The function's default behavior is to generate separate characters (`sepchars`) for unmapped composite characters.

defaultchar
Character to be used if a Unicode character cannot be represented in the specified code page. If this parameter is NULL, a system default value is used. The function is faster when both `lpDefaultChar` and `defaultchar` are not used.

Return codes:

0 No errors.

87 Incorrect code page or `codepage` value.

1004 Invalid mapping flags.

SysToUnicode

```
►►──SysToUnicode(string──┬───────────┬──┬─────────────────┬─,outstem)──────────►◄
                         └─,codepage─┘  └─,translateflags─┘
```

Maps a character string to a UNICODE string.

Parameters:

string A string containing the UNICODE characters to be mapped.

codepage
Specifies the code page used to perform the conversion. This parameter can be the value of any code page that is installed or available in the system. The default is the current original equipment manufacturer (OEM) code-page identifier for the system.

You can also specify one of the following values:

ACP	ANSI code page.
OEMCP	OEM code page.
SYMBOL	Windows 2000: symbol code page.
THREAD_ACP	Windows 2000: current thread's ANSI code page.
UTF7	Windows NT 4.0 and Windows 2000: translate using UTF-7.
UTF8	Windows NT 4.0 and Windows 2000: translate using UTF-8. When this is set, `translateflags` must be set.

translateflags
Indicates whether to translate to precomposed or composite-wide characters (if a composite form exists), whether to use glyph characters in place of control characters, and how to deal with invalid characters.

You can specify a combination of the following flags:

PRECOMPOSED	Always use precomposed characters, that is, characters in which a base character and a nonspacing character have a single character value. This is the default translation option. Cannot be used with COMPOSITE.
COMPOSITE	Always use composite characters, that is, characters in which a base character and a nonspacing character have different character values. Cannot be used with PRECOMPOSED.
ERR_INVALID_CHARS	If the function encounters an invalid input character, it fails and returns "1113".
USEGLYPHCHARS	Use glyph characters instead of control characters.

A composite character consists of a base character and a nonspacing character, each having different character values. A precomposed character has a single character value for a base-nonspacing character combination. In the character è, the ′e′ is the base character and the ′grave′ accent mark is the nonspacing character. The function′s default behavior is to translate to the precomposed form. If a precomposed form does not exist, the function attempts to translate to a composite form.

The flags PRECOMPOSED and COMPOSITE are mutually exclusive. The USEGLYPHCHARS flag and the ERR_INVALID_CHARS can be set regardless of the state of the other flags.

Return codes:

0 No errors.

87 Incorrect code page or `codepage` value.

1004 Invalid translate flags.

1113 No mapping for the Unicode character exists in the target code page.

SysGetErrortext

```
►►──SysGetErrortext(errornumber)──────────────────────────────────►◄
```

Obtains a string describing the system error identified by the error number.

Returns a string with the description of the error, or an empty string if no description is available.

Example:

```
err=SysMkDir("c:\temp)
if err \= 0 then
say "Error" err":"SysGetErrortext(err)
```

SysGetFileDateTime

```
►►──SysGetFileDateTime(filename─┬────────────┬─)──────────────────►◄
                                └─,──timesel─┘
```

Returns the selected data and time attribute of the file *filename* provided that this is supported by the operating and file system. FAT, for example, does not support Create/Access. The selector for the time to be returned can be abbreviated to the first character.

In Windows NT, the *filename* can also be a directory name.

The file that you want to query must not be opened by another process or must at least allow shared writes to query the time stamp.

Parameters:

filename	The name of the file to be queried.
timesel	The file time to be queried, namely CREATE, ACCESS, WRITE.

Return codes:

The date and time in the format YYYY-MM-DD HH:MM:SS, or –1 to indicate that the file date and time query failed

Example:

```
Say "File creation time:" SysGetFileDateTime("MyFile.Log", "C")
Say "File last access time:" SysGetFileDateTime("MyFile.Log", "A")
Say "File last update time:" SysGetFileDateTime("MyFile.Log", "W")

Say "Directory creation time:" SysGetFileDateTime("C:\MyDir", "C")
                                          /* in Windows NT */
```

SysGetKey

```
►►──SysGetKey(──┬─────┬──)──────────────────────────────────────────►◄
                └─opt─┘
```

Reads and returns the next key from the keyboard buffer. If the keyboard buffer is empty, SysGetKey waits until a key is pressed. Unlike the CHARIN built-in function, SysGetKey does not wait until the Enter key is pressed.

Parameter:

opt An option controlling screen echoing. Allowed values are:

‘ECHO’	Echo the pressed key to the screen. This is the default.
‘NOECHO’	Do not echo the pressed key.

SysIni

```
►►──SysIni(─┬─────────┬─,app,key,val,stem──)──────────────────────────►◄
            └─inifile─┘
```

Allows limited access to INI file variables. Variables are stored in the INI file under Application Names and their associated key names or keywords. You can use SysIni to share variables between applications or as a way of implementing GLOBALV in the Windows operating system. Be careful when changing application profile information.

Note: SysIni works on all types of data stored in an INI file (text, numeric, or binary).

When SysIni successfully sets or deletes key values, it returns ''. For a successful query, it returns the value of the specified application keyword.

SysIni may return the string `ERROR:` when an error occurs. Possible error conditions include:

- An attempt was made to query or delete an application/key pair that does not exist.
- An error opening the profile file occurred. You may have specified the current user or system INI file with a relative file specification. Make sure to use the full file specification (specify drive, path, and file name).

Parameters:

inifile	The name of the INI file with which you would like to work. The default is WIN.INI.
app	The application name or some other meaningful value with which you want to store keywords (some sort of data).
key	The name of a keyword to hold data.
val	The value to associate with the keyword of the specified application. This can be `'DELETE:'` or `'ALL:'`.
stem	The name of a REXX stem variable collection in which to store the resultant information. SysIni sets REXX variable *stem.0* to the number of elements returned and stores these elements in *stem.1* to *stem.n*.

Sysini has six modes. The modes and the syntax variations are as follows:

```
►►──SysIni(─┬─────────┬─,app,key,val)──────────────────────────────────────►◄
            └─inifile─┘
```

Sets a single key value.

```
►►──SysIni(─┬─────────┬─,app,key)──────────────────────────────────────────►◄
            └─inifile─┘
```

Queries a single key value.

```
►►──SysIni(─┬─────────┬─,app,key──,'DELETE:'──)────────────────────────────►◄
            └─inifile─┘
```

Deletes a single key.

```
►►──SysIni(─┬─────────┬─,app──┬─────────────┬─)────────────────────────────►◄
            └─inifile─┘       └─,'DELETE:'─┘
```

Deletes an application and all associated keys.

```
►►──SysIni(─┬─────────┬─,app──,'ALL:'──,'stem'──)──────────────────────────►◄
            └─inifile─┘
```

Queries names of all keys associated with a certain application.

```
►►──SysIni(─┬─────────┬─,'ALL:'──,'stem'──)────────────────────────────────►◄
            └─inifile─┘
```

Queries the names of all applications.

Examples:

```
/* Sample code segments */

/***  Save the user entered name under the key 'NAME' of *****
****  the application 'MYAPP'.                           ****/
pull name .
call SysIni , 'MYAPP', 'NAME', name /* Save the value        */
say  SysIni(, 'MYAPP', 'NAME')      /* Query the value       */
call SysIni , 'MYAPP'               /* Delete all MYAPP info */
exit
```

```
/****  Type all WIN.INI file information to the screen  *****/
call rxfuncadd sysloadfuncs, rexxutil, sysloadfuncs
call sysloadfuncs
call SysIni 'WIN.INI', 'All:', 'Apps.'
if Result \= 'ERROR:' then
do i = 1 to Apps.0
call SysIni 'WIN.INI', Apps.i, 'All:', 'Keys'
if Result \= 'ERROR:' then
do j=1 to Keys.0
val = SysIni('WIN.INI', Apps.i, Keys.j)
say left(Apps.i, 20) left(Keys.j, 20),
'Len=x'Left(d2x(length(val)),4) left(val, 20)
end
end
exit
```

SysLoadFuncs

```
►►──SysLoadFuncs──────────────────────────────────────────────────►◄
```

Loads all RexxUtil functions. After a REXX program calls SysLoadFuncs, the RexxUtil functions are available in all Windows operating system sessions.

SysLoadRexxMacroSpace

```
►►──SysLoadRexxMacroSpace(file)───────────────────────────────────►◄
```

Loads functions from a saved macrospace file. SysLoadRexxMacroSpace returns the RexxLoadMacroSpace return code.

Parameter:

file The file used to load functions into the REXX macrospace. SysSaveRexxMacroSpace must have created the file.

SysMkDir

```
►►──SysMkDir(dirspec)─────────────────────────────────────────────►◄
```

Creates a specified directory.

Parameter:

dirspec	The directory to be created.

Return codes:

0	Directory creation was successful.
2	File not found.
3	Path not found.
5	Access denied.
26	Not a DOS disk.
87	Invalid parameter.
108	Drive locked.
183	Directory already exists.
206	File name exceeds range.

Example:

```
/* Code */
call SysMkDir '~/rexx'
```

SysOpenEventSem

```
►►──SysOpenEventSem(name)──────────────────────────────────────────────►◄
```

Opens a Windows event semaphore. SysOpenEventSem returns a handle to the semaphore, or zero if an error occurred.

Parameter:

name	The name of the event semaphore created by SysCreateEventSem.

SysOpenMutexSem

```
►►──SysOpenMutexSem(name)──────────────────────────────────────────────►◄
```

Opens a Windows mutex semaphore. SysOpenMutexSem returns a handle to the semaphore, or zero if an error occurred.

Parameter:

name	The name of the mutex semaphore created by SysCreateMutexSem.

SysPostEventSem

```
▸▸──SysPostEventSem(handle)──────────────────────────────────────────▸◂
```

Posts a Windows event semaphore. SysPostEventSem returns the GetLastError return code of SetEvent.

Parameter:

handle	A handle returned from a previous SysCreateEventSem call.

Return codes:

0	No errors.
6	Invalid handle.

SysPulseEventSem

```
▸▸──SysPulseEventSem(handle)─────────────────────────────────────────▸◂
```

Posts and immediately resets an event semaphore. It sets the state of the event to signaled (available), releases any waiting threads, and resets it to nonsignaled (unavailable) automatically. If the event is manual, all waiting threads are released, the event is set to nonsignaled, and PulseEvent returns. If the event is automatic, a single thread is released, the event is set to nonsignaled, and PulseEvent returns. If no threads are waiting, or no threads can be released immediately, PulseEvent sets the state of the event to nonsignaled and returns.

SysPulseEventSem returns GetLastError of PulseEvent.

Parameter:

handle	The handle of a Windows event semaphore previously created by SysCreateEventSem.

SysQueryProcess

```
                              ┌─PID───┐
►►──SysQueryProcess("─┼─TID───┼─")───────────────────────────────────►◄
                              ├─PPRIO─┤
                              ├─TPRIO─┤
                              ├─PTIME─┤
                              └─TTIME─┘
```

Retrieves information about the current process or thread.

Parameter:

info The kind of information requested:

PID	Returns the process ID of the current process.
TID	Returns the thread ID of the current thread.
PPRIO	Returns the priority class of the current process.
TPRIO	Returns the relative priority of the current thread.
PTIME	Returns time information on the current process.
TTIME	Returns time information on the current thread.

Return codes:

- For PID or TID: an ID
- For PPRIO: "IDLE", "NORMAL", "HIGH", "REALTIME", or "UNKNOWN"
- For TPRIO: "IDLE", "LOWEST", "BELOW_NORMAL", "NORMAL", "ABOVE_NORMAL", "HIGHEST", "TIME_CRITICAL", or "UNKNOWN"
- For PTIME or TTIME: the creation date and time, the amount of time that the process executed in kernel mode, and the amount of time that the process executed in user mode

SysQueryRexxMacro

```
►►──SysQueryRexxMacro(name)──────────────────────────────────────────►◄
```

Queries the existence of a macrospace function. SysQueryRexxMacro returns the placement order of the macrospace function or a null string ("") if the function does not exist in the macrospace.

Parameter:

name The name of a function in the REXX macrospace.

SysReleaseMutexSem

```
►►──SysReleaseMutexSem(handle)─────────────────────────────────────────►◄
```

Releases a Windows mutex semaphore. SysReleaseMutexSem returns the GetLastError return code of ReleaseMutex.

Parameter:

handle	A handle returned from a previous SysCreateMutexSem call.

Return codes:

0	No errors.
6	Invalid handle.
105	Owner died.
288	Not owner.

SysReorderRexxMacro

```
►►──SysReorderRexxMacro(name,order)────────────────────────────────────►◄
```

Reorders a routine loaded in the REXX macrospace. SysReorderRexxMacro returns the RexxReorderMacro return code.

Parameters:

name	The name of a function in the macrospace.
order	The new macro search order. The order can be 'B' (Before) or 'A' (After).

SysRequestMutexSem

```
►►──SysRequestMutexSem(handle──┬──────────┬──)─────────────────────────►◄
                               └─,timeout─┘
```

Requests a Windows mutex semaphore. SysRequestMutexSem returns the WaitForSingleObject return code.

Parameters:

handle A handle returned from a previous SysCreateMutexSem call.

timeout
The time, in milliseconds, to wait on the semaphore. The default *timeout* is an infinite wait.

Return codes:

0	No errors.
6	Invalid handle.
103	Too many requests.
121	Error timeout.

SysResetEventSem

```
▸▸──SysResetEventSem(handle)──────────────────────────────────────▸◂
```

Resets a Windows event semaphore. SysResetEventSem returns the GetLastError return code of ResetEvent.

Parameter:

handle A handle returned from a previous SysCreateEventSem call.

Return codes:

0	No errors.
6	Invalid handle.

SysRmDir

```
▸▸──SysRmDir(dirspec)─────────────────────────────────────────────▸◂
```

Deletes a specified file directory without your confirmation.

Parameter:

dirspec The directory that should be deleted.

Return codes:

0	Directory removal was successful.

2	File not found.
3	Path not found.
5	Access denied or busy.
16	Current directory.
26	Not a DOS disk.
32	Sharing violation.
108	Drive locked.
123	Invalid name.
145	Directory not empty.
146	Is Subst Path.
147	Is Join Path.
206	File name exceeds range.

Example:

```
/* Code */
call SysRmDir 'c:\rexx'
```

SysSaveRexxMacroSpace

```
►►──SysSaveRexxMacroSpace(file)──────────────────────────────────────►◄
```

Saves the REXX macrospace. SysSaveRexxMacroSpace returns the RexxSaveMacroSpace return code.

Parameter:

file	The file used to save the functions in the REXX macrospace.

SysSearchPath

```
►►──SysSearchPath(path,filename──┬────────────┬──)───────────────────►◄
                                 └─,──option──┘
```

Searches the specified file path for the specified file. If the file is found, the search returns the full file specification of the first file found within the path, and then stops searching. If the file is not found, the search returns a null string.

Parameters:

path
: An environment variable name. The environment variable must contain a list of file directories. Examples are 'PATH' or 'DPATH'.

filename
: The file for which the path is to be searched.

option
: Specifies where the search starts.

  'C' Starts the search at the current directory and then along the specified path. This is the default.

  'N' Starts the search at the path, not at the current directory.

Example:

```
/* Code */
fspec = SysSearchPath('PATH', 'CMD.EXE')
say "CMD.EXE is located at" fspec

/* Output */
CMD.EXE is located at C:\WIN\CMD.EXE
```

SysSetFileDateTime

```
>>--SysSetFileDateTime(filename--+-------------------------------+--)----------><
                                 '-,--+---------+--+-----------+-'
                                      '-newdate-'  '-,-newtime-'
```

Modifies the "Last Modified" date and time of file *filename*. If no new date or time is specified the file date or time is set to the current time (TOUCH). If only the date is omitted, the "Last Modified" date remains unchanged. If only the time is omitted, the "Last Modified" time remains unchanged.

In Windows NT, the *filename* can also be a directory name.

The file that you want to change must not be opened by another process or must at least allow shared writes to update the time stamp.

Parameters:

filename
: The name of the file to be updated.

newdate
: The new date for the file, to be specified in the format YYYY-MM-DD, where YYYY > 1800.

newtime
: The new time for the file, to be specified in the format HH:MM:SS (24–hour format).

Return codes:

0 The file date and time were updated correctly.

–1 The update of the file date or time failed.

Example:

```
Call SysSetFileDateTime "MyFile.Log"  /* touch file */
Call SysSetFileDateTime "MyFile.Log", "1998-12-17"
Call SysSetFileDateTime "MyFile.Log",, "16:37:21"
Call SysSetFileDateTime "MyFile.Log", "1998-12-17", "16:37:21"

Call SysSetFileDateTime "C:\MyDir"    /* touch dir on Windows NT */
```

SysSetPriority

```
►►──SysSetPriority(class,delta)──────────────────────────────────────►◄
```

Changes the priority of the current process. A return code of 0 indicates no error.

Parameters:

class The new process priority class. The allowed classes are:

0 or "IDLE"
Idle time priority

1 or "NORMAL"
Regular priority

2 or "HIGH"
High or time-critical priority

3 or "REALTIME"
Real-time priority

delta The change applied to the process priority level. *delta* must be in the range -15 to +15. It can also be a symbolic name:

- "IDLE" for –15
- "LOWEST" for –2
- "BELOW_NORMAL" for –1
- "NORMAL" for 0
- "ABOVE_NORMAL" for 1
- "HIGHEST" for 2
- "TIME_CRITICAL" for 15

0 No errors.

307 Invalid priority class.

SysSleep

```
►►──SysSleep(secs)──────────────────────────────────────────────────►◄
```

Pauses a REXX program for a specified time interval.

Parameter:

secs The number of seconds for which the program is to be paused. You can specify up to seven decimal places in the number.

Example:

```
Say "Now paused for 2 seconds ..."
Call SysSleep 2
Say "Now paused for 0.1234567 seconds ..."
Call SysSleep 0.1234567

Call SysSleep 0.12345678 -- Error 40: Incorrect call to routine
```

SysStemCopy

```
►►──SysStemCopy──(──fromstem──,──tostem──────────────────────────────►

►──┬──────────────────────────────────────────────────────────┬──)──►◄
   └─,──┬──────┬──┬──────────────────────────────────────────┬─┘
        └─from─┘  └─,──┬────┬──┬────────────────────────────┬─┘
                       └─to─┘  └─,──┬───────┬──┬──────────────┬─┘
                                    └─count─┘  └─,──"──┬─I─┬──"─┘
                                                       └─O─┘
```

Copies items from the source stem to the target stem. Items in the source stem are copied starting at the *from* index (default is 1) into the target stem beginning at the *to* index (default is 1). The number of items to be copied to the target stem can be specified with the count. The default is to copy all items in the source stem.

You can also specify that the items are to be inserted into the target stem at the position and the existing items are shifted to the end.

This function operates only on stem arrays that specify the number of items in stem.0 and all items must be numbered from 1 to n without omitting an index.

Parameters:

fromstem
The name of the source stem.

tostem The name of the target stem.

from The first index in the source stem to be copied.

to The position at which the items are to be inserted in the target stem.

count The number of items to be copied or inserted.

insert Either of the following values:

I	Insert items.
O	Overwrite items.

Return codes:

0	The stem was copied successfully.
–1	Copying the stem failed.

Example:

```
Source.0 = 3
Source.1 = "Hello"
Source.2 = "from"
Source.3 = "REXX"
Call SysStemCopy "Source.", "Target."

Call SysStemCopy "Source.", "Target.", 1, 5, 2, "I"
```

SysStemDelete

```
►►─SysStemDelete(stem,startitem─┬─────────────┬─)────────────────────►◄
                                └─,─itemcount─┘
```

Deletes the specified item at the index *startitem* in the stem. If more than one item is to be deleted the *itemcount* must be specified. After deleting the requested items the stem is compacted, which means that items following the deleted items are moved to the vacant positions.

This function operates only on stem arrays that specify the number of items in stem.0 and all items must be numbered from 1 to n without omitting an index.

Parameters:

stem The name of the stem from which the item is to be deleted.

startitem
The index of the item to be deleted.

itemcount
The number of items to be deleted if more than one.

Return codes:

0 Deleting was successful.

–1 Deleting failed.

Example:

```
Call SysStemDelete "MyStem.", 5
Call SysStemDelete "MyStem.", 5, 4
```

SysStemInsert

```
►►──SysStemInsert(stem,position,value)──────────────────────────────►◄
```

Inserts a new item at *position* in the stem. All items in the stem following this position are shifted down by one position.

This function operates only on stem arrays that specify the number of items in stem.0 and all items must be numbered from 1 to n without omitting an index.

Parameters:

stem The name of the stem in which an item is to be inserted.

position
The index at which the new item is to be inserted.

value The value of the new item.

Return codes:

0 Inserting was successful.

–1 Inserting failed.

Example:

```
Call SysStemInsert "MyStem.", 5, "New value for item 5"
```

SysStemSort

```
                                 ┌─A─┐           ┌─C─┐             ┌─1─────┐
►►──SysStemSort──(──stem──,──"───┴─D─┴────"──,──"───┴─I─┴────"──,───┴─start─┴──────────►

          ┌─1────────┐
►──,──end──,───┴─firstcol─┴────,──lastcol──)──────────────────────────────────────►◄
```

Sorts all or the specified items in the stem. The items can be sorted in ascending or descending order and the case of the strings being compared can be respected or ignored. Sorting can be further narrowed by specifying the first and last item to be sorted or the columns used as sort keys. Because the sort uses a quick-sort algorithm, the order of sorted items according to the sort key is undetermined.

This function operates only on stems that specify the number of items in stem.0 and all items must be numbered from 1 to n without omitting an index.

Parameters:

stem	The name of the stem to be sorted.
order	Either "A" for ascending or "D" for descending. The default is "A".
type	The type of comparison: either "C" for case or "I" for ignore. The default is "C".
start	The index at which the sort is to start. The default is 1.
end	The index at which the sort is to end. The default is the last item.
firstcol	The first column to be used as sort key. The default is 1.
lastcol	The last column to be used as sort key. The default is the last column.

Return codes:

0	The sort was successful.
–1	The sort failed.

Example:

```
/* sort all elements descending, use cols 5 to 10 as key */
Call SysStemSort "MyStem.", "D",,,,5, 10

/* sort all elements ascending, ignore the case */
Call SysStemSort "MyStem.", "A", "I"

/* sort elements 10 to 20 ascending, use cols 1 to 10 as key */
Call SysStemSort "MyStem.",,,10, 20, 1, 10
```

SysSwitchSession

```
►►──SysSwitchSession(name)─────────────────────────────────────────►◄
```

Makes the named application the foreground application. SysSwitchSession returns GetLastError of SetForegroundWindow.

Parameter:

name The name of the application you want to be the foreground application.

SysSystemDirectory

```
►►──SysSystemDirectory()───────────────────────────────────────────►◄
```

Returns the Windows system directory.

SysTempFileName

```
►►──SysTempFileName(template──┬─────────┬──)───────────────────────►◄
                              └─,filter─┘
```

Returns a unique name for a file or directory that does not currently exist. If an error occurs or SysTempFileName cannot create a unique name from the template, it returns a null string (""). SysTempFileName is useful when a program requires a temporary file.

Parameters:

template
: The location and base form of the temporary file or directory name. The *template* is a valid file or directory specification with up to five filter characters.

filter
: The filter character used in *template*. SysTempFileName replaces each filter character in *template* with a numeric value. The resulting string represents a file or directory that does not exist. The default filter character is **?**.

Examples:

```
/* Code */
say SysTempFileName('C:\TEMP\MYEXEC.???')
say SysTempFileName('C:\TEMP\??MYEXEC.???')
say SysTempFileName('C:\MYEXEC@.@@@', '@')

/* Output */
C:\TEMP\MYEXEC.251
C:\TEMP\10MYEXEC.392
C:\MYEXEC6.019
```

SysTempFileName generates the filter character replacements with a random number algorithm. If the resulting file or directory already exists, SysTempFileName increments the replacement value until all possibilities have been exhausted.

SysTextScreenRead

```
►►──SysTextScreenRead(─row,column──┬──────┬──)─────────────────────────────►◄
                                   └─,len─┘
```

Reads characters from a specified screen location. These include any carriage return and linefeed characters if the number of character reads spans multiple lines.

Parameters:

row
: The row from which to start reading.

col
: The column from which to start reading.

len
: The number of characters to read. The default is to read to the end of the screen.

Limitations: This function reads in only screen characters and does not consider the color attributes of each character read. When restoring a character string to the screen with SAY or the CHAROUT built-in function, the previous color settings are lost.

Examples:

```
/* Reading the entire screen */
screen = SysTextScreenRead(0, 0)

/* Reading one line */
line = SysTextScreenRead(2, 0, 80)
```

SysTextScreenSize

```
►►──SysTextScreenSize()──────────────────────────────────────────────────►◄
```

Returns the size of the screen in the format: `row col`.

Example:

```
/* Code */
call RxFuncAdd 'SysTextScreenSize', 'RexxUtil', 'SysTextScreenSize'
parse value SysTextScreenSize() with row col
say 'Rows='row', Columns='col
```

SysUtilVersion

```
►►──SysUtilVersion()─────────────────────────────────────────────────────►◄
```

Returns a version number that identifies the current level of the REXX Utilities package. It can be used to verify the availability of certain functions.

Return code: The REXXUTIL version number in the format `n.mm`.

Examples:

Because this function was not part of the original packaging, a sample logic to check for a certain level of REXXUTIL can look as follows:

```
If RxFuncQuery("SysUtilVersion") = 1 |,
   SysUtilVersion() < "2.00" Then
     Say "Your REXXUTIL.DLL is not at the current level"
```

If a specific function should be used that was added at a later REXXUTIL level a similar check can be performed by querying this function as follows:

```
If RxFuncQuery("SysSetFileDateTime") = 1 Then
  Say "Your REXXUTIL.DLL is not at the current level"
```

SysVersion

```
►►──SysVersion()──────────────────────────────────────────────────►◄
```

Returns a string to identify the operating system and version. The first word of the returned string contains the identifier for the operating system and the second word the version: `WindowsNT x.xx` or `Windows95 x.xx`.

Return code: The operating system and version. Possible output for operating systems supported by Object REXX are:

```
Say SysVersion()   ->  "WindowsNT 4.00"
Say SysVersion()   ->  "WindowsNT 5.00"
```

Note: This function can be used to replace the operating-system-specific functions SysOS2Ver(), SysWinVer(), and SysLinVer().

SysVolumeLabel

```
►►──SysVolumeLabel("drive")───────────────────────────────────────►◄
```

Returns the label of the specified or the current drive.

Parameter:

drive The drive letter in the form `D:`. If omitted, the letter of the current drive is assumed.

SysWaitEventSem

```
►►──SysWaitEventSem(handle──┬──────────┬──)───────────────────────►◄
                            └─,timeout─┘
```

Waits on a Windows event semaphore. SysWaitEventSem returns the WaitForSingleObject return code.

Parameters:

handle A handle returned from a previous SysCreateEventSem call.

timeout
: The time, in milliseconds, to wait on the semaphore. The default *timeout* is an infinite wait.

Return codes:

0	No errors.
6	Invalid handle.
121	Timeout.

SysWaitNamedPipe

```
►►──SysWaitNamedPipe(name──┬──────────┬──)──────────────────────────────►◄
                           └─,timeout─┘
```

Performs a timed wait on a named pipe and returns the WaitNamedPipe return code.

Parameters:

name The name of the pipe in the form "\\servername\pipe\pipename."

timeout
: The number of microseconds to be waited. If you omit *timeout* or specify 0, SysWaitNamedPipe uses the default timeout value. To wait until the pipe is no longer busy, you can use a value of -1.

SysWinDecryptFile

```
►►──SysWinDecryptFile(filename)──────────────────────────────────────►◄
```

Decrypts a given file (Windows 2000 only).

Parameter:

filename
: The file to be decrypted.

Return codes:

0	Decryption was successful.
2	File not found.

4	Cannot open file.
5	Access denied.
82	Cannot decrypt.

SysWinEncryptFile

```
►►──SysWinEncryptFile(filename)──────────────────────────────────►◄
```

Encrypts a given file (Windows 2000 only).

Parameter:

filename
The file to be encrypted.

Return codes:

0	Encryption was successful.
2	File not found.
4	Cannot open file.
5	Access denied.
82	Cannot encrypt.

SysWinVer

```
►►──SysWinVer()──────────────────────────────────────────────────►◄
```

Returns a string specifying the Windows operating system version information in the form `x.xx`.

Chapter 10. Parsing

The parsing instructions are ARG, PARSE, and PULL (see “ARG” on page 43, “PARSE” on page 63, and “PULL” on page 69).

The data to be parsed is a *source string*. Parsing splits the data in a source string and assigns pieces of it to 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 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 the characters in the source string to specify where it is to be split. (See “Templates Containing String Patterns” on page 405 for details.)

Positional patterns
: Indicate the character positions at which the source string is to be split. (See “Templates Containing Positional (Numeric) Patterns” on page 407 for details.)

Parsing is essentially a two-step process:
1. Parse the source string into appropriate substrings using patterns.
2. Parse each substring into words.

Simple Templates for Parsing into Words

Here is a parsing instruction:

```
parse value 'time and tide' with var1 var2 var3
```

The template in this instruction is: `var1 var2 var3`. The data to be parsed 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 be parsed is a literal string, `time and tide`. In the next example, the source string is a variable.

```
/* PARSE VALUE using a variable as the source string to parse    */
string='time and tide'
parse value string with var1 var2 var3          /* same results */
```

PARSE VALUE does not convert lowercase a–z in the source string to uppercase A–Z. If you want to convert characters to uppercase, use PARSE UPPER VALUE. See "Using UPPER, LOWER, and CASELESS" on page 412 for a summary of the effect of parsing instructions on the case.

Note that if you specify the CASELESS option on a PARSE instruction, the string comparisons during the scanning operation are made independently of the alphabetic case. That is, a letter in uppercase is equal to the same letter in lowercase.

All of the parsing instructions assign the parts of a source string to the variables named in a template. There are various parsing instructions because of the differences in the nature or origin of source strings. For a summary of all the parsing instructions, see "Parsing Instructions Summary" on page 413.

The PARSE VAR instruction is similar to PARSE VALUE except that the source string to be parsed 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 the entire parsing: for parsing into words with simple templates and for parsing with templates containing patterns. Here is an example of 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. And it saves the overhead of unneeded variables.

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' */
```

Templates Containing String Patterns

A *string pattern* matches characters in the source string to indicate where to split it. A string pattern can be either of the following:

Literal string pattern
One or more characters within quotation marks.

Variable string pattern
A variable within parentheses with no plus (+), minus (-), or equal sign (=) before the left parenthesis. (See “Parsing with Variable Patterns” on page 411 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`
- 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 to 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 (see “Combining String and Positional Patterns” on page 415) in which a template with a string pattern does not omit matching data in the source string.) The pattern `', '` (with a blank) is used 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, 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 the data in the source string is to be split. 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 411 for details on variable positional patterns.)

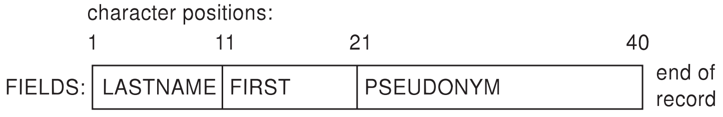
The number specifies the absolute character position at which the source string is to be split.

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`
- 11 through 20 into `variable2`
- 21 to the end into `variable3`

Positional patterns are probably most useful for working with a file of records, such as:



The following example uses this record structure:

```
/* Parsing with absolute positional patterns in template         */
record.1='Clemens   Samuel    Mark Twain          '
record.2='Evans     Mary Ann  George Eliot        '
record.3='Munro     H.H.      Saki                '
do n=1 to 3
```

```
  parse var record.n lastname 11 firstname 21 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end                         /* Says 'By George!' after record 2  */
```

The source string is split at character position 11 and at position 21. The language processor assigns characters 1 to 10 to `lastname`, characters 11 to 20 to `firstname`, and characters 21 to 40 to `pseudonym`.

The template could have been:

```
1 lastname 11 firstname 21 pseudonym
```

instead of

```
lastname 11 firstname 21 pseudonym
```

Specifying 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 are equal:

```
lastname  11 first  21 pseudonym
```

```
lastname =11 first =21 pseudonym
```

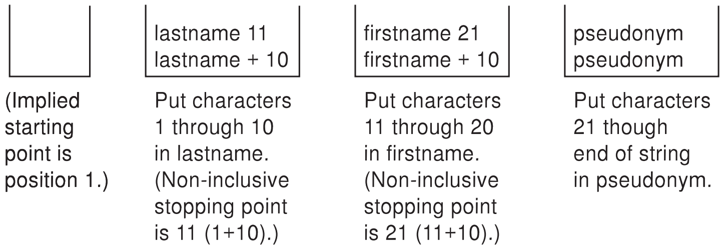
A *relative positional pattern* is a number with a plus (+) or minus (-) sign preceding it. It can also be a variable within parentheses, with a plus (+) or minus (-) sign preceding the left parenthesis; for details see “Parsing with Variable Patterns” on page 411.

The number specifies the relative character position at which the source string is to be split. 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" on page 415) when a string pattern precedes a variable name and a positional pattern follows the variable name. The templates from the examples of absolute and relative positional patterns give the same results.


lastname 11
lastname + 10
firstname 21
firstname + 10
pseudonym
pseudonym
(Implied starting point is position 1.)
Put characters 1 through 10 in lastname. (Non-inclusive stopping point is 11 (1+10).)
Put characters 11 through 20 in firstname. (Non-inclusive stopping point is 21 (11+10).)
Put characters 21 though end of string in pseudonym.

With positional patterns, a matching operation can 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"                           */
```

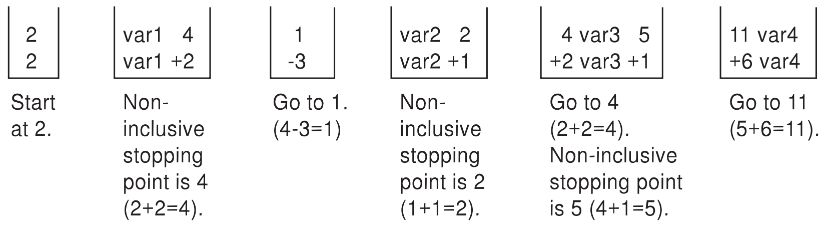
The absolute positional pattern 1 backs up to the first character in the source string.

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 previous two examples are equivalent.


2
2
Start at 2.
var1 4
var1 +2
Non-inclusive stopping point is 4 (2+2=4).
1
-3
Go to 1. (4-3=1)
var2 2
var2 +1
Non-inclusive stopping point is 2 (1+1=2).
4 var3 5
+2 var3 +1
Go to 4 (2+2=4). Non-inclusive stopping point is 5 (4+1=5).
11 var4
+6 var4
Go to 11 (5+6=11).

You can use templates with positional patterns to make several assignments:

```
/* Making several assignments                                   */
books='Silas Marner, Felix Holt, Daniel Deronda, Middlemarch'
parse var books 1 Eliot 1 Evans
/* Assigns the (entire) value of books to Eliot and to Evans.    */
```

Combining Patterns and Parsing into Words

If a template contains patterns that divide the source string into sections containing several 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` has 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 403.)

```
/* 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 403.)

Parsing with Variable Patterns

You might 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 character string 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 11/15/98 */
```

```
pull date
parse var date month 3 delim +1 day +2 (delim) year
/* Sets: month='11'; delim='/'; day='15'; year='98'  */
```

If an equal, a plus, or a minus sign precedes the left parenthesis, 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'    */
```

The positional pattern 6 is needed in the template for the following reason: 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, LOWER, and CASELESS

Specifying UPPER on any of the PARSE instructions converts lowercase a–z to uppercase A–Z before parsing.

The ARG instruction is a short form of PARSE UPPER ARG. The PULL instruction is 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 PARSE PULL instead of PULL or PARSE UPPER PULL.

Specifying LOWER on any of the PARSE instructions converts uppercase A–Z to lowercase a–z before parsing.

Specifying CASELESS means the comparisons during parsing are independent of the case—that is, a letter in uppercase is equal to the same letter in lowercase.

Parsing Instructions Summary

All parsing instructions assign parts of the source string to 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 LINEIN	Next line in the default input stream.
PULL PARSE PULL	The string at the head of the external data queue. (If the queue is empty, it uses default input, typically the terminal.)
PARSE SOURCE	System-supplied string giving information about the executing program.
PARSE VALUE	Expression between the keywords VALUE and 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 Several Strings” on page 414.

PARSE ARG is similar to ARG except that PARSE ARG does not convert alphabetic characters to uppercase before parsing.

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:                 */
                                    /* "Windows"                 */
```

PARSE VALUE example is on page 403.

PARSE VAR examples are throughout the chapter, starting with page 404.

PARSE VERSION

```
parse version . level .
say level               /* Displays: "Oryx 3.00 Jun 9 1993" */
```

PULL is similar to PARSE PULL except that PULL converts alphabetic characters to uppercase before parsing.

Advanced Topics in Parsing

This section includes parsing several strings and flow charts illustrating a conceptual view of parsing.

Parsing Several Strings

Only ARG and PARSE ARG can have more than one source string. To parse several strings, you can specify several 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 be parsed 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 several strings in a subroutine                    */
num='3'
musketeers="Porthos Athos Aramis D'Artagnan"
CALL Sub num,musketeers  /* Passes num and musketeers to sub    */
SAY total; say fourth /* Displays: "4" and " D'Artagnan"        */
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 several argument strings for parsing if:

- One REXX program calls another REXX program with the CALL instruction or a function call
- 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 subsequent commas) or contains no variable names, parsing proceeds to the next template and source string.

Combining String and Positional Patterns

There is a special case in which absolute and relative positional patterns do not work identically. Parsing with a template containing a string pattern skips the data in the source string that matches the pattern (see “Templates Containing String Patterns” on page 405). But a template containing the sequence string pattern, variable name, and relative position pattern does not skip 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 any data.          */
string='REstructured eXtended eXecutor'
parse var string var1 3 junk 'X' var2 +1 junk 'X' var3 +1 junk
say var1||var2||var3 /* Concatenates variables; displays: "REXX" */
```

Here is how this template works:

var1 3	junk 'X'	var2 +1	junk 'X'	var3 +1	junk
Put characters 1 through 2 in var1. (Stopping point is 3.)	Starting at 3, put characters up to (not including) first 'X' in junk.	Starting with first 'X' put 1 (+1) character in var2.	Starting with character after first 'X' put up to second 'X' in junk.	Starting with second 'X' put 1 (+1) character in var3.	Starting with character after second 'X' put rest in junk.
var1='RE'	junk= 'structured e'	var2='X'	junk= 'tended e'	var3='X'	junk= 'ecutor'

Conceptual Overview of Parsing

The following figures are to help you understand the concept of parsing.

The figures include the following terms:

string start
is the beginning of the source string (or substring).

string end
is the end of the source string (or substring).

length is the length of the source string.

match start
is in the source string and is the first character of the match.

match end
is in the source string. For a string pattern, it is the first character after the end of the match. For a positional pattern, it is the same as match start.

match position
is in the source string. For a string pattern, it is the first matching character. For a positional pattern, it is the position of the matching character.

token is a distinct syntactic element in a template, such as a variable, a period, a pattern, or a comma.

value is the numeric value of a positional pattern. This can be either a constant or the resolved value of a variable.

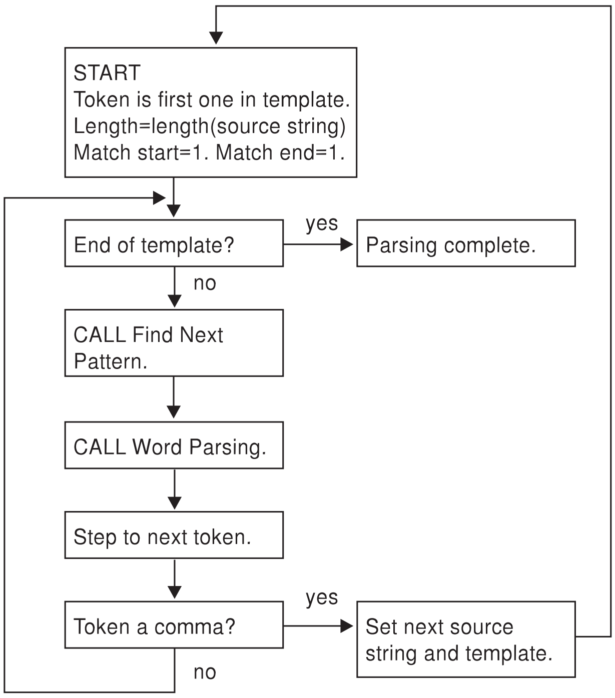

START
Token is first one in template.
Length=length(source string)
Match start=1. Match end=1.
End of template?
yes
Parsing complete.
no
CALL Find Next Pattern.
CALL Word Parsing.
Step to next token.
Token a comma?
yes
Set next source string and template.
no

Figure 13. Conceptual Overview of Parsing

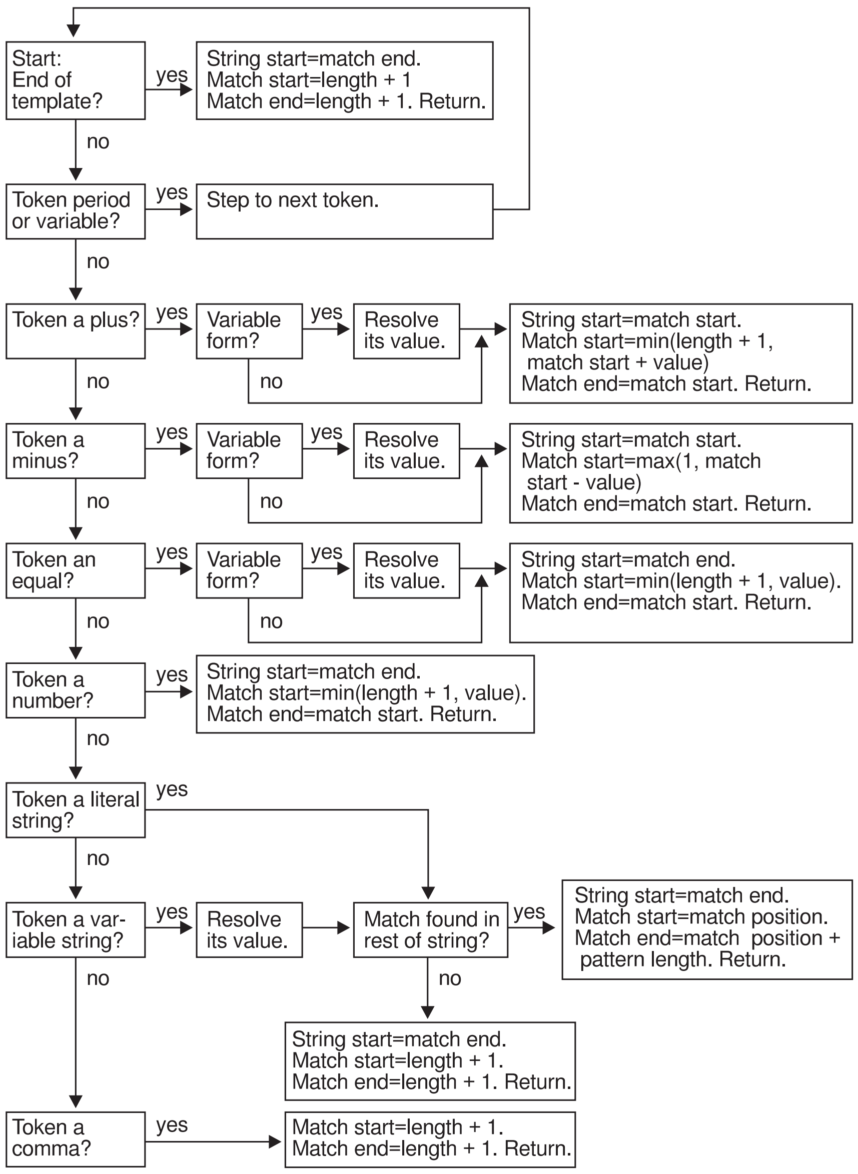

Start: End of template?
yes
String start=match end. Match start=length + 1 Match end=length + 1. Return.
no
Token period or variable?
yes
Step to next token.
no
Token a plus?
yes
Variable form?
yes
Resolve its value.
no
String start=match start. Match start=min(length + 1, match start + value) Match end=match start. Return.
no
Token a minus?
yes
Variable form?
yes
Resolve its value.
no
String start=match start. Match start=max(1, match start - value) Match end=match start. Return.
no
Token an equal?
yes
Variable form?
yes
Resolve its value.
no
String start=match end. Match start=min(length + 1, value). Match end=match start. Return.
no
Token a number?
yes
String start=match end. Match start=min(length + 1, value). Match end=match start. Return.
no
Token a literal string?
yes
no
Token a variable string?
yes
Resolve its value.
Match found in rest of string?
yes
String start=match end. Match start=match position. Match end=match position + pattern length. Return.
no
String start=match end. Match start=length + 1. Match end=length + 1. Return.
no
Token a comma?
yes
Match start=length + 1. Match end=length + 1. Return.

Figure 14. Conceptual View of Finding Next Pattern

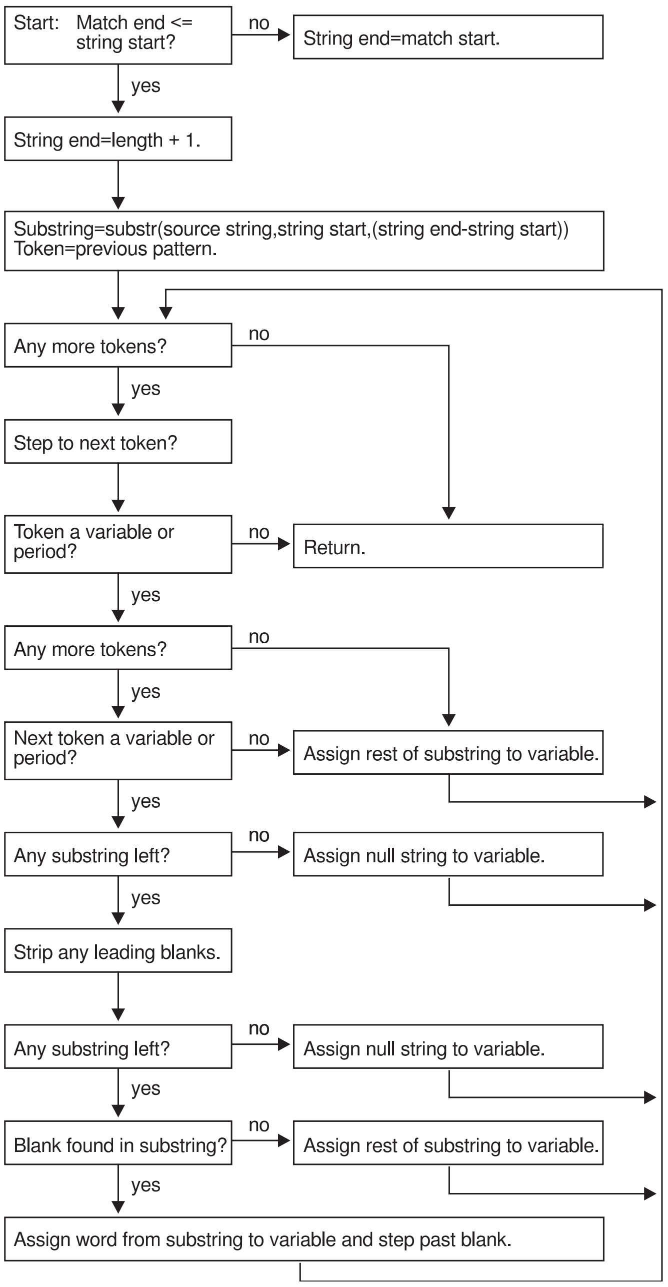

Start: Match end <= string start?
no
String end=match start.
yes
String end=length + 1.
Substring=substr(source string,string start,(string end-string start))
Token=previous pattern.
Any more tokens?
no
yes
Step to next token?
Token a variable or period?
no
Return.
yes
Any more tokens?
no
yes
Next token a variable or period?
no
Assign rest of substring to variable.
yes
Any substring left?
no
Assign null string to variable.
yes
Strip any leading blanks.
Any substring left?
no
Assign null string to variable.
yes
Blank found in substring?
no
Assign rest of substring to variable.
yes
Assign word from substring to variable and step past blank.

Figure 15. Conceptual View of Word Parsing

Note: The figures do not include error cases.

Chapter 11. Numbers and Arithmetic

This chapter gives an overview of the arithmetic facilities of the REXX language.

Numbers can be expressed flexibly. Leading and trailing blanks are permitted, and exponential notation can be used. Valid numbers are, for example:

```
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             */
```

A number in REXX is defined as follows:

```
>>──┬────────┬──┬──────────────────┬──┬─digits────────┬──┬────────┬──────────><
    └─blanks─┘  └─sign─┬────────┬──┘  ├─digits.digits─┤  └─blanks─┘
                       └─blanks─┘     ├─.digits───────┤
                                      └─digits.───────┘
```

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.

The *arithmetic operators* include addition (+), subtraction (-), multiplication (*), power (**), division (/), prefix plus (+), and prefix minus (-). In addition, it includes integer divide (%), which divides and returns the integer part, and remainder (//), which divides and returns the remainder. For examples of the arithmetic operators, see "Operator Examples" on page 424.

The result of an arithmetic operation is formatted as a character string according to specific rules. The most important rules are:

- Results are calculated up to a maximum number of significant digits. The default is 9, but you can alter it with the NUMERIC DIGITS instruction. Thus, if a result requires more than 9 digits, it is rounded to 9 digits. For example, the division of 2 by 3 results in 0.666666667.

- Except for division and power, trailing zeros are preserved. For example:

  ```
  2.40 + 2    ->    4.40
  2.40 - 2    ->    0.40
  2.40 * 2    ->    4.80
  2.40 / 2    ->    1.2
  ```

  If necessary, you can remove trailing zeros with the STRIP method (see "STRIP" on page 231), the STRIP function (see "STRIP" on page 344), 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 */
  ```

Precision

Precision is the maximum number of significant digits that can result from an operation. This is controlled by the instruction:

```
►►──NUMERIC DIGITS──┬────────────┬──;──────────────────────────────────►◄
                    └─expression─┘
```

The *expression* is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) of a calculation. 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`.

NUMERIC DIGITS can set values smaller than nine. However, use small values with care because the loss of precision and rounding 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).

Before every arithmetic operation, the terms 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 improves accuracy because it is inspected at the end of an operation, when a number is rounded to the required precision. When a number is truncated, the LOSTDIGITS condition is raised if a SIGNAL ON LOSTDIGITS condition trap is active. The operation is then carried out under up to double that precision. When the operation is completed, the result is rounded, if necessary, to the precision specified by the NUMERIC DIGITS instruction.

The values are rounded as follows: 5 through 9 are rounded up, and 0 through 4 are rounded down.

Power

The ** *(power) operator* raises a number to a power, which can 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 "Limits and Errors when REXX Uses Numbers Directly" on page 427. If negative, the absolute value of the power is used, and the result is inverted (that is, the number 1 is divided by the result). For calculating the power, the number is multiplied by itself for the number of times expressed by the power. Trailing zeros are then removed as though the result were divided by 1.

Integer Division

The % *(integer divide) operator* divides two numbers and returns the integer part of the result. The result is calculated by repeatedly subtracting the divisor from the dividend as long as 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.

If the result cannot be expressed as a whole number, the operation is in error and fails—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.

Remainder

The // *(remainder) operator* returns the remainder from an integer division and is defined to be the residue of the dividend after integer division. 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 fails, the remainder cannot be calculated.

Operator Examples

```
/* 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
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
```

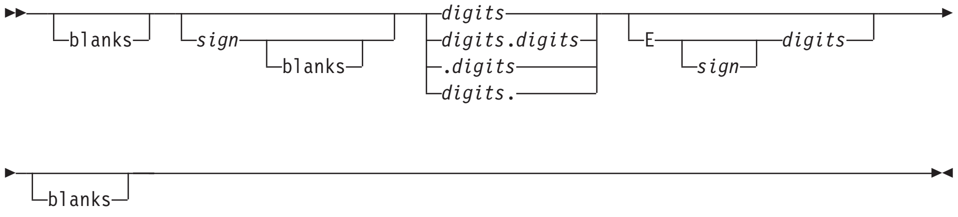
Exponential Notation

For both large and small numbers, an exponential notation can be useful. For example:

```
numeric digits 5
say 54321*54321
```

would display `2950800000` in the long form. Because this is misleading, the result is expressed as `2.9508E+9` instead.

The definition of numbers is, therefore, extended as follows:



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, such as 0E123 (0 raised to the 123 power) and 1E342 (1 raised to the 342 power). Also, 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 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, the exponential form is used. The exponential form the language processor generates always has a sign following the E to improve readability. If the exponent is 0, the exponential part is omitted—that is, an exponential part of E+0 is not generated.

You can explicitly convert numbers to exponential form, or force them to be displayed in the long form, by using the FORMAT built-in function (see “FORMAT” on page 325).

Scientific notation is a form of exponential notation that adjusts the power of ten so that the number contains only one nonzero digit before the decimal point. *Engineering notation* is a form of exponential notation in which up to three digits appear before the decimal point, and the power of ten is always a multiple of three. The integer part can, therefore, range from 1 through 999. You can control whether scientific or engineering notation is used with the following instruction:

```
                  ┌─SCIENTIFIC──────────┐
►►──NUMERIC FORM──┼─────────────────────┼─;──────────────────────────────►◄
                  ├─ENGINEERING─────────┤
                  └─┬───────┬─expression┘
                    └─VALUE─┘
```

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 Comparisons

The comparison operators are listed in “Comparison” on page 21. You can use any of them 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.

Numeric values are compared 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.

Fuzz affects the comparison of two numbers. It controls how much two numbers can differ and still be considered equal in a comparison. The FUZZ value is set by the following instruction:

```
►►──NUMERIC FUZZ──┬────────────┬──;──────────────────────────────────────►◄
                  └─expression─┘
```

expression must result in a positive whole number or zero. The default is `0`.

Fuzz is to temporarily reduce the value of DIGITS. That is, the numbers are subtracted with a precision of DIGITS minus FUZZ digits during the comparison. The FUZZ setting must always be less than DIGITS.

If, for example, 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"    */
```

Limits and Errors when REXX Uses Numbers Directly

When REXX uses numbers directly, that is, numbers that have not been involved in an arithmetic operation, they are rounded, if necessary, according to the setting of NUMERIC DIGITS.

The following table shows which numbers must be whole numbers and what their limits are:

Power values (right-hand operand of the power operator)	999999999
Values of *exprr* and *exprf* in the DO instruction	The current numeric precision (up to 999999999)
Values given for DIGITS or FUZZ in the NUMERIC instruction	999999999 (Note: FUZZ must always be less than DIGITS.)
Positional patterns in parsing templates	999999999
Number given for *option* in the TRACE instruction	999999999

When REXX uses numbers directly, the following types of errors can occur:

- Overflow or underflow.

  This error occurs if the exponential part of a result exceeds the range that the language processor can handle, when the result is formatted according to the current settings of NUMERIC DIGITS and NUMERIC FORM. The language defines a minimum capability for the exponential part, namely the largest number that can be expressed as an exact integer in default precision. Because the default precision is `9`, you can use exponents in the range `-999999999` through `999999999`.

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

  Storage is needed for calculations and intermediate results, and if an arithmetic operation fails because of lack of storage. This is considered as a terminating error.

Chapter 12. Conditions and Condition Traps

A *condition* is an 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 45 and "SIGNAL" on page 77).

```
►►──┬─CALL───┬──┬─OFF──┬─condition──────────┬──────────────────────────┬─;──────────►◄
    └─SIGNAL─┘  │      └─USER─usercondition─┘                          │
                └─ON──┬─condition──────────┬──┬──────────────────┬─────┘
                      └─USER─usercondition─┘  └─NAME─trapname────┘
```

condition, *usercondition*, 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* or *usercondition* occurs, control passes to the routine or label *trapname* if you have specified *trapname*. Otherwise, control passes to the routine or label *usercondition* or *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 only be an internal label.

The conditions and their corresponding events that can be trapped are:

ANY traps any condition that a more specific condition trap does not trap. For example, if NOVALUE is raised and there is no NOVALUE trap enabled, but there is a SIGNAL ON ANY trap, the ANY trap is called for the NOVALUE condition. For example, a CALL ON ANY trap is ignored if NOVALUE is raised because CALL ON NOVALUE is not allowed.

ERROR

raised if a command indicates an error condition upon return. It is also raised if any command indicates failure and none of the following is active:
- CALL ON FAILURE
- SIGNAL ON FAILURE

- CALL ON ANY
- SIGNAL ON ANY

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.

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.

An attempt to enter a command to an unknown subcommand environment also raises a FAILURE condition.

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 processed when the external interruption occurred. When a REXX program is running in a full-screen or command prompt session, the Ctrl+Break key sequence raises the halt condition. However, if Ctrl+Break is pressed while a command or non-REXX external function is processing, the command or function ends.

Notes:

1. Application programs that use the REXX language processor might use the RXHALT exit or the RexxStart programming interface to halt the execution of a REXX macro. (See the *Object REXX for Windows: Programming Guide* for details about exits.)
2. Only SIGNAL ON HALT or CALL ON HALT can trap error 4, described in “Appendix C. Error Numbers and Messages” on page 527.

LOSTDIGITS
raised if a number used in an arithmetic operation has more digits than the current setting of NUMERIC DIGITS. Leading zeros are not counted in this comparison. You can specify the LOSTDIGITS condition only for SIGNAL ON.

NOMETHOD
raised if an object receives a message for which it has no method defined, and the object does not have an UNKNOWN method. You can specify the NOMETHOD condition only for SIGNAL ON.

NOSTRING
raised when the language processor requires a string value from an

object and the object does not directly provide a string value. See "Required String Values" on page 105 for more information. You can specify the NOSTRING condition only for SIGNAL ON.

NOTREADY
raised if an error occurs during an input or output operation. See "Errors during Input and Output" on page 472. 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
- The *name* following the VAR subkeyword of a PARSE instruction
- A variable reference in a parsing template, an EXPOSE instruction, a PROCEDURE instruction, or a DROP instruction
- A method selection override specifier in a message term

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:

- True syntax errors
- "Run-time" errors (such as attempting an arithmetic operation on nonnumeric terms)
- Syntax errors propagated from higher call or method invocation levels
- Untrapped HALT conditions
- Untrapped NOMETHOD conditions

You can specify this condition only for SIGNAL ON.

Notes:

1. SIGNAL ON SYNTAX cannot trap the errors 3 and 5.
2. SIGNAL ON SYNTAX can trap the errors 6 and 30 only if they occur during the execution of an INTERPRET instruction.

For information on these errors, refer to “Appendix C. Error Numbers and Messages” on page 527.

USER
: raised if a condition specified on the USER option of CALL ON or SIGNAL ON occurs. USER conditions are raised by a RAISE instruction that specifies a USER option with the same *usercondition* name. The specified *usercondition* can be any symbol, including those specified as possible values for *condition*.

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, and any OFF reference disables the trap for CALL or SIGNAL.

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 NOMETHOD, a SYNTAX condition is raised with the appropriate REXX error number.
- For SYNTAX conditions, the clause in error is terminated, and a SYNTAX condition is propagated to each CALL instruction, INTERPRET instruction, message instruction, or clause with function or message invocations active at the time of the error, terminating each instruction if a SYNTAX trap is not active at the instruction level. If the SYNTAX condition is not trapped at any of the higher levels, processing stops, and a message (see “Appendix C. Error Numbers and Messages” on page 527) 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, 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 (for example, ERROR or FAILURE) 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, and the caller usually receives an indication of failure.

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 SIGNAL proceeds as usually (see "SIGNAL" on page 77).

  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, a usual syntax error termination occurs if the SIGNAL ON SYNTAX label name is not found.
- If the action taken is a CALL, the `CALL` ***`trapname`*** proceeds in the usual way (see "CALL" on page 45) when the instruction completes. The call does not affect the special variable RESULT. If the routine should RETURN any data, that data is ignored.

  When 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 and FAILURE) or (for the other conditions) any action (including the updating of the condition information) is delayed until one of the following events occurs:

  1. A CALL ON or SIGNAL ON for the delayed condition is processed. In this case, a CALL or SIGNAL takes place immediately after the new CALL ON or SIGNAL ON instruction has been processed.
  2. A CALL OFF or SIGNAL OFF for the delayed condition is processed. In this case, the condition trap is disabled and the default action for the condition occurs at the end of the CALL OFF or SIGNAL OFF instruction.
  3. A RETURN is made from the subroutine. In this case, the condition trap is no longer delayed and the subroutine is called again immediately.

  On RETURN from the CALL, the original flow of execution is resumed, that is, the flow is not affected by the CALL.

  Notes:

  1. 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 can

only be 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 traps for which CALL ON is enabled can only occur at clause boundaries. If these conditions arise in the middle of an INTERPRET instruction, execution of INTERPRET can be interrupted and resumed later. Similarly, other instructions, for example DO or SELECT, can be temporarily interrupted by a CALL at a clause boundary.

2. 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 “CALL” on page 45 for details of other information that is saved during a subroutine call.
3. 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.
4. 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 the exit from the program but is trapped specially and then ignored after a message is given.
5. The system interface detects certain execution errors either before the 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, several labels are allowed before another type of clause.

Condition Information

When a 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 313).

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

- A descriptive string (see "Descriptive Strings") associated with that condition
- Optional additional object information (see "Additional Object Information" on page 436)

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 and across method invocations. Therefore, a routine called by 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.
LOSTDIGITS	The number with excessive digits that caused the LOSTDIGITS condition.
NOMETHOD	The name of the method that could not be found.
NOSTRING	The readable string representation of the object causing the NOSTRING condition.
NOTREADY	The name of the stream being manipulated when the error occurred and the NOTREADY condition was raised. If the stream was a default stream with no defined name, then the null string might be returned.
NOVALUE	The derived name of the variable whose attempted reference caused the NOVALUE condition.
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.
USER	Any string specified by the DESCRIPTION option of the RAISE instruction that raised the condition. If a description string was not specified, a null string is used.

Additional Object Information

The language processor can provide additional information, depending on the condition trapped:

NOMETHOD	The object that raised the NOMETHOD condition.
NOSTRING	The object that caused the NOSTRING condition.
NOTREADY	The stream object that raised the NOTREADY condition.
SYNTAX	An array containing the objects substituted into the secondary error message (if any) for the syntax error. If the message did not contain substitution values, a zero element array is used.
USER	Any object specified by an ADDITIONAL or ARRAY option of the RAISE instruction that raised the condition.

The Special Variable RC

When an ERROR or FAILURE condition is trapped, the REXX special variable RC is set to the command return code before control is transferred to the target label (whether by CALL or by SIGNAL).

Similarly, when SIGNAL ON SYNTAX traps a SYNTAX condition, the special variable RC is set to the syntax error number before control is transferred to the target label.

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. If 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. The setting of SIGL is especially useful after a SIGNAL ON SYNTAX trap when the number of the line in error can be used, for example, to control a text editor. Typically, code following the SYNTAX label can PARSE SOURCE to find the source of the data and then call an editor to edit the source file, positioned at the line in error. Note that in this case you might have to run the program again before any changes made in the editor can take effect.

Alternatively, SIGL can 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.

Condition Objects

A *condition object* is a directory returned by the Object option of the CONDITION built-in function. This directory contains all information currently available on a trapped condition. The information varies with the trapped condition. The NIL object is returned for any entry not available to the condition. The following entries can be found in a condition object:

ADDITIONAL
The additional information object associated with the condition. This is the same object that the Additional option of the CONDITION built-in function returns. The ADDITIONAL information may be specified with the ADDITIONAL or ARRAY options of the RAISE instruction.

DESCRIPTION
The string describing the condition. The Description option of the CONDITION built-in function also returns this value.

INSTRUCTION
The keyword for the instruction executed when the condition was trapped, either `CALL` or `SIGNAL`. The Instruction option of the CONDITION built-in function also returns this value.

CONDITION
The name of the trapped condition. The Condition name option of the CONDITION built-in function also returns this value.

RESULT
Any result specified on the RETURN or EXIT options of a RAISE instruction.

RC
The major REXX error number for a SYNTAX condition. This is the same error number assigned to the special variable RC.

CODE
The detailed identification of the error that caused a SYNTAX condition. This number is a nonnegative number in the form *nn.nnn*. The integer portion is the REXX major error number (the same value as the RC entry). The fractional portion is a subcode that gives a precise indication of the error that occurred.

ERRORTEXT	The primary error message for a SYNTAX condition. This is the same message available from the ERRORTEXT built-in function.
MESSAGE	The secondary error message for a SYNTAX condition. The message also contains the content of the ADDITIONAL information.
POSITION	The line number in source code at which a SYNTAX condition was raised.
PROGRAM	The name of the program where a SYNTAX condition was raised.
TRACEBACK	A single-index list of formatted traceback lines.
PROPAGATED	The value `0` (false) if the condition was raised at the same level as the condition trap or the value `1` (true) if the condition was reraised with RAISE PROPAGATE.

Chapter 13. Concurrency

Conceptually, each REXX object is like a small computer with its own processor to run its methods, its memory for object and method variables, and its communication links to other objects for sending and receiving messages. This is *object-based concurrency*. It lets more than one method run at the same time. Any number of objects can be active (running) at the same time, exchanging messages to communicate with, and synchronize, each other.

Early Reply

Early reply provides concurrent processing. A running method returns control, and possibly a result, to the point from which it was called; meanwhile it continues running. The following figure illustrates this concept.

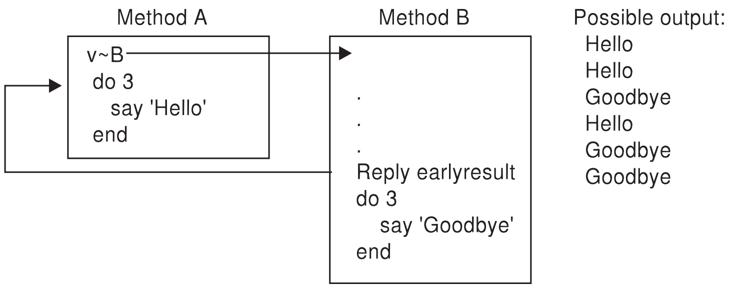


Figure 16. Early Reply

Method A includes a call to Method B. Method B contains a REPLY instruction. This returns control and a result to method A, which continues processing with the line after the call to Method B. Meanwhile, Method B also continues running.

The chains of execution represented by method A and method B are called *activities*. An activity is a thread of execution that can run methods concurrently with methods on other activities.

An activity contains a stack of *invocations* that represent the REXX programs running on the activity. An invocation can be a main program invocation, an internal function or subroutine call, an external function or subroutine call, an INTERPRET instruction, or a message invocation. An invocation is activated when an executable unit is invoked and removed (popped) when execution completes. In Figure 16, the programs begins with a single activity. The activity contains a single invocation, method A. When method A invokes method B, a second invocation is added to the activity.

When method B issues a REPLY, a new activity is created (activity 2). Method B's invocation is removed from activity 1, and pushed on to activity 2. Because activities can execute concurrently, both method A and method B continue processing. The following figures illustrate this concept.

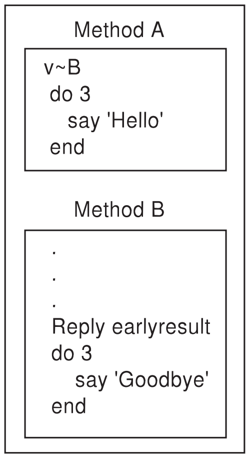


Figure 17. Before REPLY

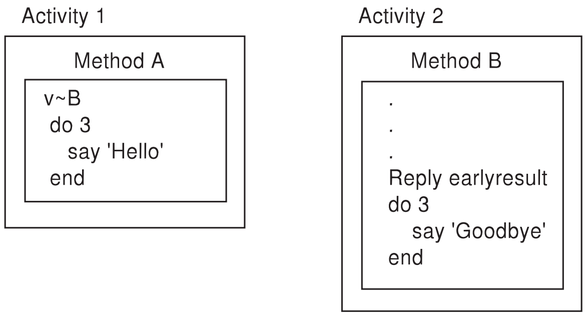


Figure 18. After REPLY

Here is an example of using early reply to run methods concurrently.

```
/* Example of early reply */

object1 = .example~new
object2 = .example~new

say object1~repeat(10, 'Object 1 running')
say object2~repeat(10, 'Object 2 running')
say 'Main ended.'
```

```
exit

::class example
::method repeat
use arg reps,msg
reply 'Repeating' msg',' reps 'times.'
do reps
  say msg
end
```

Message Objects

A message object (see “The Message Class” on page 170) is an intermediary between two objects that enables concurrent processing. All objects inherit the START method (page 173) from the object class. To obtain a message object, an object sends a START message to the object to which the message object will convey a message. The message is an argument to the START message as in the following example:

```
a=p~start('REVERSE')
```

This line of code creates a message object, `A`, and sends it a start message. The message object then sends the REVERSE message to object `P`. Object `P` receives the message, performs any needed processing, and returns a result to message object `A`. Meanwhile the object that obtained message object `A` continues its processing. When message object `A` returns, it does not interrupt the object that obtained it. It waits until this object requests the information. Here is an example of using a message object to run methods concurrently.

```
/* Example of using a message object */

object1 = .example~new
object2 = .example~new

a = object1~start('REPEAT',10,'Object 1 running')
b = object2~start('REPEAT',10,'Object 2 running')

say a~result
say b~result
say 'Main ended.'
exit

::class example
::method repeat
use arg reps,msg
do reps
  say msg
end
return 'Repeated' msg',' reps 'times.'
```

Default Concurrency

The instance methods of a class use the EXPOSE instruction to define a set of object variables. This collection of variables belonging to an object is called its object variable pool. The methods a class defines and the variables these methods can access is called a scope. REXX's default concurrency exploits the idea of scope. The object variable pool is a set of object subpools, each representing the set of variables at each scope of the inheritance chain of the class from which the object was created. Only methods at the same scope can access object variables at any particular scope. This prevents any name conflicts between classes and subclasses, because the object variables for each class are in different scopes.

If you do not change the defaults, only one method of a given scope can run on a single object at a time. Once a method is running on an object, the language processor blocks other methods on other activities from running in the same object at the same scope until the method that is running completes. Thus, if different activities send several messages within a single scope to an object the methods run sequentially.

The next example shows how the default concurrency works.

```
/* Example of default concurrency for methods of different scopes */

object1 = .subexample~new

say object1~repeat(8, 'Object 1 running call 1')   /* These calls run */
say object1~repeater(8, 'Object 1 running call 2') /*   concurrently  */
say 'Main ended.'
exit

::class example
::method repeat
use arg reps,msg
reply 'Repeating' msg',' reps 'times.'
do reps
  say msg
end

::class subexample subclass example
::method repeater
use arg reps,msg
reply 'Repeating' msg',' reps 'times.'
do reps
  say msg
end
```

The preceding example produces output such as the following:

```
Repeating Object 1 running call 1, 8 times.
Object 1 running call 1
Repeating Object 1 running call 2, 8 times.
```

```
Object 1 running call 1
Object 1 running call 2
Main ended.

Object 1 running call 1
Object 1 running call 2
Object 1 running call 1
Object 1 running call 2
Object 1 running call 1
Object 1 running call 2
Object 1 running call 1
Object 1 running call 2
Object 1 running call 1
Object 1 running call 2
Object 1 running call 1
Object 1 running call 2
Object 1 running call 2
```

The following example shows that methods of the same scope do not run concurrently by default.

```
/* Example of methods with the same scope not running concurrently*/

object1 = .example~new

say object1~repeat(10,'Object 1 running call 1') /* These calls   */
say object1~repeat(10,'Object 1 running call 2') /* cannot run    */
say 'Main ended.'                                /* concurrently. */
exit

::class example
::method repeat
use arg reps,msg
reply 'Repeating' msg',' reps 'times.'
do reps
  say msg
end
```

The REPEAT method includes a REPLY instruction, but the methods for the two REPEAT messages in the example cannot run concurrently. This is because REPEAT is called twice at the same scope and requires exclusive access to the object variable pool. The REPLY instruction causes the first REPEAT message to transfer its exclusive access to the object variable pool to a new activity and continue execution. The second REPLY message also requires exclusive access and waits until the first method completes.

If the original activity has more than one method active (nested method calls) with exclusive variable access, the first REPLY instruction is unable to transfer its exclusive access to the new activity and must wait until the exclusive access is again available. This may allow another method on the same object to run while the first method waits for exclusive access.

Sending Messages within an Activity

Whenever a message is invoked on an object, the activity acquires exclusive access (a lock) for the object's scope. Other activities that send messages to the same object that required the locked scope waits until the first activity releases the lock.

Suppose object A is running method Y, which includes:

```
self~z
```

Sequential processing does not allow method Z to begin until method Y has completed. However, method Y cannot complete until method Z runs. A similar situation occurs when a subclass's overriding method does some processing and passes a message to its superclasses' overriding method. Both cases require a special provision: If an invocation running on an activity sends another message to the same object, this method is allowed to run because the activity has already acquired the lock for the scope. This allows nested, nonconcurrent method invocations on a single activity without causing a deadlock situation. The language processor regards these additional messages as subroutine calls.

Here is an example showing the special treatment of single activity messages. The REPEATER and REPEAT methods have the same scope. REPEAT runs on the same object at the same time as the REPEATER method because a message to SELF runs the REPEAT method. The language processor treats this as a subroutine call rather than as concurrently running two methods.

```
/* Example of sending message to SELF */

object1 = .example~new
object2 = .example~new

say object1~repeater(10, 'Object 1 running')
say object2~repeater(10, 'Object 2 running')

say 'Main ended.'
exit

::class example
::method repeater
use arg reps,msg
reply 'Entered repeater.'
say self~repeat(reps,msg)
::method repeat
use arg reps,msg
do reps
  say msg
end
return 'Repeated' msg',' reps 'times.'
```

The activity locking rules also allow indirect object recursion. The following figure illustrates indirect object recursion.

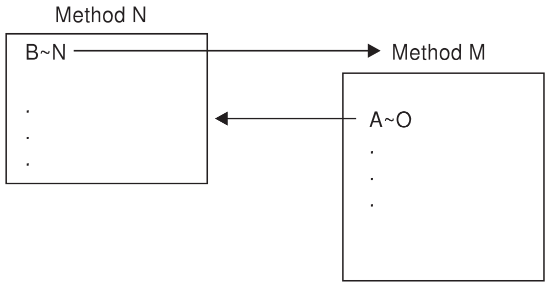


Figure 19. Indirect Object Recursion

Method M in object A sends object B a message to run method N. Method N sends a message to object A, asking it to run method O. Meanwhile, method M is still running in object A and waiting for a result from method N. A deadlock would result. Because the methods are all running on the same activity, no deadlock occurs.

Using Additional Concurrency Mechanisms

REXX has additional concurrency mechanisms that can add full concurrency so that more than one method of a given scope can run in an object at a time:

- The SETUNGUARDED method of the Method class and the UNGUARDED option of the METHOD directive provide unconditional concurrency
- GUARD OFF and GUARD ON control a method’s exclusive access to an object’s scope

SETUNGUARDED Method and UNGUARDED Option

The SETUNGUARDED method of the Method class and the UNGUARDED option of the ::METHOD directive control locking of an object's scope when a method is invoked. Both let a method run even if another method is active on the same object.

Use the SETUNGUARDED method or UNGUARDED option only for methods that do not need exclusive use of their object variable pool, that is, methods whose execution can interleave with another method's execution without affecting the object's integrity. Otherwise, concurrent methods can produce unexpected results.

To use the SETUNGUARDED method for a method you have created with the NEW method of the Method class, you specify:

```
methodname~SETUNGUARDED
```

(See “SETUNGUARDED” on page 177 for details about SETUNGUARDED.)

Alternately, you can define a method with the ::METHOD directive, specifying the UNGUARDED option:

```
::METHOD methodname UNGUARDED
```

GUARD ON and GUARD OFF

You might not be able to use the SETUNGUARDED method or UNGUARDED option in all cases. A method might need exclusive use of its object variables, then allow methods on other activities to run, and perhaps later need exclusive use again. You can use GUARD ON and GUARD OFF to alternate between exclusive use of an object’s scope and allowing other activities to use the scope.

By default, a method must wait until a currently running method is finished before it begins. GUARD OFF lets another method (running on a different activity) that needs exclusive use of the same object variables become active on the same object. See “GUARD” on page 55 for more information.

Guarded Methods

Concurrency requires the activities of concurrently running methods to be synchronized. Critical data must be safeguarded so diverse methods on other activities do not perform concurrent updates. *Guarded methods* satisfy both these needs.

A guarded method combines the UNGUARDED option of the ::METHOD directive or the SETUNGUARDED method of the Method class with the GUARD instruction.

The UNGUARDED option and the SETUNGUARDED method both provide unconditional concurrency. Including a GUARD instruction in a method makes concurrency conditional:

```
GUARD ON WHEN expression
```

If the *expression* on the GUARD instruction evaluates to `1` (true), the method continues to run. If the *expression* on the GUARD instruction evaluates to `0` (false), the method does not continue running. GUARD reevaluates the *expression* whenever the value of an exposed object variable changes. When the expression evaluates to 1, the method resumes running. You can use GUARD to block running any method when proceeding is not safe. (See “GUARD” on page 55 for details about GUARD.)

Note: It is important to ensure that you use an expression that can be fulfilled. If the condition expression cannot be met, GUARD ON WHEN puts the program in a continuous wait condition. This can occur in particular when several activities run concurrently. In this case, a second activity can make the condition expression invalid before GUARD ON WHEN can use it.

To avoid this, ensure that the GUARD ON WHEN statement is executed before the condition is set to true. Keep in mind that the sequence of running activities is not determined by the calling sequence, so it is important to use a logic that is independent of the activity sequence.

Additional Examples

The following example uses REPLY in a method for a write-back cache.

```
/* Method Write_Back                                 */
use arg data          /* Save data to be written    */
reply 0               /* Tell the sender all was OK */
self~disk_write(data)  /* Now write the data         */
```

The REPLY instruction returns control to the point at which method Write_Back was called, returning the result `0`. The caller of method Write_Back continues processing from this point; meanwhile, method Write_Back also continues processing.

The following example uses a message object. It reads a line asynchronously into the variable `nextline`:

```
mymsg = infile~start('READLINE') /* Gets message object to carry    */
/* message to INFILE                                                */
/* do other work */
nextline=mymsg~result           /* Gets result from message object */
```

This creates a message object that waits for the read to finish while the sender continues with other work. When the line is read, the `mymsg` message object obtains the result and holds it until the sender requests it.

Semaphores and monitors (bounded buffers) synchronize concurrency processes. Giving readers and writers concurrent access is a typical concurrency problem. The following sections show how to use guarded methods to code semaphore and monitor mechanisms and to provide concurrency for readers and writers.

Semaphores

A semaphore is a mechanism that controls access to resources, for example, preventing simultaneous access. Synchronization often uses semaphores. Here is an example of a semaphore class:

```
/******************************************************************************/
/*  A REXX Semaphore Class.                                                   */
/*                                                                            */
/* This file implements a semaphore class in REXX.  The class is defined to   */
/* the Global REXX Environment.  The following methods are defined for        */
/* this class:                                                                */
/*   init - Initializes a new semaphore.  Accepts the following positional    */
/*          parameters:                                                       */
/*                           'name' - global name for this semaphore          */
/*                              if named default to set name in               */
/*                              the class semDirectory                        */
/*                           noshare -  do not define named semaphore         */
/*                              in class semDirectory                         */
/*                           Initial state (0 or 1)                           */
/*   setInitialState - Allow for subclass to have some post-initialization,   */
/*     and do setup based on initial state of semaphore                       */
/*   Waiting  - Is the number of objects waiting on this semaphore.           */
/*   Shared   - Is this semaphore shared (Global).                            */
/*   Named    - Is this semaphore named.                                      */
/*   Name     - Is the name of a named semaphore.                             */
/*   setSem   - Sets the semaphore and returns previous state.                */
/*   resetSem - Sets state to unSet.                                          */
/*   querySem - Returns current state of semaphore.                           */
/*                                                                            */
/* SemaphoreMeta - Is the metaclass for the semaphore classes.  This class is */
/*   set up so that when a namedSemaphore is shared, it maintains these       */
/*   named/shared semaphores as part of its state.  These semaphores are      */
/*   maintained in a directory, and an UNKNOWN method is installed on the     */
/*   class to forward unknown messages to the directory.  In this way the     */
/*   class can function as a class and "like" a directory, so [] syntax can   */
/*   be used to retrieve a semaphore from the class.                          */
/*                                                                            */
/*                                                                            */
/* The following are in the subclass EventSemaphore.                          */
/*                                                                            */
/*   Post  - Posts this semaphore.                                            */
/*   Query - Queries the number of posts since the last reset.                */
/*   Reset - Resets the semaphore.                                            */
/*   Wait  - Waits on this semaphore.                                         */
/*                                                                            */
/*                                                                            */
/* The following are in the subclass MutexSemaphore                           */
/*                                                                            */
/*   requestMutex - Gets exclusive use of semaphore.                          */
/*   releaseMutex - Releases to allow someone else to use semaphore.          */
/*        NOTE: Currently anyone can issue a release (need not be the owner). */
/******************************************************************************/
```

Figure 20. Example of a REXX Semaphore Class (Part 1 of 5)

```
/* ========================================================================== */
/* ===          Start of Semaphore class.                              ===== */
/* ========================================================================== */
::class SemaphoreMeta subclass class
::method init
  expose semDict
                                           /* Be sure to initialize parent     */
  .message~new(self, .array~of('INIT', super), 'a', arg(1,'a'))~send
  semDict = .directory~new

::method unknown
  expose semDict
  use arg msgName, args
                                           /* Forward all unknown messages     */
                                           /* to the semaphore dictionary      */
  .message~new(semDict, msgName, 'a', args)~send
  if var('RESULT') then
    return result
  else
    return


::class Semaphore subclass object metaclass SemaphoreMeta

::method init
  expose sem waits shared name
  use arg semname, shr, state

  waits = 0                                /* No one waiting                   */
  name = ''                                /* Assume unnamed                   */
  shared = 0                               /* Assume not shared                */
  sem = 0                                  /* Default to not posted            */

  if state = 1 Then                        /* Should initial state be set?     */
    sem = 1
                                           /* Was a name specified?            */
  if VAR('SEMNAME') & semname \= '' Then Do
    name = semname                         /* Yes, so set the name             */


    if shr \= 'NOSHARE' Then Do            /* Do we want to share this sem?    */
      shared = 1                           /* Yes, mark it shared              */
                                           /* Shared add to semDict            */
      self~class[name] = self
    End

  End
```

Figure 20. Example of a REXX Semaphore Class (Part 2 of 5)

```
self~setInitialState(sem)                    /* Initialize initial state        */

::method setInitialState
                                             /* This method intended to be      */
nop                                          /* overridden by subclasses        */
::method setSem
expose sem
oldState = sem
sem = 1                                      /* Set new state to 1              */
return oldState

::method resetSem
expose sem
sem = 0
return 0

::method querySem
expose sem
return sem


::method shared
expose shared
return shared                                /* Return true 1 or false 0        */

::method named
expose name
                                             /* Does semaphore have a name?     */
if name = '' Then return 0                   /* No, not named                   */
Else return 1                                /* Yes, it is named                */

::method name
expose name
return name                                  /* Return name or ''               */

::method incWaits
expose waits
waits = waits + 1                            /* One more object waiting         */

::method decWaits
expose Waits
waits = waits - 1                            /* One object less waiting         */

::method Waiting
expose Waits
return waits                                 /* Return number of objects waiting */
```

Figure 20. Example of a REXX Semaphore Class (Part 3 of 5)

```
/* ========================================================================== */
/* ===         Start of EventSemaphore class.                             === */
/* ========================================================================== */

::class EventSemaphore subclass Semaphore public
::method setInitialState
  expose posted posts
  use arg posted

  if posted  then posts = 1
  else posts = 0
::method post
  expose posts posted

  self~setSem                     /* Set semaphore state           */
  posted = 1                      /* Mark as posted                */
  reply
  posts = posts + 1               /* Increase the number of posts  */

::method wait
  expose posted

  self~incWaits                   /* Increment number waiting      */
  guard off
  guard on when posted            /* Now wait until posted         */
  reply                           /* Return to caller              */
  self~decWaits                   /* Cleanup, 1 less waiting       */

::method reset
  expose posts posted

  posted = self~resetSem          /* Reset semaphore               */
  reply                           /* Do an early reply             */
  posts = 0                       /* Reset number of posts         */

::method query
  expose posts
                                  /* Return number of times        */
  return posts                    /* Semaphore has been posted     */
```

Figure 20. Example of a REXX Semaphore Class (Part 4 of 5)

```
/* ========================================================================= */
/* ===         Start of MutexSemaphore class.                            === */
/* ========================================================================= */

::class MutexSemaphore subclass Semaphore public

::method setInitialState
  expose owned
  use arg owned


::method requestMutex
  expose Owned

  Do forever                                /* Do until we get the semaphore  */
    owned = self~setSem
    if Owned = 0                            /* Was semaphore already set?     */
      Then leave                            /* Wasn't owned; we now have it   */
    else Do
      self~incWaits
      guard off                             /* Turn off guard status to let   */
                                            /* others come in                 */
      guard on when \Owned                  /* Wait until not owned and get   */
                                            /* guard                          */
      self~decWaits                         /* One less waiting for MUTEX     */
    End
                                            /* Go up and see if we can get it */
  End


::method releaseMutex
  expose owned
  owned = self~resetSem                     /* Reset semaphore                */
```

Figure 20. Example of a REXX Semaphore Class (Part 5 of 5)

Note: There are functions available that use system semaphores. See “SysCreateEventSem” on page 366, and “SysCreateMutexSem” on page 367.

Monitors (Bounded Buffer)

A monitor object consists of a number of client methods, WAIT and SIGNAL methods for client methods to use, and one or more condition variables. Guarded methods provide the functionality of monitors. Do not confuse this with the Monitor class (see “The Monitor Class” on page 177).

```
::method init
/* Initialize the bounded buffer */
expose size in out n
use arg size
in = 1
out = 1
n = 0
```

```
::method append unguarded
/* Add to the bounded buffer if not full */
expose n size b. in
guard on when n < size
use arg b.in
in = in//size+1
n = n+1

::method take
/* Remove from the bounded buffer if not empty */
expose n b. out size
guard on when n > 0
reply b.out
out = out//size+1
n = n-1
```

Readers and Writers

The concurrency problem of the readers and writers requires that writers exclude writers and readers, whereas readers exclude only writers. The UNGUARDED option is required to allow several concurrent readers.

```
::method init
expose readers writers
readers = 0
writers = 0

::method read unguarded
/* Read if no one is writing */
expose writers readers
guard on when writers = 0
readers = readers + 1
guard off

/* Read the data */
say 'Reading (writers:' writers', readers:' readers').'
guard on
readers = readers - 1

::method write unguarded
/* Write if no-one is writing or reading */
expose writers readers
guard on when writers + readers = 0
writers = writers + 1

/* Write the data */
say 'Writing (writers:' writers', readers:' readers').'
writers = writers - 1
```

Chapter 14. Built-in Objects

REXX provides some objects that all programs can use. To access these *built-in objects*, you use the special environment symbols, which start with a period (.).

.METHODS

The .METHODS environment symbol identifies a directory (see "The Directory Class" on page 127) of methods that ::METHOD directives in the currently running program define. The directory indexes are the method names. The directory values are the method objects. See "The Method Class" on page 175.

Only methods that are not preceded by a ::CLASS directive are in the .METHODS directory. If there are no such methods, the .METHODS symbol has the default value of `.METHODS`.

Example:

```
use arg class, methname
class~define(methname,.methods['a'])
::method a
use arg text
say text
```

.RS

.RS is set to the return status from any executed command (including those submitted with the ADDRESS instruction). The .RS environment symbol has a value of `-1` when a command returns a FAILURE condition, a value of `1` when a command returns an ERROR condition, and a value of `0` when a command indicates successful completion. The value of .RS is also available after trapping the ERROR or FAILURE condition.

Note: Commands executed manually during interactive tracing do not change the value of .RS. The initial value of .RS is `.RS`.

Chapter 15. The Security Manager

The security manager provides a special environment that is safe even if agent programs try to perform unexpected actions. The security manager is called if an agent program tries to:

- Call an external function
- Use a host command
- Use the ::REQUIRES directive
- Access the .LOCAL directory
- Access the .ENVIRONMENT directory
- Use a stream name in the input and output built-in functions (CHARIN, CHAROUT, CHARS, LINEIN, LINEOUT, LINES, and STREAM)

Calls to the Security Manager

When the language processor reaches any of the defined security checkpoints, it sends a message to the security manager for the particular checkpoint. The message has a single argument, a directory of information that pertains to the checkpoint. If the security manager chooses to handle the action instead of the language processor, the security manager uses the checkpoint information directory to pass information back to the language processor.

Security manager methods must return a value of either `0` or `1` to the language processor. A value of `0` indicates that the program is authorized to perform the indicated action. In this case, processing continues as usual. A value of `1` indicates that the security manager performed the action itself. The security manager sets entries in the information directory to pass results for the action back to the language processor. The security manager can also use the RAISE instruction to raise a program error for a prohibited access. Error message 98.948 indicates authorization failures.

The defined checkpoints, with their arguments and return values, are:

CALL sent for all external function calls. The information directory contains the following entries:

NAME The name of the invoked function.

ARGUMENTS
An array of the function arguments.

When the CALL method returns `1`, indicating that it handled the external call, the security manager places the function result in the information directory as the entry RESULT.

COMMAND

sent for all host command instructions. The information directory contains the following entries:

COMMAND

The string that represents the host command.

ADDRESS

The name of the target ADDRESS environment for the command.

When the COMMAND method returns 1, indicating that it handled the command, the security manager uses the following information directory entries to return the command results:

RC	The command return code. If the entry is not set, a return code of 0 is used.
FAILURE	If a FAILURE entry is added to the information directory, a REXX FAILURE condition is raised.
ERROR	If an ERROR entry is added to the information directory, a REXX ERROR condition is raised. The ERROR condition is raised only if the FAILURE entry is not set.

REQUIRES

sent whenever a ::REQUIRES directive in the file is processed. The information directory contains the following entry:

NAME	The name of the file specified on the ::REQUIRES directive.

When the REQUIRES method returns 1, indicating that it handled the request, the entry NAME in the information directory is replaced with the name of the actual file to load for the request. The REQUIRES method can also provide a security manager to be used for the program loaded by the ::REQUIRES directive by setting the information direction entry SECURITYMANAGER into the desired security manager object.

LOCAL

sent whenever REXX is going to access an entry in the .LOCAL directory as part of the resolution of the environment symbol name. The information directory contains the following entry:

NAME

The name of the target directory entry.

When the LOCAL method returns 1, indicating that it handled the request, the information directory entry RESULT contains the directory entry. When RESULT is not set and the method returns 1, this is the same as a failure to find an entry in the .LOCAL directory. REXX continues with the next step in the name resolution.

ENVIRONMENT
sent whenever REXX is going to access an entry in the .ENVIRONMENT directory as part of the resolution of the environment symbol name. The information directory contains the following entry:

NAME	The name of the target directory entry.

When the ENVIRONMENT method returns 1, indicating that it handled the request, the information directory entry RESULT contains the directory entry. When RESULT is not set and the method returns 1, this is the same as a failure to find an entry in the .ENVIRONMENT directory. REXX continues with the next step in the name resolution.

STREAM
sent whenever one of the REXX input and output built-in functions (CHARIN, CHAROUT, CHARS, LINEIN, LINEOUT, LINES, or STREAM) needs to resolve a stream name. The information directory contains the following entry:

NAME	The name of the target stream.

When the STREAM method returns 1, the information directory STREAM must be set to an object to be used as the stream target. This should be a stream object or another object that supports the Stream class methods.

METHOD
sent whenever a secure program attempts to send a message for a protected method (see the ::METHOD directive "::METHOD" on page 89) to an object. The information directory contains the following entries:

OBJECT	The object the protected method is issued against.
NAME	The name of the protected method.
ARGUMENTS	An array containing the method arguments.

When the METHOD method returns 1, indicating that it handled the external call, the function result can be placed in the information directory as the method RESULT.

Example

The following agent program includes all the actions for which the security manager defines checkpoints (for example, by calling an external function).

```
/* Agent */
interpret 'echo Hello There'
'dir foo.bar'
call rxfuncadd sysloadfuncs, rexxutil, sysloadfuncs
say result
say syssleep(1)
say linein('c:\profile')
say .array
.object~setmethod('SETMETHOD')
::requires agent2.cmd
```

Figure 21. Agent Program

The following server implements the security manager with three levels of security. For each action the security manager must check (for example, by calling an external routine):

1. The audit manager (Dumper class) writes a record of the event but then permits the action.
2. The closed cell manager (noWay class) does not permit the action to take place and raises an error.
3. The replacement execution environment (Replacer class, a subclass of the noWay class) replaces the prohibited action with a different action.

```
/* Server implements security manager */
parse arg program
method = .method~newfile(program)
say "Calling program" program "with an audit manager:"
pull
method~setSecurityManager(.dumper~new(.output))
.go~new~~run(method)
say "Calling program" program "with a function replacement execution environment:"
pull
method~setSecurityManager(.replacer~new)
.go~new~~run(method)
say "Calling program" program "with a closed cell manager:"
pull
signal on syntax
method~setSecurityManager(.noWay~new)
.go~new~~run(method)
exit

syntax:
  say "Agent program terminated with an authorization failure"
  exit

::class go subclass object

::method run         -- this is a NON-PRIVATE method!
  use arg m
  self~run:super(m)  -- a PRIVATE method is called here!

::class dumper
::method init
  expose stream                        /* target stream for output          */
  use arg stream                       /* hook up the output stream         */
::method unknown                       /* generic unknown method            */
  expose stream                        /* need the global stream            */
  use arg name, args                   /* get the message and arguments     */
                                       /* write out the audit event         */
  stream~lineout(time() date() 'Called for event' name)
  stream~lineout('Arguments are:')     /* write out the arguments           */
  info = args[1]                       /* info directory is the first arg   */
  do name over info                    /* dump the info directory           */
    stream~lineout('Item' name':' info[name])
  end
return 0                               /* allow this to proceed             */

::class noWay
::method unknown                       /* everything trapped by unknown     */
                                       /* and everything is an error        */
  raise syntax 98.948 array("You didn't say the magic word!")
::class replacer subclass noWay        /* inherit restrictive UNKNOWN method*/
::method command                       /* issuing commands                  */
  use arg info                         /* access the directory              */
  info~rc = 1234                       /* set the command return code       */
  info~failure = .true                 /* raise a FAILURE condition         */
  return 1                             /* return "handled" return value     */
```

Figure 22. Example of Server Implementing Security Manager (Part 1 of 2)

```
::method call                          /* external function/routine call    */
  use arg info                         /* access the directory               */
                                       /* all results are the same           */
  info~setentry("RESULT","uh, uh, uh...you didn't say the magic word")
  return 1                              /* return "handled" return value    */
::method stream                        /* I/O function stream lookup         */
  use arg info                         /* access the directory               */
                                       /* replace with a different stream    */
  info~stream = .stream~new('c:\sample.txt')
return 1
                                       /* return "handled" return value      */
::method local                         /* .LOCAL variable lookup             */
                                       /* no value returned at all           */
  return 1                             /* return "handled" return value      */
::method environment                   /* .ENVIRONMENT variable lookup       */
                                       /* no value returned at all           */
  return 1                             /* return "handled" return value      */
::method method                        /* protected method invocation        */
  use arg info                         /* access the directory               */
                                       /* all results are the same           */
  info~setentry("RESULT","uh, uh, uh...you didn't say the magic word")
  return 1                             /* return "handled" return value      */
::method requires                      /* REQUIRES directive                 */
  use arg info                         /* access the directory               */
                                       /* switch to load a different file    */
  info~name = 'c:\samples\agent.cmd'
  info~securitymanager = self          /* load under this authority          */
  return 1                             /* return "handled" return value      */
```

Figure 22. Example of Server Implementing Security Manager (Part 2 of 2)

Chapter 16. Input and Output Streams

REXX defines Stream class methods to handle input and output and maintains the I/O functions for input and output externals. Using a mixture of REXX I/O methods and REXX I/O functions can cause unpredictable results. For example, using the LINEOUT method and the LINEOUT function on the same persistent stream object can cause overlays.

When a REXX I/O function creates a stream object, the language processor maintains the stream object. When a REXX I/O method creates a stream object, it is returned to the program to be maintained. Because of this, when REXX I/O methods and REXX I/O functions referring to the same stream are in the same program, there are two separate stream objects with different read and write pointers. The program needs to synchronize the read and write pointers of both stream objects, or overlays occur.

To obtain a stream object (for example, `MYFIL`), you could use:

```
MyStream = .stream~new('MYFIL')
```

You can manipulate stream objects with character or line methods:

```
nextchar = MyStream~charin()
nextline = MyStream~linein()
```

In addition to stream objects, the language processor defines an external data queue object for interprogram communication. This queue object understands line functions only.

A stream object can have a variety of sources or destinations including files, serial interfaces, displays, or networks. It can be transient or dynamic, for example, data sent or received over a serial interface, or persistent in a static form, for example, a disk file.

Housekeeping for stream objects (opening and closing files, for example) is not explicitly part of the language definition. However, REXX provides methods, such as CHARIN and LINEIN, that are independent of the operating system and include housekeeping. The COMMAND method provides the *stream_command* argument for those situations that require more granular access to operating system interfaces.

The Input and Output Model

The model of input and output for REXX consists of the following logically distinct parts:

- One or more input stream objects
- One or more output stream objects
- One or more external data queue objects

The REXX methods, instructions, and built-in routines manipulate these elements as follows.

Input Streams

Input to REXX programs is in the form of a serial character stream generated by user interaction or has the characteristics of one generated this way. You can add characters to the end of some stream objects asynchronously; other stream objects might be static or synchronous.

The methods and instructions you can use on input stream objects are:

- CHARIN method—reads input stream objects as characters.
- LINEIN method—reads input stream objects as lines.
- PARSE PULL and PULL instructions—read the default input stream object (`.INPUT`), if the external data queue is empty. PULL is the same as PARSE UPPER PULL except that uppercase translation takes place for PULL.
- PARSE LINEIN instruction—reads lines from the default input stream object regardless of the state of the external data queue. Usually, you can use PULL or PARSE PULL to read the default input stream object.

In a persistent stream object, the REXX language processor maintains a current read position. For a persistent stream:

- The CHARS method returns the number of characters currently available in an input stream object from the read position through the end of the stream (including any line-end characters).
- The LINES method determines if any data remains between the current read position and the end of the input stream object.
- You can move the read position to an arbitrary point in the stream object with:
  - The SEEK or POSITION method of the Stream class
  - The COMMAND method's SEEK or POSITION argument
  - The *start* argument of the CHARIN method
  - The *line* argument of the LINEIN method

  When the stream object is opened, this position is the start of the stream.

In a transient stream, no read position is available. For a transient stream:

- The CHARS and LINES methods attempt to determine if data is present in the input stream object. These methods return the value 1 for a device if data is waiting to be read or a determination cannot be made. Otherwise, these methods return `0`.
- The SEEK and POSITION methods of the Stream class and the COMMAND method's `SEEK` and `POSITION` arguments are not applicable to transient streams.

Output Streams

Output stream methods provide for output from a REXX program. Output stream methods are:

- SAY instruction—writes to the default output stream object (`.OUTPUT`).
- CHAROUT method—writes in character form to either the default or a specified output stream object.
- LINEOUT method—writes in lines to either the default or a specified output stream object.

LINEOUT and SAY write the new-line character at the end of each line. Depending on the operating system or hardware, other modifications or formatting can be applied; however, the output data remains a single logical line.

The REXX language processor maintains the current write position in a stream. It is separate from the current read position. Write positioning is usually at the end of the stream (for example, when the stream object is first opened), so that data can be appended to the end of the stream. For persistent stream objects, you can set the write position to the beginning of the stream to overwrite existing data by giving a value of 1 for the CHAROUT *start* argument or the LINEOUT *line* argument. You can also use the CHAROUT *start* argument, the LINEOUT *line* argument, the SEEK or POSITION method, or the COMMAND method's `SEEK` or `POSITION` *stream_command* to direct sequential output to some arbitrary point in the stream.

Note: Once data is in a transient output stream object (for example, a network or serial link), it is no longer accessible to REXX.

External Data Queue

REXX provides queuing services entirely separate from interprocess communications queues.

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 REXX programs can have access to the queue.

The external data queue forms a REXX-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 while a REXX program is running, except when control leaves the program and is manipulated by external means (such as when an external command or routine is called).

There are two kinds of queues in REXX. Both kinds are accessed and processed by name.

Unnamed Queues

One *unnamed queue* is automatically provided for eachREXX program in operation. Its name is always “QUEUE:”, and the language processor creates it when REXX is called and no queue is currently available. All processes that are children of the process that created the queue can access it as long as the process that created it is still running. However, other processes cannot share the same unnamed queue. The queue is deleted when the process that created it ends.

Named Queues

Your program creates (and deletes) *named queues*. You can name the queue yourself or leave the naming to the language processor. Your program must know the name of the queue to use a named queue. To obtain the name of the queue, use the RXQUEUE function:

```
previous_queue=rxqueue("set",newqueuename)
```

This sets the new queue name and returns the name of the previous queue.

The following REXX instructions manipulate the queue:
- PULL or PARSE PULL—reads a string from the head of the queue. If the queue is empty, these instructions take input from .INPUT.
- PUSH—stacks a line on top of the queue (LIFO).
- QUEUE—adds a string to the tail of the queue (FIFO).

REXX functions that manipulate QUEUE: as a device name are:
- LINEIN('QUEUE:')—reads a string from the head of the queue. If the queue is empty, this takes input from .INPUT.
- LINEOUT('QUEUE:','string')—adds a string to the tail of the queue (FIFO).
- QUEUED—returns the number of items remaining in the queue.

Here is an example of using a queue:

```
/*                                                                           */
/* push/pull WITHOUT multiprogramming support                                */
/*                                                                           */
push date() time()                               /* push date and time       */
do 1000                                          /* let's pass some time     */
  nop                                            /* doing nothing            */
end                                              /* end of loop              */
pull a b                                         /* pull them                */
say 'Pushed at ' a b ', Pulled at ' date() time() /* say now and then        */

/*                                                                           */
/*             push/pull WITH multiprogramming support                       */
/*       (no error recovery, or unsupported environment tests)               */
/*                                                                           */
newq = RXQUEUE('Create')                         /* create a unique queue    */
oq = RXQUEUE('Set',newq)                         /* establish new queue      */
push date() time()                               /* push date and time       */
do 1000                                          /* let's spend some time    */
  nop                                            /* doing nothing            */
end                                              /* end of loop              */
pull a b                                         /* get pushed information   */
say 'Pushed at ' a b ', Pulled at ' date() time() /* tell user               */
call RXQUEUE 'Delete',newq                  /* destroy unique queue created  */
call RXQUEUE 'Set',oq              /* reset to default queue (not required)  */
```

Figure 23. Sample REXX Procedure Using a Queue

Special considerations:

- External programs that must communicate with a REXX procedure through defined data queues can use the REXX-provided queue or the queue that QUEUE: references (if the external program runs in a child process), or they can receive the data queue name through some interprocess communication technique, including argument passing, placement on a prearranged logical queue, or the use of usual interprocess communication mechanisms (for example, pipes, shared memory, or IPC queues).
- Named queues are available across the entire system. Therefore, the names of queues must be unique within the system. If a queue named `anyque` exists, using the following function:

  ```
  newqueue = RXQUEUE('Create', 'ANYQUE')
  ```

  results in an error.

Multiprogramming Considerations

The top-level REXX program in a process tree owns an unnamed queue. However, any child process can modify the queue at any time. No specific process or user owns a named queue. The operations that affect the queue are atomic—the subsystem serializes the resource so that no data integrity problems can occur. However, you are responsible for the synchronization of requests so that two processes accessing the same queue get the data in the order it was placed on the queue.

A specific process owns (creates) an unnamed queue. When that process ends, the language processor deletes the queue. Conversely, the named queues created with `RxQueue('Create',` *queuename*`)` exist until you explicitly delete them. The end of a program or procedure that created a named queue does not force the deletion of the private queue. When the process that created a queue ends, any data on the queue remains until the data is read or the queue is deleted. (The function call `RxQueue('Delete',` *queuename*`)` deletes a queue.)

If a data queue is deleted by its creator, a procedure, or a program, the items in the queue are also deleted.

Default Stream Names

A stream name can be a file, a queue, a pipe, or any device that supports character-based input and output. If the stream is a file or device, the name can be any valid file specification.

Windows defines three default streams:
- stdin (file descriptor 0) - standard input
- stdout (file descriptor 1) - standard output
- stderr (file descriptor 2) - standard error (output)

REXX provides .INPUT and .OUTPUT public objects. They default to the default input and output streams of the operating system. The appropriate default stream object is used when the call to a REXX I/O function includes no stream name. The following REXX statements write a line to the default output stream of the operating system:

```
Lineout(,'Hello World')
.Output~lineout('Hello World')
```

REXX reserves the names `STDIN`, `STDOUT`, and `STDERR` to allow REXX functions to refer to these stream objects. The checks for these names are not case-sensitive; for example, `STDIN`, `stdin`, and `sTdIn` all refer to the standard input stream object. If you need to access a file with one of these names, qualify the name with a directory specification, for example, `\stdin`.

REXX also provides access to arbitrary file descriptors that are already open when REXX is called. The stream name used to access the stream object is `HANDLE:`*x*. *x* is the number of the file descriptor you wish to use. You can use `HANDLE:`*x* as any other stream name; it can be the receiver of a Stream class method. If the value of *x* is not a valid file descriptor, the first I/O operation to that object fails.

Notes:

1. Once you close a `HANDLE:`*x* stream object, you cannot reopen it.
2. `HANDLE:x` is reserved. If you wish to access a file or device with this name, include a directory specification before the name. For example, `\HANDLE:x` accesses the file HANDLE:x in the current directory.
3. Programs that use the .INPUT and .OUTPUT public objects are independent of the operating environment.

Line versus Character Positioning

REXX lets you move the read or write position of a persistent stream object to any location within the stream. You can specify this location in terms of characters or lines.

Character positioning is based upon the view of a stream as a simplecollection of bytes of data. No special meaning is given to any single character. Character positioning alone can move the stream pointer. For example:

```
MyStream~charin(10,0)
```

moves the stream pointer so that the tenth character in MyStream is the next character read. But this does not return any data. If MyStream is opened for reading or writing, any output that was previously written but is still buffered is eliminated. Moving the write position always causes any buffered output to be written.

Line positioning views a stream as a collection of lines of data. There are two ways of positioning by lines. If you open a stream in binary mode and specify a record length of *x* on the open, a line break occurs every *x* characters. Line positioning in this case is an extension of character positioning. For example, if you open a stream in binary mode with record length 80, then the following two lines are exactly equivalent.

```
MyStream~command(position 5 read line)
MyStream~command(position 321 read char)
```

Remember that streams and other REXX objects are indexed starting with one rather than zero.

The second way of positioning by lines is for non-binary streams. New-line characters separate lines in non-binary streams. Because the line separator is contained within the stream, ensure accurate line positioning. For example, it is possible to change the line number of the current read position by writing extra new-line characters ahead of the read position or by overwriting existing new-line characters. Thus, line positioning in a non-binary stream object has the following characteristics:

- To do line positioning, it is necessary to read the stream in circumstances such as switching from character methods to line methods or positioning from the end of the stream.
- If you rewrite a stream at a point prior to the read position, the line number of the current read position could become inaccurate.

Note that for both character and line positioning, the index starts with one rather than zero. Thus, character position 1 and line position 1 are equivalent, and both point to the top of the persistent stream object. The REXX I/O processing uses certain optimizations for positioning. These require that no other process is writing to the stream concurrently and no other program uses or manipulates the same low-level drive, directory specification, and file name that the language processor uses to open the file. If you need to work with a stream in these circumstances, use the system I/O functions.

Implementation

Usually, the dialog between a REXX program and you as the user takes place on a line-by-line basis and is, therefore, carried out with the SAY, 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 you generally type. 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 serial interfaces the CHARIN and CHAROUT methods provide. These methods are important for input and output in transient stream objects, such as keyboards, printers, or network environments.

Opening and closing of persistent stream objects, such as files, is largely automatic. Generally the first CHARIN, CHAROUT, CHARS, LINEIN, LINEOUT, or LINES message sent to a stream object opens that stream object. It remains open until you explicitly close it with a CHAROUT or LINEOUT or until the program ends. Using the LINEOUT method with only the name of a stream object (and no output *string* or *line*) closes the named stream object. The Stream class also provides OPEN and CLOSE methods and the COMMAND method, which can explicitly open or close a stream object.

If you open a stream with the CHARIN, CHAROUT, LINEIN, or LINEOUT methods, it is opened for both reading and writing, if possible. You can use the OPEN method or the COMMAND method to open a stream for read-only or write-only operations.

Operating System Specifics

The COMMAND method of the Stream class determines the state of an input or output stream object and carries out specific operations (see "COMMAND" on page 191). It allows REXX programs to open and close selected stream objects for read-only, write-only, or read and write operations, to move the read and write position within a stream object, to control the locking and buffering characteristics, and to obtain information (such as the size and the date of the last update).

Examples of Input and Output

In most circumstances, communication with a user running a REXX program uses the default input and output stream objects. For a question and answer dialog, the recommended technique is to use the SAY and PULL instructions on the .INPUT and .OUTPUT objects. (You can use PARSE PULL if case-sensitive input is needed.)

It is generally necessary to write to, or read from, stream objects other than the default. For example, the following program copies the contents of one stream to another.

```
/* FILECOPY.CMD */
/* This routine copies, as lines, the stream or     */
/* file that the first argument names to the stream */
/* or file the second argument names. It is assumed */
/* that the name is not an object, as it could be   */
/* if it is passed from another REXX program.       */

parse arg inputname, outputname

inputobject = .stream~new(inputname)
outputobject = .stream~new(outputname)

signal on notready

do forever
  outputobject~lineout(inputobject~linein)
end
exit

notready:
return
```

As long as lines remain in the named input stream, a line is read and is then immediately written to the named output stream. This program is easy to change so that it filters the lines before writing them.

The following example illustrates how character and line operations can be mixed in a communications program. It converts a character stream into lines.

```
/* COLLECT.CMD */
/* This routine collects characters from the stream */
/* the first argument names until a line is         */
/* complete, and then places the line on the        */
/* external data queue.                             */
/* The second argument is a single character that   */
/* identifies the end of a line.                    */
parse arg inputname, lineendchar
inputobject = .stream~new(inputname)

buffer=''      /* zero-length character accumulator */
do forever
  nextchar=inputobject~charin
  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 up in a variable called BUFFER. When the line is complete (for example, when the user presses the Enter key) the loop ends and the language processor places the contents of BUFFER on the external data queue. The program then ends.

Errors during Input and Output

The REXX language offers considerable flexibility in handling errors during input or output. This is provided in the form of a NOTREADY condition that the CALL ON and SIGNAL ON instructions can trap. The STATE and DESCRIPTION methods can elicit further information.

When an error occurs during an input or output operation, the function or method called usually continues without interruption (the output method 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:

- If the NOTREADY condition is not trapped, processing continues without interruption. The NOTREADY condition remains in the OFF state.
- If SIGNAL ON NOTREADY traps the NOTREADY condition, the NOTREADY condition is raised. Processing of the current clause stops immediately, and the SIGNAL takes place as usual for condition traps.
- If CALL ON NOTREADY traps 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 processing continues until the end of the current clause. While processing continues, input methods that refer to the same stream can return the null

string and output methods can 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.
- If the NOTREADY condition is in the DELAY state (CALL ON NOTREADY traps the NOTREADY condition, which has already been raised), processing continues, and the NOTREADY condition remains in the DELAY state.

After the NOTREADY condition has been raised and is in DELAY state, the `"O"` option of the CONDITION function returns the stream object being processed when the stream error occurred.

The STATE method of the Stream class returns the stream object state as `ERROR`, `NOTREADY`, or `UNKNOWN`. You can obtain additional information by using the DESCRIPTION method of the Stream class.

Note: `SAY .OUTPUT` and `PULL .INPUT` never raise the NOTREADY condition. However, the STATE and DESCRIPTION methods can return NOTREADY.

Summary of REXX I/O Instructions and Methods

The following lists REXX I/O instructions and methods:
- CHARIN (see "CHARIN" on page 190)
- CHAROUT (see "CHAROUT" on page 190)
- CHARS (see "CHARS" on page 191)
- CLOSE (see "CLOSE" on page 191)
- COMMAND (see "COMMAND" on page 191)
- DESCRIPTION (see "DESCRIPTION" on page 198)
- FLUSH (see "FLUSH" on page 198)
- INIT (see "INIT" on page 198)
- LINEIN (see "LINEIN" on page 198)
- LINEOUT (see "LINEOUT" on page 199)
- LINES (see "LINES" on page 199)
- MAKEARRAY (see "MAKEARRAY" on page 200)
- OPEN (see "OPEN" on page 200)
- PARSE LINEIN (see "PARSE" on page 63)
- PARSE PULL (see "PARSE" on page 63)
- POSITION (see "POSITION" on page 202)
- PULL (see "PULL" on page 69)
- PUSH (see "PUSH" on page 70)
- QUALIFY (see "QUALIFY" on page 202)

- QUERY (see “QUERY” on page 202)
- QUEUE (see “QUEUE” on page 70)
- QUEUED (see “QUEUED” on page 333)
- SAY (see “SAY” on page 75)
- SEEK (see “SEEK” on page 204)
- STATE (see “STATE” on page 205)

Chapter 17. Debugging Aids

In addition to the TRACE instruction described in "TRACE" on page 79, there are the following debugging aids.

Interactive Debugging of Programs

The debug facility permits interactively controlled execution of a program. Adding the prefix character **?** to the TRACE instruction or the TRACE function (for example, `TRACE ?I` or `TRACE(?I)`) turns on interactive debugging and indicates to the user that interactive debugging 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 page 476 for the exceptions). When the language processor pauses, the following debug actions are available:

- Entering a null line causes the language processor to continue with the execution until the next pause for debugging 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.
- Entering an equal sign (=) with no blanks causes the language processor to reexecute the clause last traced. For example, if an IF clause is about to take the wrong branch, you can change the value of the variables on which it depends, and then reexecute it.

  Once the clause has been reexecuted, the language processor pauses again.
- 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 for the INTERPRET instruction (for example, DO-END constructs must be complete). If an instruction contains a syntax error, a standard message is displayed and you are prompted for input again. Similarly, all other SIGNAL conditions are disabled while the string is processed to prevent unintentional transfer of control.

  During interpretation of the string, no tracing takes place, except that nonzero return codes from commands are displayed. The special variable RC and the environment symbol .RS are not set by commands executed from the string. Once the string has been processed, the language processor pauses again for further debugging input.

Interactive debug is turned off in either of the following cases:

- A TRACE instruction uses the **?** prefix while interactive debug is in effect
- At any time, if `TRACE O` or `TRACE` with no options is entered

The numeric form of the TRACE instruction can be used to allow sections of the program to be executed without pause for debugging input. `TRACE n` (that is, a positive result) allows execution to continue, skipping the next `n` pauses (when interactive debugging is or becomes active). `TRACE -n` (that is, a negative result) allows execution to continue without pause and with tracing inhibited for `n` clauses that would otherwise be traced. The trace action a TRACE instruction selects is saved and restored across subroutine calls. This means that if you are stepping through a program (for example, after using `TRACE ?R` to trace results) and then enter a subroutine in which you have no interest, you can enter `TRACE O` to turn off tracing. 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 debugging are turned off until the next entry to the subroutine.

Because any instructions can be executed in interactive debugging you have considerable control over the execution.

The following are some examples:

```
Say expr     /* displays the result of evaluating the        */
             /* expression                                   */

name=expr    /* alters the value of a variable               */

Trace O      /* (or Trace with no options) turns off         */
             /* interactive debugging and all tracing        */

Trace ?A     /* turns off interactive debugging but          */
             /* continues tracing all clauses                */

exit         /* terminates execution of the program          */

do i=1 to 10; say stem.i; end
             /* displays ten elements of the array stem.     */
```

Exceptions: Some clauses cannot safely be reexecuted, 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.
- All THEN, ELSE, OTHERWISE, or null clauses.
- All RETURN and EXIT clauses.
- All SIGNAL clauses (but the language processor pauses after the target label is traced).

- Any clause that causes a syntax error. They can be trapped by SIGNAL ON SYNTAX, but cannot be reexecuted.

A pause occurs after a REPLY instruction, but the REPLY instruction cannot be reexecuted.

RXTRACE Variable

When the interpreter starts the interpretation of a REXX procedure it checks the setting of the special environment variable, *RXTRACE*. If *RXTRACE* has been set to `ON` (not case-sensitive), the interpreter starts in interactive debug mode as if the REXX instruction `TRACE '?R'` had been the first interpretable instruction. All other settings of *RXTRACE* are ignored. *RXTRACE* is only checked when starting a new REXX procedure.

Use the SET command to set or query an environment variable or query all environment variables. To delete an environment variable, use SET *variable*=.

Object REXX Workbench

The Object REXX Workbench is an integrated development environment included in the Object REXX Development Edition that can be used as a comfortable alternative to debug your REXX programs. The Workbench supports, for example, multi-color trace output, in-source debugging, tracepoints, step in, step out, step over, run to cursor, a call stack, and inspect variables.

Chapter 18. Reserved Keywords

Keywords can be used as ordinary symbols in many unambiguous situations. The precise rules are given in this chapter.

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

Within particular instructions, some symbols can be reserved to separate the parts of the instruction. These symbols are referred to as keywords. Examples of REXX keywords are the WHILE keyword in a DO instruction and the THEN keyword, 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. The symbols can be freely used elsewhere in clauses without being understood as keywords.

Be careful with host commands or subcommands with the same name as REXX keywords. To avoid problems, enclose at least the command or subcommand in quotation marks. For example:

```
'DELETE' Fn'.'Ext
```

You can then also use the SIGNAL ON NOVALUE condition to check the integrity of an executable.

Alternatively, you can precede such command strings with two adjacent quotation marks to concatenate the null string to the beginning. For example:

```
'Erase Fn'.'Ext
```

A third option is to enclose the entire expression, or the first symbol, in parentheses. For example:

```
(Erase Fn'.'Ext)
```

Chapter 19. Special Variables

A special variable can be set automatically during processing of a REXX program. There are five special variables:

RC
: is set to the return code from any executed command (including those submitted with the ADDRESS instruction). After the trapping of ERROR or FAILURE conditions, it is also set to the command return code. When the SYNTAX condition is trapped, RC is set to the syntax error number (1–99). RC is unchanged when any other condition is trapped.

  Note: Commands executed manually during interactive tracing do not change the value of RC.

RESULT
: is set by a RETURN instruction in a subroutine that has been called, or a method that was activated by a message instruction, if the RETURN instruction specifies an expression. (See "EXIT" on page 51, "REPLY" on page 73, and "RETURN" on page 74.) If the RETURN instruction has no expression, RESULT is dropped (becomes uninitialized). Note that an EXIT or REPLY instruction also sets RESULT.

SELF
: is set when a method is activated. Its value is the object that forms the execution context for the method (that is, the receiver object of the activating message). You can use SELF to:

  - Run a method in an object in which a method is already running. For example, a Find_Clues method is running in an object called Mystery_Novel. When Find_Clues finds a clue, it sends a Read_Last_Page message to Mystery_Novel:

    ```
    self~Read_Last_Page
    ```
  - Pass references about an object to the methods of other objects. For example, a Sing method is running in object Song. The code `Singer2~Duet(self)` would give the Duet method access to the same Song.

SIGL
: is set to the line number of the last instruction that caused a transfer of control to a label (that is, any SIGNAL, CALL, internal function call, or trapped condition). See "The Special Variable SIGL" on page 436.

SUPER
: is set when a method is activated. Its value is the class object that is the usual starting point for a superclass method lookup for the SELF object. This is the first immediate superclass of

the class that defined the method currently running. (See "Classes and Instances" on page 7.)

The special variable SUPER lets you call a method in the superclass of an object. For example, the following Savings class has INIT methods that the Savings class, Account class, and Object class define.

```
::class Account

::method INIT
expose balance
use arg balance
self~init:super         /* Forwards to the Object INIT method  */

::method TYPE
return "an account"

::method name attribute

::class Savings subclass Account

::method INIT
expose interest_rate
use arg balance, interest_rate
self~init:super(balance) /* Forwards to the Account INIT method */

::method type
return "a savings account"
```

When the INIT method of the Savings class is called, the variable SUPER is set to the Account class object. The instruction:

```
self~init:super(balance) /* Forwards to the Account INIT method */
```

calls the INIT method of the Account class rather than recursively calling the INIT method of the Savings class. When the INIT method of the Account class is called, the variable SUPER is assigned to the Object class.

```
self~init:super         /* Forwards to the Object INIT method  */
```

calls the INIT method that the Object class defines.

You can alter these variables like any other variable, but the language processor continues to set RC, RESULT, and SIGL automatically when appropriate. The EXPOSE, PROCEDURE, USE and DROP instructions also affect these variables.

REXX also supplies functions that indirectly affect the execution of a program. An example is the name that the program was called by and the source of the program (which are available using the PARSE SOURCE instruction). In addition, PARSE VERSION makes available the language version and date of REXX implementation that is running. The built-in functions ADDRESS, DIGITS, FUZZ, FORM, and TRACE return other settings that affect the execution of a program.

Chapter 20. Useful Services

The following section describes useful commands and services.

Windows Commands

COPY	copies files.
DELETE	deletes files.
DIR	displays disk directories.
ERASE	erases files.
MODE	controls input and output device characteristics.
PATH	defines or displays the search path for commands and REXX programs. See also "Search Order" on page 299.
SET	displays or changes Windows environment variables. See also "VALUE" on page 350.

Subcommand Handler Services

For a complete subcommand handler description, see the *Object REXX for Windows Programming Guide*.

The RXSUBCOM Command

The RXSUBCOM command registers, drops, and queries REXX subcommand handlers. A REXX procedure or script file can use RXSUBCOM to register dynamic-link library subcommand handlers. Once the subcommand handler is registered, a REXX program can send commands to the subcommand handler with the REXX ADDRESS instruction. For example, REXX Dialog Manager programs use RXSUBCOM to register the ISPCIR subcommand handler.

```
'RXSUBCOM REGISTER ISPCIR ISPCIR ISPCIR'
Address ispcir
```

See "ADDRESS" on page 42 for details of the ADDRESS instruction.

RXSUBCOM REGISTER

RXSUBCOM REGISTER registers a dynamic-link library subcommand handler. This command makes a command environment available to REXX.

```
►►──RXSUBCOM──REGISTER──envname──dllname──procname──────────────────────────►◄
```

Parameters:

envname
The subcommand handler name. The REXX ADDRESS instruction uses *envname* to send commands to the subcommand handler.

dllname
The name of the dynamic-link library file containing the subcommand handler routine.

procname
The name of the dynamic-link library procedure within *dllname* that REXX calls as a subcommand handler.

Return codes:

0	The command environment has been registered.
10	A duplicate registration has occurred. An *envname* subcommand handler in a different dynamic-link library has already been registered. Both the new subcommand handler and the existing subcommand handler can be used.
30	The registration has failed. Subcommand handler *envname* in library *dllname* is already registered.
1002	RXSUBCOM was unable to obtain the memory necessary to register the subcommand handler.
-1	A parameter is missing or incorrectly specified.

RXSUBCOM DROP

RXSUBCOM DROP deregisters a subcommand handler.

```
►►──RXSUBCOM──DROP──envname──┬─────────┬─────────────────────────────────►◄
                             └─dllname─┘
```

Parameters:

envname
The name of the subcommand handler.

dllname
The name of the dynamic-link file containing the subcommand handler routine.

Return codes:

0	The subcommand handler was successfully deregistered.
30	The subcommand handler does not exist.
40	The environment was registered by a different process as RXSUBCOM_NONDROP.
-1	A parameter is missing or specified incorrectly.

RXSUBCOM QUERY

RXSUBCOM QUERY checks the existence of a subcommand handler. The query result is returned.

```
►►──RXSUBCOM──QUERY──envname──┬─────────┬──────────────────────────────►◄
                              └─dllname─┘
```

Parameters:

envname
: The name of the subcommand handler.

dllname
: The name of the dynamic-link file containing the subcommand handler routine.

Return codes:

0	The subcommand handler is registered.
30	The subcommand handler is not registered.
-1	A parameter is missing or specified incorrectly.

RXSUBCOM LOAD

RXSUBCOM LOAD loads a subcommand handler dynamic-link library.

```
►►──RXSUBCOM──LOAD──envname──┬─────────┬───────────────────────────────►◄
                             └─dllname─┘
```

Parameters:

envname
: The name of the subcommand handler.

libname
: The name of the dynamic-link file containing the subcommand handler routine.

Return codes:

0	The dynamic-link library was located and loaded successfully.
50	The dynamic-link library was not located or could not be loaded.
-1	A parameter is missing or incorrectly specified.

The RXQUEUE Filter

```
►►──RXQUEUE──┬───────────┬──┬─────────┬──────────────────────────────────────►◄
             └─queuename─┘  ├─/FIFO───┤
                            ├─/LIFO───┤
                            └─/CLEAR──┘
```

The RXQUEUE filter usually operates on the default queue named SESSION. However, if an environment variable named RXQUEUE exists, the RXQUEUE value is used for the queue name.

For a full description of REXX queue services for applications programming, see "External Data Queue" on page 465.

Parameters:

queuename/**LIFO**
: stacks items from STDIN last in, first out (LIFO) on a REXX queue.

queuename/**FIFO**
: queues items from STDIN first in, first out (FIFO) on a REXX queue.

queuename/**CLEAR**
: removes all lines from a REXX queue.

RXQUEUE takes output lines from another program and places them on a REXX queue. A REXX procedure can use RXQUEUE to capture Windows command and program output for processing. RXQUEUE can direct output to any REXX queue, either FIFO (first in, first out) or LIFO (last in, first out).

RXQUEUE uses the environment variable RXQUEUE for the default queue name. When RXQUEUE does not have a value, RXQUEUE uses SESSION for the queue name.

The following example obtains the Windows version number with RXQUEUE:

```
/* Sample program to show simple use of RXQUEUE */
/* Find out the Windows version number, using the  */
/* VER command.  VER produces two lines of      */
/* output; one blank line, and one line with the*/
/* format "The Windows Version is n.nn"           */

'VER |RXQUEUE'          /* Put the data on the Queue      */
pull .                  /* Get and discard the blank line */
Pull . "VERSION" number "]" /* The bracket is required for
Windows 95, not for Windows NT */
Say 'We are running on Windows Version' number
```

Note that the syntax of the version string that is returned by Windows can vary, so the parsing syntax for retrieving the version number may be different.

The following example processes output from the DIR command:

```
/* Sample program to show how to use the RXQUEUE filter */
/* This program filters the output from a DIR command,  */
/* ignoring small files.  It displays a list of the     */
/* large files, and the total of the sizes of the large */
/* files.                                               */

size_limit = 10000            /* The dividing line      */
/* between large and small*/
size_total = 0                /* Sum of large file sizes*/
NUMERIC DIGITS 12             /* Set up to handle very  */
/* large numbers          */

/* Create a new queue so that this program cannot       */
/* interfere with data placed on the queue by another   */
/* program.                                             */

queue_name = rxqueue('Create')
Call rxqueue 'Set', queue_name

'DIR /N | RXQUEUE' queue_name

/* DIR output starts with five header lines             */
Do 5
Pull .                     /* discard header line    */
End

/* Now all the lines are file or directory lines,       */
/* except for one at the end.                           */

Do queued() - 1               /* loop for lines we want */
Parse Pull . . size . name ./* get one name and size */
/* If the size field says "<DIR>", we ignore this    */
/* line.                                             */
If size <> '<DIR>' Then
/* Now check size, and display                    */
If size > size_limit Then Do
Say format(size,12) name
size_total = size_total + size
End
End

Say 'The total size of those files is' size_total

/* Now we are done with the queue.  We delete it, which */
/* discards the line remaining in it.                   */

Call rxqueue 'DELETE', queue_name
```

Distributing Programs without Source

The Object REXX for Windows Development Edition comes with a utility called REXXC. You can use this utility to produce versions of your programs that do not include the original program source. You can use these programs to replace any REXX program file that includes the source, with the following restrictions:

1. The SOURCELINE built-in function returns `0` for the number of lines in the program and raises an error for all attempts to retrieve a line.
2. A sourceless program may not be traced. The TRACE instruction runs without error, but no tracing of instruction lines, expression results, or intermediate expression values occurs.

The syntax of the REXXC utility is:

```
>>--REXXC--inputfile--+------------+--+----+----------------------------><
                      '-outputfile-'  '-/s-'
```

If you specify the *outputfile*, the language processor processes the *inputfile* and writes the executable version of the program to the *outputfile*. If the *outputfile* already exists, it is replaced.

If the language processor detects a syntax error while processing the program, it reports the error and stops processing without creating a new output file. If you omit the *outputfile*, the language processor performs a syntax check on the program without writing the executable version to a file.

You can use the */s* option to suppress the display of the information about the interpreter used.

An alternative to REXXC is provided by the "Tools — Tokenize Script" command of the Object REXX Workbench.

Note: You can use the in-storage capabilities of the RexxStart programming interface to process the file image of the output file.

With version 2.1, the tokenized form has changed. All Object REXX for Windows editions contain a utility called REXXRT that can be used to change old tokenized forms to the new one. The recommended procedure is to create a new tokenized file from the original source with the new version of Object REXX. However, if the source code is no longer available, REXXRT can be used to convert the old tokenized file. The syntax of the REXXRT utility is:

```
►►──REXXRT──inputfile──outputfile────────────────────────────────────────►◄
```

Chapter 21. Windows Scripting Host Engine

This chapter describes the use of Object REXX as a Windows Scripting Host (WSH) engine.

Object REXX as a Windows Scripting Host Engine

Object REXX is automatically enabled as an engine for Windows Scripting Host at installation. This chapter gives a brief description of WSH and how Object REXX interacts with it, and shows you how you can best use this feature.

The easiest part of this feature to understand and to become immediately productive with is its ability to use Object REXX as a scripting language for Microsoft's Web browser, Internet Explorer. To go quickly to using this technique, see "Invocation by the Browser" on page 495.

Windows Scripting Host Overview

Windows Scripting Host (WSH) is a unified scripting environment for all Microsoft products. It is usable by any macro language that follows its specification. WSH is the mechanism that allows users to customize and dynamically control the products that support its hosting standard.

The Windows Scripting Host engine for Object REXX enables users to drive Microsoft's products, notably Internet Explorer. Other products that can be driven include the components of the Office suite, like Word, Excel, and so on.

The difference between WSH and the OLE support that Object REXX provides is the context in which the script resides. OLE scripts are exterior to the product, and WSH scripts can be embedded in the files that the product uses. The advantage of embedding the script is that the user has fewer files to manage. The Object REXX engine for WSH enables users to accomplish this in a seamless fashion.

There are two components to WSH. The first is the *host* – the product that can be scripted. The second is the *engine* – the product that interprets the script.

Object REXX supplies the engine component of WSH.

The Gestation of WSH

As with many new technologies today, WSH introduces several new concepts and terms. The best way to describe these is to start with an overview of the problem that WSH addresses, and its history.

Object REXX as a Windows Scripting Host Engine

Until recently, Microsoft provided users simply with a COM (Common Object Model) interface to their office products. COM is a binary, as opposed to text, command input system. These commands drove the product – by, for example, telling Microsoft Word to print the current document – and did not contain any logic or decision-making capabilities. Users who wanted such capabilities developed them in programs external to the COM object. Accessing the interface required the user to develop the logic to drive the COM object at first in C++, then later in Visual Basic. The investment for the user, in development time, was quite significant.

In order to satisfy customer demand, a particular version of a scripting language (based on Visual Basic) was developed for each Microsoft product. In addition, the emergence of scripting languages such as JavaScript, with their ability to dynamically control Web browsers, led Microsoft to develop two more scripting languages, VBScript and JScript.

WSH is a consolidation of the scripting language proliferation. Borrowing heavily from the browser paradigm, the host interprets a language-independent XML file that contains one or more scripts where each script is encapsulated in a script (script tag) that denotes the language of the script, and any other necessary environmental parameters. The host extracts the script from the file, and passes it to the appropriate interpreter.

Hosts Provided by Microsoft

Microsoft provides three fully-implemented scripting hosts. They are Microsoft Internet Explorer, CScript, and WScript. As an expansion on the concept of using a scripting language to drive external products, CScript and WScript were developed to control the Windows operating system. The two modules are so similar that they are sometimes referred to as C/WScript. CScript is intended to be used from the command line, and WScript is best used in the Windows environment. Both provide their services to the script through the WScript object. Using the default method for output `WScript~Echo()`, CScript sends the output to a console screen in the same manner as the Object REXX command `Say`, whereas `WScript~Echo()` in a script controlled by WScript will create a pop-up box in which the user must click the OK button to make it disappear.

Scripting in the Windows Style

Each flavor of WSH has an associated file type. This section gives a brief example of scripting for each file type, and suggestions that are appropriate in each case. If you need to, see the appropriate documentation for the exact syntax of WSH’s XML format, and the syntax of an HTML file.

Invocation by the Browser

Invocation by the Web browser is probably the easiest scripting technique to illustrate, and the most familiar use of WSH. The following is a small HTML file that shows Object REXX as the scripting language. There are three paragraphs that have the animating power of Object REXX behind them. Each uses an Internet Explorer pop-up window to denote a particular mouse action. The appropriate activity takes place when the mouse is rolled over the first paragraph, when it leaves the second, and when it is used to click the third.

```
<HTML>
<!--
/*******************************************************/
/* DISCLAIMER OF WARRANTIES. The following [enclosed]   */
/* code is sample code created by IBM Corporation. This */
/* sample code is not part of any standard or IBM       */
/* product and is provided to you solely for the        */
/* purpose of assisting you in the development of your  */
/* applications.  The code is provided "AS IS", without */
/* warranty of any kind.  IBM shall not be liable for   */
/* any damages arising out of your use of the sample    */
/* code, even if they have been advised of the          */
/* possibility of such damages.                         */
/*******************************************************/
 !-->

<HEAD>
  <TITLE>A simple event</TITLE>
  <script language="Object REXX" >
::Routine Display Public
   Window~Alert(Arg(1))
   Return   "something to keep the mouseover function call happy"
   </script>
</HEAD>

<BODY BGCOLOR="#ffffff">
  <FONT FACE="Arial, Helvetica" COLOR="#f00000">
  <H1>How to use events</H1>

  <FONT COLOR="#000000">
  <P>Moving the cursor over the following paragraphs will cause two
  events, respectively: one when you move onto the text, and one when
  you leave it. At both times a pop-up message will inform you about this.</P>

  <!-- in both cases the "alert" function of the object "window" is called !-->
  <FONT COLOR="#0000ff">
  <P onmouseout="alert('Cursor left paragraph')" LANGUAGE="Object REXX">
  Event takes place when cursor leaves this paragraph.</P>

  <P onmouseover="a = Display('Cursor is over paragraph')"
LANGUAGE="Object REXX">
Event takes place when cursor moves over this paragraph.</P>

  <FONT COLOR="#000000">
  <P>The following paragraph reacts when you click it:</P>
  <FONT COLOR="#0000ff">
  <P onclick="call Display 'Thank you! The current time is' time()',' date()"
  LANGUAGE="Object REXX">Click me!</P>
  </FONT>

</BODY>

</HTML>
```

The important things to note in this example are:

- The `LANGUAGE="Object REXX"` attribute on each tag that contains code.
- The `<script>` tag in the `<HEAD>` section defines a function that can be called from any other code section in this HTML file.
  - The Object REXX keyword `PUBLIC` must be on the `::ROUTINE` statement, or Object REXX will not be able to make that name accessible outside of that script block.
  - The `Window` object is accessible, even though it was not declared and the `::ROUTINE` statements have the variable scope of an external routine.
  - Some text was put on the `RETURN` statement simply as a precaution. Those familiar with Object REXX know that routines called as functions demand a return value.
- All of the code for the `mouseout=` is completely contained within the `<p>` and `</p>` tags.
  - Also note the lack of the leading `"Window~"` on the Alert(). See "Changes in Object REXX due to WSH" on page 509.
- The second event references the routine that was defined earlier as a function. The return value is assigned to the variable "a", and discarded as soon as the event finishes processing. Unlike the situation in JScript, function return values in WSH must be used in an expression, or assigned to a variable.
- The third event also references the routine that was defined earlier, but this time as a procedure and not as a function. The `CALL` statement forces this kind of access.
  - `CALL` statements do not produce an error if no value is returned. If a value is returned, and `CALL` was used to activate the routine, the value can be obtained from the special variable `RESULT`.

Additional examples can be found in the Samples\WSH subdirectory of your Object REXX for Windows installation directory.

WSH File Types and Formats

Two main file types are used by WSH. Both follow an XML format that wraps the script code. The XML tags are interpreted by C/WScript, and direct it to the correct scripting engine to process the code inside. The file type `.wsf` is used to define scripts that are executed as commands. This is similar to the conventional way of invoking Object REXX in the Windows environment. The file type .wsc is used to define scripts that are treated as COM objects. The XML tags here denote the properties, methods, and events of the COM object, as well as the correct engine to invoke for scripts.

Note that these XML files are well formed, but not valid. There is no associated Document Type Definition (DTD).

.wsf

The .wsf file type is as easy to invoke as HTML, and is very similar in appearance, with only minor differences. The .wsf file is used to drive the operating system in the same way that an HTML file is used to drive the browser. The file is an Object REXX script file with an XML wrapper.

The following sample prints the version of the JScript engine and the version of the scripting host. If this file had the name "SimpleOREXX.wsf", the command to invoke it would be "CScript //nologo SimpleOREXX.wsf", or "WScript //nologo SimpleOREXX.wsf".

```
<?xml version="1.0"?>
<?job error="true" debug="true" ?>

<package id="wstest">

<!--
/*********************************************************/
/* DISCLAIMER OF WARRANTIES. The following [enclosed]   */
/* code is sample code created by IBM Corporation. This */
/* sample code is not part of any standard or IBM       */
/* product and is provided to you solely for the        */
/* purpose of assisting you in the development of your  */
/* applications.  The code is provided "AS IS", without */
/* warranty of any kind.  IBM shall not be liable for   */
/* any damages arising out of your use of the sample    */
/* code, even if they have been advised of the          */
/* possibility of such damages.                         */
/*********************************************************/
 !-->

<!--  Just a small file to demonstrate the *.wsf file format, and
    ---  what Windows provides by default.
    -->
<job idid="RunByDefault">

<!---
  ---   These functions are provided by WSH.
  -->
 <script language="JScript"><![CDATA[


function GetScriptEngineInfo(){
   var s;
```

```
      s = ""; // Build string with necessary info.
      s += ScriptEngine() + " Version ";  //  Except this function.  It can
                                          // only be accessed from JScript
                                          // or VBscript.
      s += ScriptEngineMajorVersion() + ".";
      s += ScriptEngineMinorVersion() + ".";
      s += ScriptEngineBuildVersion();
      return(s);
      }


]]></script>
<!---
  ---  Not all of the script needs to be within one tag, or use the
  ---  same language.
  -->
 <script language="Object REXX"><![CDATA[

Say "This is "GetScriptEngineInfo()
Ver = "Accessing the version info from Object REXX yields"
Ver = Ver ScriptEngineMajorVersion()"."
Ver = Ver||ScriptEngineMinorVersion()"."ScriptEngineBuildVersion()
Say Ver

WScript~Echo("Done!")

]]></script>
</job>

</package>
```

The important things to note in this example are:

- Accept the two XML tags (`<? ... ?>`) at the beginning as boilerplate, although the `debug="true"` can also be `'debug="false"` without any detrimental effect.
- All XML tag names and attributes are in lower case.
- All XML tags have a beginning and an end tag. The beginning tag looks like `<tag>`, and the end tag `</tag>`. Where the tag contains only attributes, and there is no content between the beginning and the end tag, it is acceptable to abbreviate `<tag attribute=""></tag>` to `<tag attribute=""/>`.
- Comments are the same as in HTML.
- Following the `<script>` tag is the tag `<![CDATA[`, and preceding the `<script/>` tag is]]>. This tells the XML parser to ignore this text. If this is not done, many of the operators and special characters in the script will confuse the XML parser, and it will abort the script.
- There are several `<script>` tags; here Object REXX is invoking a JScript function.

- The functions that begin with `ScriptEngine...` and the WScript object are not declared, yet Object REXX finds them. They are implicit, and their scope is global.

Additional examples can be found in the Samples\WSH subdirectory of your Object REXX for Windows installation directory.

.wsc

The `.wsc` file type is much more elaborate than the `.wsf` type. Since a `.wsc` file is used as a COM object, the XML must describe the object in a way that is independent of the script language. Consider the following example.

```
<?xml version="1.0"?>
<?component error="true" debug="true" ?>

<package id="SimpleObjectREXXCOMScriptTest">

<!--
/*******************************************************/
/* DISCLAIMER OF WARRANTIES. The following [enclosed]   */
/* code is sample code created by IBM Corporation. This */
/* sample code is not part of any standard or IBM       */
/* product and is provided to you solely for the        */
/* purpose of assisting you in the development of your  */
/* applications.  The code is provided "AS IS", without */
/* warranty of any kind.  IBM shall not be liable for   */
/* any damages arising out of your use of the sample    */
/* code, even if they have been advised of the          */
/* possibility of such damages.                         */
/*******************************************************/
!-->


<!---
  ---  An example script to demonstrate the features provided by the
  ---  COM structure.  Register our own typelib, create methods,
  ---  and create a property.
  !-->


<!---

  ---    This section registers the script as a COM
  ---  object when Register is chosen from the list of commands
  ---  that appear when this file is right-clicked.
  ---
  ---    The value of progid= is how the world will find us.
  ---  Two GUID's are needed, one for the COM object, and one
  ---  for the Typelib that will be generated. The routine's
  ---  Register and Unregister mimic those required in a COM
  ---  *.dll.  Even within these routines, there is full
  ---  Object REXX capability.
  !-->
<component id="SimpleOREXXCOM">
```

```
  <registration
     progid="SimpleObjectREXX.Com"
     description="Test of the COM scriptlet interface as seen by Object
     REXX."
     version="1.0"
     clsid="{6550bac9-b31d-11d4-9306-b9d506515f14}">
   <script language="Object REXX"><![CDATA[
::Routine Register Public
  Shell = .OLEObject~New("WScript.Shell")
  Typelib = .OLEObject~New("Scriptlet.TypeLib")
  Shell~Popup("We are registering, n o w . . . .")
     /*
      *  Please note that the name that follows must match
      * our file name exactly, or this fails when registering
      * with an "OLE exception", Code 800C0005 or Code 800C0009.
      */
     Typelib~AddURL("SimpleOREXXCOM.wsc")
     Typelib~Path= "SimpleOREXXCOM.tlb"
     Typelib~Doc = "Test component typelib for Simple Object REXX.Com"
     Typelib~Name = "Test component typelib for Simple Object REXX.Com"
     Typelib~MajorVersion = 1
     Typelib~MinorVersion = 0
     Typelib~GUID = "{6550bac5-b31d-11d4-9306-b9d506515f14}"
     Typelib~Write()
     Typelib~Reset()
     Shell~Popup("We've really done it n o w . . . .")

   ::Routine Unregister Public
     Shell = .OLEObject~New("WScript.Shell")
     Shell~Popup("We are outa here!")

     ]]></script>
    </registration>

<!---
  ---    This section is what describes this COM object to the outside
  ---  world. There is one property, and there are two methods named.
  ---  One of the methods is the default, since its dispid is 0.
  ---  Object REXX does not support calling the default in a shorthand
  ---  manner. All calls are as follows:
  ---
  ---  Obj = .OLEObject~New("SimpleObjectREXX.Com")
  ---  Obj~DefaultMethod("Some Parm")
  ---
!-->
<public>
  <property name="ExternalPropertyName"
internalName="InternalPropertyName" dispid="3">
     </property>
  <method name="NamedRoutine">
     <parameter name="NamedParameter"/>
     </method>
  <method name="DefaultMethod" dispid="0">
     <parameter name="ReallyForTheOutsideWorld" />
     </method>
  </public>


<!---
  ---    This is the actual script code. Note that the property
  ---  is declared at the highest scope. If this is not done,
  ---  then the property will not be found, and the script
  ---  will not abend when the property is referenced.
!-->
  <script language="Object REXX" ><![CDATA[
InternalPropertyName = "Sample Property"

::Routine NamedRoutine Public
say "There are "Arg()" args."
a = RxMessageBox("Is executing, now.","NamedRoutine","OK",)
  Return

::Routine DefaultMethod Public
say "There are "Arg()" args."
a = RxMessageBox("Is executing, now.","DefaultMethod","OK",)
WShell = .OLEObject~New("WScript.Shell")
a = WShell~Popup("A message via an implicit COM object.");
  Return  "a value"

  ]]></script>

</component>

</package>
```

The important things to note are:

- There are three distinct sections in this file, and two of them contain Object REXX code.
  - The first section identifies this as a COM object. The `progid=`, `version=`, and `clsid=` attributes of the `<registration>` tag are given so that this file can be entered into the Windows Registry as a COM object. This is one of the sections that has code. The code here generates the Typelib when the script is registered as a COM object.
  - The second section lists all of the entry points to this object, their parameters, and any data that is being externalized. When the Typelib is generated, this information is used to create its contents. This is more of a designer's wish list than something that is enforced. The designer states what he or she believes to be the minimal number of parameters. The designer must then enforce this within the subroutine. However, be aware that other routines calling these listed here may pass more, or fewer, parameters than this section suggests. This is especially true for procedures named with `<method>` tags. WSH passes the named parameter `THIS`, which Object REXX passes on to the routine.
  - The third section is the actual code.

- Read the comments before each section; they contain important information about that particular section.
- Any code that is put in the same scope as the property being assigned its value is called *immediate code*. Immediate code is executed when the COM object is loaded, before any of its pieces (methods, properties, or events) are accessed. It executes even if none of the external pieces are accessed.

Additional examples can be found in the Samples\WSH subdirectory of your Object REXX for Windows installation directory.

Invocation from a Command Prompt

Invocation from a command prompt covers many possible means:

Opening a DOS window to type the command into;

Selecting Start->Run from the Windows taskbar;

Starting from a file association made in Windows Explorer.

A conventional Object REXX file is one in which every line is valid Object REXX syntax, and makes no assumptions about global objects. It contains no XML wrapper as described in the section on `.wsf` files.

Consider what happens when a file named `WSH.rex` contains the single line: `'WScript~Echo("WSH is available.")'`; another file named `WSH.wsf` contains the same line of code in the .wsf wrapper described above; and another file, `Safe.rex`, contains the line `"Say 'Conventional REXX file' Arg(1)"`.

As a Conventional Object REXX File

From a command prompt, `"REXX WSH.rex"`, will stop with an error 97: `Object "WScript" does not understand message "Echo"`.

From a command prompt, `"REXX WSH.wsf"`, will stop with an error 35: `Invalid expression detected at "<"`.

From a command prompt, `"REXX Safe.rex GREAT!"`, produces one line of output, `"Conventional REXX File GREAT!"`.

As a Windows Scripting Host File

Both CScript and WScript will invoke a file from the command line. All of their parameters begin with a double slash. Two useful parameters are: `//nologo` and `//e:`. The `//nologo` parameter prevents the banner from being displayed, and `//e:` tells WSH not to interpret this file, and to pass the complete contents to the named engine. Enter `CScript` or `WScript` with no parameters or file names to see a complete list of parameters.

WScript converts all WScript~Echo() output into pop-up text boxes, whereas with CScript they are displayed as output lines in a DOS window. If CScript is executed from outside a DOS window (either from Start->Run, or from the use of Windows Explorer), a DOS window will be created for the output. Note, however, that it is removed when the script is complete. Usually, this means that the lifetime of the DOS window is long enough for a person to detect it, but not to actually read it.

From a command prompt, "`cscript //e:"Object REXX" WSH.rex`" produces one line of output, "`WSH is available.`" From a command prompt, "`wscript //e:"Object REXX" WSH.rex`", produces a pop-up box that contains the title "`Windows Script Host`", an **OK** button, and the text "`WSH is available.`"

From a command prompt, "`cscript //e:"Object REXX" WSH.wsf`" will stop with an error 35: `Invalid expression detected at "<"`. From a command prompt, "`wscript //e:"Object REXX" WSH.wsf`", will seem as if it produced no output at all. Though Object REXX is still generating the error message, WScript does not detect the output to STDOUT, and no DOS window is created.

From a command prompt, "`cscript //e:"Object REXX" Safe.rex GREAT!`" produces one line of output, "`Conventional REXX File`". Note the lack of the word GREAT!. WSH does not pass the command line `args` to Object REXX. The `WScript~Arguments` method/object must be used, as in the following code:

```
/*    Note that the WScript object is not declared. It just appears
 * courtesy of CScript and WScript
 */
Say  "The arguments as WSH sees them."
If WScript~Arguments~length > 0 Then Do I = 0 To (WScript~Arguments~length - 1)
  Say i WScript~Arguments(i)
  End
Else Say "No arguments were sent."
```

From a command prompt, "`wscript //e:"Object REXX" Safe.rex GREAT!`", will seem as if it produced no output at all. As when `WSH.wsf` is run by WScript with a known engine (see the relevant paragraph earlier), Object REXX is still executing the `SAY` instruction, WScript does not detect the output to STDOUT, and no DOS window is created.

Invocation as a COM Object

This is the most intricate of the script files to execute. Multiple steps are involved, and there is no command that directly invokes the script. C/WScript cannot be used to directly invoke a `.wsc` file. It must be processed by other means first. Once created, the file must be registered.

Once registered, this can be invoked by any program that can call COM objects. It does not have to be another script; that program could be Visual Basic or C++. If the COM object is to be invoked by Visual Basic, it is a good idea to generate a Typelib. This helps Visual Basic to form its parameter list.

Registering the COM Object

Use either of two methods to register a `.wsc` file. The first is to right-click it in Windows Explorer, and choose `Register` from the list of commands that appears. The second is from the command line. For example, to register `WSH.wsc`, at a command prompt, enter the command, `"regsvr32 /c WSH.wsf"`.

The GUID in the `clsid=` attribute must be unique for the machine the COM object is being registered on. In other words, no other COM object may use the GUID. Once it is registered, the script cannot be moved. The path to a COM object is stored in the Registry as a complete path. If the script is moved, then Windows will not know how to find it.

Generating a Typelib

Use either of two methods to generate the Typelib. One is using code in the Register method of the `<registration>` section. See the sample `.wsc` code above for an example of this. The other is to choose `Generate Type Library` from the list of commands that appear when the file name is right-clicked in Windows Explorer.

Invoking

The easiest method of invoking the script, once it is a COM object, is to use an OLE-enabled application, such as Object REXX. The following Object REXX code shows how to define the object in Object REXX, and invoke its methods.

```
<?xml version="1.0"?>
<?job error="true" debug="true" ?>

<package id="wstest">

<!--
/*******************************************************/
/* DISCLAIMER OF WARRANTIES.  The following [enclosed]  */
/* code is sample code created by IBM Corporation. This */
/* sample code is not part of any standard or IBM       */
/* product and is provided to you solely for the        */
/* purpose of assisting you in the development of your  */
/* applications.  The code is provided "AS IS", without */
/* warranty of any kind.  IBM shall not be liable for   */
/* any damages arising out of your use of the sample    */
/* code, even if they have been advised of the          */
/* possibility of such damages.                         */
/*******************************************************/
 !-->

<!---
  ---     This example shows how easy it is to
  ---  invoke a COM object that is a script by means of
  ---  Object REXX.
  -->
<job id="RunByDefault">
  <script language="Object REXX"><![CDATA[
Say "Creating the ObjectREXX.Com object. "
Sample = .OLEObject~new("SimpleObjectREXX.Com")
Say "Just before the default method "
ReturnValue = Sample~DefaultMethod("A parm");
ReturnValue = Sample~NamedRoutine("A parm");
  ]]></script>
  </job>
</package>
```

Object REXX is not the only way to invoke the script. Any application that can call COM objects can invoke it. For further information, see the relevant documentation.

Events

When scripts are turned into COM objects they can initiate events. Several types of events are supported: the default COM events, HTML or Behavior events, and ASP events. The type of event that the COM object supports is denoted by the `type=` attribute of the `<implements>` tag. An in-depth discussion of events and how to create, code, and handle them is beyond the scope of this documentation. However, there are a few concepts that should be mentioned.

COM Events: In the `<public>` section, where the external attributes of the COM object are disclosed, `<event>` tags can be added. They name the events that the script could possibly activate. When the script that calls the COM object instantiates it by using the method provided by WScript, rather than the Object REXX method, it can inform the COM object that it will handle the events that the COM object fires. Note that when a script agrees to handle the events of an object, it must handle *all* of the events of that object.

For example, suppose the public section looked as follows:

```
<public>
  <event name="Event1" />
  <event name="Event2" />
</public>
```

and the script that instantiated the COM objects code looked as follows:

```
REXXObject = WScript~CreateObject("ObjectREXX.Com","Event_");
```

In that case, the instantiating script would be required to define the two routines below.

```
::Routine Event_Event1 Public
::Routine Event_Event2 Public
```

It is not acceptable if only one of the events is supported. Also, note the naming convention. The second parameter of CreateObject() names the prefix of the routine name that will support the event. The remainder of the routine name is composed of the event name from the `<event>` tag of the `<public>`

section. Neither the prefix nor the empty string can be elided. In other words, neither `CreateObject("object",)` nor `CreateObject("object","")` is allowed. The script host will generate an error.

Internet Explorer Events: When coding Internet Explorer events, the user should be aware of the following. The section of code between the quotes on an HTML tag has to be complete, with correct syntax. The `THIS` object is implicity defined for the scope of the section. If the section calls a function, and the function needs access to `THIS`, then the section must pass `THIS` as a variable to the function. `THIS` is the browser's object that represents the tag that the event was fired from. For all of the exact properties and methods associated with `THIS`, see the documentation for the corresponding tag.

To illustrate, consider the following code extract:

```
<p onmouseover="Call RxMouseOver This" id="SomeTag">
"HOT" text, get your "HOT" text right here
</p>

<script language="Object REXX">
::Routine RxMouseOver Public
Use Arg This
Text = "This is a <"This~tagName"> tag named '"THIS~id"'"
a = RxMessageBox(Text,"RxMouseOver","OK",)
Return "OK"
</script>
```

The code for the `onmouseover= "Call RxMouseOver This"` is complete and correct. If a function call had been used instead, the code would be something similar to `"a = RxMouseOver(This)"`. Do not forget to assign the results of a function call to something. If `THIS` is not passed as an argument to RxMouseOver, it will have the default value of a string whose content is `THIS`.

To cancel Internet Explorer events, the Object REXX Boolean value `.false` must be returned. The integer values `0` and `1` are not appropriate alternatives. For example:

```
<a onmouseover="call SomeFunction; return .false" href="someURL">
```

WSH Samples

There are more features to WSH than are listed here. The Samples\WSH subdirectory of your Object REXX for Windows installation directory contains some appropriate samples and an explanation of the relevant features. Before running any samples, make sure that the latest version of Windows Scripting Host is installed on the machine.

Several sample files are stand-alone; these are all of the file types `.htm`, `.wsf` or `.rex`. However, all of the samples covering the aspects of using Object REXX scripts as COM objects are in pairs or, in one case, a group of three. One file is the COM object, and the other is the script that instantiates it. All of the COM objects are of the file type `.wsc`. The files that instantiate them are either `.wsf` or `.rex`. The sample that uses three files illustrates the `include=` attribute of the `<script>` tag. All of the `.wsc` files must be registered before they can be used (see “Registering the COM Object” on page 504).

To view the `.htm` samples, use Windows Explorer to view the sample directory. Right-click the desired sample file, and choose Open With->Internet Explorer from the menu that appears.

To view the `.wsf` or `.rex` samples, use either a DOS window or Windows Explorer. From Windows Explorer, double-click the desired file. It will execute automatically. From the DOS window, make the sample directory the current directory, and use either CScript or WScript to execute the sample. The file `Print.rex` is an include file. It is not intended for direct execution.

Samples whose names begin with ″w″ use only Window pop-up boxes for output. Samples without the leading ″w″ are best viewed from the DOS window. They produce output that will not display in a Windows-only environment. Samples whose name begins with ″call″ are used to instantiate the COM objects once they are installed. If they are not installed, the error message `"Error 98.909: Class "......" not found"` will be issued.

Interpretation of and Deviation from the WSH Specification

This section deals with a number of issues to do with interpreting the WSH specification and with deviations from it.

Windows Scripting Host (WSH) Advanced Overview

Accommodating to WSH has necessitated some deviations from the Object REXX standard. To best understand what these deviations are, you need to be aware of the components of WSH. In addition to the products that are *hosts*, there are special COM objects and different mechanisms for initiating the engine.

Hosts Provided by Microsoft

Microsoft provides three fully-implemented scripting hosts. They are Microsoft Internet Explorer, CScript, and WScript. As an expansion on the concept of using a scripting language to drive external products, CScript and WScript were developed to control the Windows operating system. The two modules are so similar that they are sometimes referred to as C/WScript. CScript is intended to be used from the command line, and WScript is best used in the Windows environment. Both provide their services to the script through the WScript object. Using the default method for output `WScript~Echo()`, CScript sends the output to a console screen in the same manner as the Object REXX command `Say`, whereas `WScript~Echo()` in a script controlled by WScript will create a pop-up box in which the user must click the OK button to make it disappear.

These are not the only Microsoft products that have WSH capabilities. The core of C/Wscript is `scrobj.dll`. Several Microsoft products implement various parts of the scripting host architecture by using `scrobj.dll`.

Additional COM Objects

Since JScript and VBScript were developed primarily to manipulate the Web browser DOM (Domain Object Model), they lack many of the features associated with a language that drives an operating system. They have no native facilities for I/O (Input and Output), or for controlling the file system. These powers are granted through several additional COM objects.

Most of the literature on WSH describes these objects. Most of the features in these additional COM objects are native to Object REXX; for further information, see "The OLEObject Class" on page 284. Further documentation on the additional COM objects is readily available from other sources.

Object REXX, since it is OLE-enabled, has access to all of these objects. OLE (Object Linking and Embedding) is an advanced protocol based on COM. Be aware that the automatic object WScript is only available when Object REXX is activated by C/Wscript. Access cannot be obtained if Object REXX is initiated by Internet Explorer, or when it is initiated in the classical method `"REXX someFile.rex"`, either from the command line or from a command issued by the file explorer as an association with a file type. This is not a limitation of Object REXX. It is a consequence of the manner in which this object is loaded.

The WScript object is not registered in the Windows Registry. It exists only when C/WScript dynamically creates it and then passes the pointer to Object REXX. All scripting languages, including JScript and VBScript, have this limitation.

Where to Find Additional Documentation

The best source of up-to-date information on WSH is the World Wide Web. The keyword to use when searching the help facilities provided by Microsoft is ″scripting″. If you are using a search engine (available when you click ″Search″ on your browser's menu bar), insert ″activescript″ as the keyword.

In addition, there are several books on the subject. When browsing online bookstores, use the keyword, ″activescript″. The MSDN (Microsoft Developers Network) is a good reference source for the syntax of the XML used to define the WSH files.

Note that the correct file type to use for the XML file that C/WScript processes is `.wsf`. Existing documentation often states misleadingly that the file type to use is `.ws`. C/WScript requires the full file name, including file type, and it processes the file correctly only when the file type is `.wsf`. This seems to be hard coded into C/WScript, and no workaround is available.

Object REXX in the WSH Environment

Object REXX is fully compatible with the WSH environment. Interaction with JScript and VBScript is transparent. Legacy applications developed with these languages will not have to be discarded.

Object REXX Features Available

All of the features normally associated with Object REXX are available when Object REXX is loaded by WSH. In addition, OLE support is loaded automatically. Scripts do not need to include `'::requires "OREXXOLE.CLS"'`. However, when Object REXX is invoked by Internet Explorer, it honors the ″sandbox″ settings that the user has set in the browser's security panel. Access to I/O, the file system, external commands, and COM objects may not be granted.

Changes in Object REXX due to WSH

To comply with the WSH definition, some of the scoping rules and default behavior of Object REXX have been modified. The default behavior has been altered to allow some objects to be implicitly defined. The normal scoping rules now allow ″global″ objects to appear at any procedure depth, without requiring the use of EXPOSE, or the passing of the object as a parameter. Second-level objects can now be accessed without specifying the first level. These changes only apply to objects that WSH provides to Object REXX. All other objects and variables behave in the standard ways.

Normally, access to objects requires explicit declaration through one of the OLE methods, as in:

```
"Window = .OLEObject~new("window")"
```

Some, like WScript, can only be passed in; others – `window`, for example – have a history of being implicitly available. Full documentation is not yet available as to what objects have these features, and therefore only a few will be mentioned.

As previously mentioned, the WScript object is implicitly available when Object REXX is started by C/WScript. The ″window″ object is implicitly available when Object REXX is initiated by Internet Explorer. For events associated with an HTML tag – `ONMOUSEOVER`, for example – the scriptlet in the HTML tag has `THIS` implicitly defined. Unlike `"WINDOW"`, `THIS` is not global. Typically, this scriptlet calls a procedure, and `THIS` must be passed to the procedure if the procedure needs to reference `THIS`.

Normally, you reference an object by naming the top level object, followed by the objects at second and subsequent levels, separated by the tilde symbol (~). However, in order to emulate the current behavior of Internet Explorer, the engine must resolve object names starting at the *second* level to the appropriate top level that owns them. The shorthand `"Document~WriteLn()"` or `"Alert()"` is just as acceptable as `"Window~Document~WriteLn()"` or `"Window~Alert()"`. It is preferable, as good coding practice, to explicitly state this relationship. Stating `"Doc = Window~Document"` removes all doubt as to which global object WriteLn() is associated with when the statement `"Doc~WriteLn()"` is encountered.

Note: This applies only to global objects supplied by WSH. Objects created in or supplied by Object REXX must be named in the normal fashion.

Parameters

A called routine may receive more parameters than expected. This is not necessarily an error on the caller's part; WSH adds extra parameters on occasion. When WSH does this, Object REXX adds the parameters at the end. There is an exception to this. The documentation is ambiguous in certain sections about defining properties for scripts that are used as COM objects. If the XML that defines the script states that a name should be a property, but Object REXX finds it defined as a function, then Object REXX will prepend the parameter list with `GET` or `PUT`, depending on the direction of the property access. For more information, see the sample file `Call_ExtraParms.wsf` in the Samples\WSH subdirectory of your Object REXX for Windows installation directory.

Properties

WSH defines properties as variable values that a COM script exposes to outside routines, or strings and numbers extracted from a Typelib. Properties are to be treated as global variables within the accessing script. Properties can be implemented as variables or as functions.

Object REXX supports declaring and defining properties in the intent of the specification (see the section on `.wsc` files). That means that the variables at the highest scope, the closest to what could be considered as global, may have their values exposed as properties for other programs to use.

For another program to reference these properties, it must instantiate the COM object, and the object name must precede the property name. For example:

```
Object = .OLEObject~New("SimpleObjectREXX.Com")
/*  The next line is a property GET */
Say "The value for ExternalPropertyName is:" Object~ExternalPropertyName
Object~ExternalPropertyName = "New Value"  -- This is a PUT
```

If you experiment, you will find that there is also a shorthand method, as follows:

```
Object = .OLEObject~New("SimpleObjectREXX.Com")
/*  The next line is a property GET */
Say "The ExternalPropertyName value is:" Object~ExternalPropertyName()
Say Object~ExternalPropertyName("New Value")
```

In the case of the second reference, the method is both a PROPERTYGET and a PROPERTYPUT. It gets the old value, replacing the current one with the parameter inside the parenthesis. If more than one parameter is passed, the additional parameters are ignored.

Note: This does not always work, and is supported only by Object REXX. The cases in which it does not work are where the properties are defined as functions and not as simple variables. These calls are, in fact, methods and not property references. When Object REXX receives method calls for properties, it converts them to the appropriate action. In the case of properties defined as functions, WSH translates the property action into a function action. However, when the action is initiated as a function and not as a property, WSH does not always make the appropriate or correct translation.

Object REXX does not support the concept of global variables. For a COM script to reference its own properties, and to react to outside scripts changing them, then the properties have to be global. To meet the requirement that properties are global in scope within the defining script, the Built-In Function (BIF) Value() has been expanded to accept ″WSHPROPERTY″ as a selector when referencing properties. As with variables accessed with the ″ENVIRONMENT″ selector, these variables persist only during the life of the COM object that supplies the properties. The next time that the COM is run, the values will be at initial coded state.

The WSH supports various syntax combinations in the case of implementing a property as a function. In all combinations, the function is named in the `<property>` section or tag. It assumes that, when no function is named, the property is a variable; however, it does not enforce this assumption. It is possible to name a property and define it as a function. Object REXX defines this to mean that the function must be invoked whenever a property access is attempted. Object REXX notifies the function of the intended access direction by inserting `GET` or `PUT` as the first argument, and shifting all original arguments accordingly; that is, the original first argument is the second, the second is the third, and so on. For a demonstration of this behavior, see the `Call_PropertyORexx.wsf` sample in the Samples\WSH subdirectory of your Object REXX for Windows installation directory.

The WSH also establishes that Type Library variables may be made accessible to the script. This violates the default value and scope mechanisms of Object REXX. To meet the requirement that properties are global in scope within the defining script, the Built-In Function (BIF) Value() has been expanded to accept ″WSHTYPELIB″ as a selector when referencing elements in a Type Library. As with variables accessed with the ″ENVIRONMENT″ selector, these variables (because they are external to Object REXX) are global and persist only during the life of the COM object that supplies the properties. In addition, they are *read only*. They are immutable; they cannot be changed.

The Object REXX ″Sandbox″

Object REXX contains a feature known as the Security Manager. When this is enabled it can restrict and audit the other native abilities of Object REXX. When used with WSH, Object REXX honors the IObjectSafety interface and its methods GetInterfaceSafetyOptions() and SetInterfaceSafetyOptions() by translating their calls into Security Manager settings. This means that when Object REXX is in the Internet Explorer's sandbox, it will restrict itself to the user's settings. The most secure situation is one where Object REXX does not interact with the user's desktop (no reads or writes to the hard disk, no external commands, and so on).

Implications of Browser Applications That Run Outside the ″Sandbox″

The most useful aspect of this feature is that the user may select the most secure settings for the Internet, but allow desktop interaction for pages delivered by the local intranet server. In keeping with the current trend in IT, Object REXX allows users to leverage their investment in desktop software. This facility is intended for clients who use the intranet to lighten the client, or put a Web interface on legacy applications. A lighter client desktop means less software on the user desktop to maintain.

Features Duplicated in Object REXX and WSH

Several features are available from both WSH and Object REXX. However, the overlap is not exact, and knowing the differences can aid the user in deciding which is more appropriate to use.

Declaring Objects with Object REXX or WScript

When instantiating COM or OLE objects as REXX objects, either the native REXX .OLEObject~new() method, or the WScript~CreateObject() method can be used. The WSH method has the advantage of allowing the script to support the events that the object might fire. This is part of its definition, and no scripting language will have access to this ability in its native object enabler. The disadvantage is that it is a COM object performing a function that can be done internally.

Another disadvantage of using the WSH method becomes evident if the script is executed outside of the context of WSH. The WScript object will not exist. Therefore, unless the ability to sink events is necessary, it is suggested that the native Object REXX method be used.

Subcom versus the Host Interface

With the advent of WSH, there are two ways to use Object REXX to drive a product. The first is through the Object REXX Subcom interface. The second is for the product to become a Windows Scripting Host. The advantage of the WSH interface to the product is that it is a COM interface. This positions the product to take advantage of DCOM. This interface also allows the package developed by the user to pass objects to Object REXX.

The disadvantage is the loss of richness contained in the Subcom interface, and the loss of the close integration that a `.dll` connection has over a COM connection. The Subcom interface allows the package to tailor Object REXX in ways that are not possible through the COM interface, especially when the Object REXX Exit Handlers are implemented.

When writing a product that will be a WSH to Object REXX, refer to the sections "Concurrency" and "COM Interfaces" in "Windows Scripting Host Interface", in the *Object REXX for Windows: Programming Guide*.

.dll vs COM

There are several issues that should be considered when a choice needs to be made between a COM or a `.dll` interface. These issues stem from the intended purposes of each interface.

The `.dll` interface was developed to extend code reuse by allowing global scope subroutines and functions to be externalized into a module that is separate from the executable. When more than one executable wanted these functions, they all shared the same code that was loaded into memory. The code that was in the `.dll` executed in the frame of the `.exe` module. It had the same address space and other environmental parameters. Multiple copies of a `*.dll` code exist on a machine at one time. The first one that was found in the search path was loaded.

COM was developed to embody a flat model world; only one copy per machine. It was developed to solve two problems with the `*.dll` interface. The first was entry point resolution, and the other was using the wrong `*.dll` because the search path was not correct. COM does this by using RPC, a mechanism that was designed to communicate between different machines. For conceptual purposes, COM modules then function in a different address space from that of the invoking `*.exe`. Therefore, there is overhead in making any data that is to be passed back and forth opaque on the sender's side, and converting it into usable data on the receiver's side.

Appendix A. Using the DO Keyword

This appendix provides you with additional information about the DO keyword.

Simple DO Group

If you specify neither *repetitor* nor *conditional*, the DO construct only groups a number of instructions together. They are processed once. For 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, 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 518.)

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 processed until the condition is satisfied or a REXX instruction ends the loop (for example, LEAVE).

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 *control1*, 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 as long as the end condition (determined by the result of *exprt*) is not met. If *exprb* is positive or 0, the loop is ended when *control1* is greater than *exprt*. If negative, the loop is ended when *control1* is less than *exprt*.

The *expri*, *exprt*, and *exprb* options must result in numbers. They are evaluated only once, 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 infinitely 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 can affect the iteration of the loop. Altering the value of the control variable is not considered good programming practice, though it can 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 limited 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 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 once—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 *control1* name describing the control variable can be specified on the END clause. This *name* must match *control1* in the DO clause in all respects except the case (note that no substitution for compound variables is carried out). Otherwise, a syntax error results. 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 can affect the successive values of the control variable because REXX arithmetic rules apply to the computation of stepping the control variable.

Repetitive Loops over Collections

A *collection loop* specifies a control variable, *control2*, which receives a different value on each repetition of the loop. (For more information on *control2*, see “DO” on page 49.) These different values are taken from successive values of *collection*. The *collection* is any expression that evaluates to an object that provides a MAKEARRAY method, including stem variables. The collection returned determines the set of values and their order.

If the collection is a stem variable, the values are the tail names that have been explicitly assigned to the given stem. The order of the tail names is unspecified, and a program should not rely on any order.

For other collection objects, the MAKEARRAY method of the specific collection class determines the values assigned to the control variable.

All values for the loop iteration are obtained at the beginning of the loop. Therefore, changes to the target collection object do not affect the loop iteration. For example, using DROP to change the set of tails associated with a stem or using a new value as a tail does not change the number of loop iterations or the values over which the loop iterates.

As with controlled repetition, you can specify the symbol that describes the control variable on the END clause. The control variable is referenced by name, and you can change it within the loop (although this would not usually be useful). You can also specify the control variable name on an ITERATE or LEAVE instruction.

Example:

```
Astem.=0
Astem.3='CCC'
Astem.24='XXX'
do k over Astem.
say k Astem.k
end k
```

This example can produce:

```
3 CCC
24 XXX
```

or:

```
24 XXX
3 CCC
```

See Figure 25 on page 521 for a diagram.

Conditional Phrases (WHILE and UNTIL)

A conditional phrase can modify the iteration of a repetitive DO loop. It can 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 after each loop using the latest values of all variables, 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.

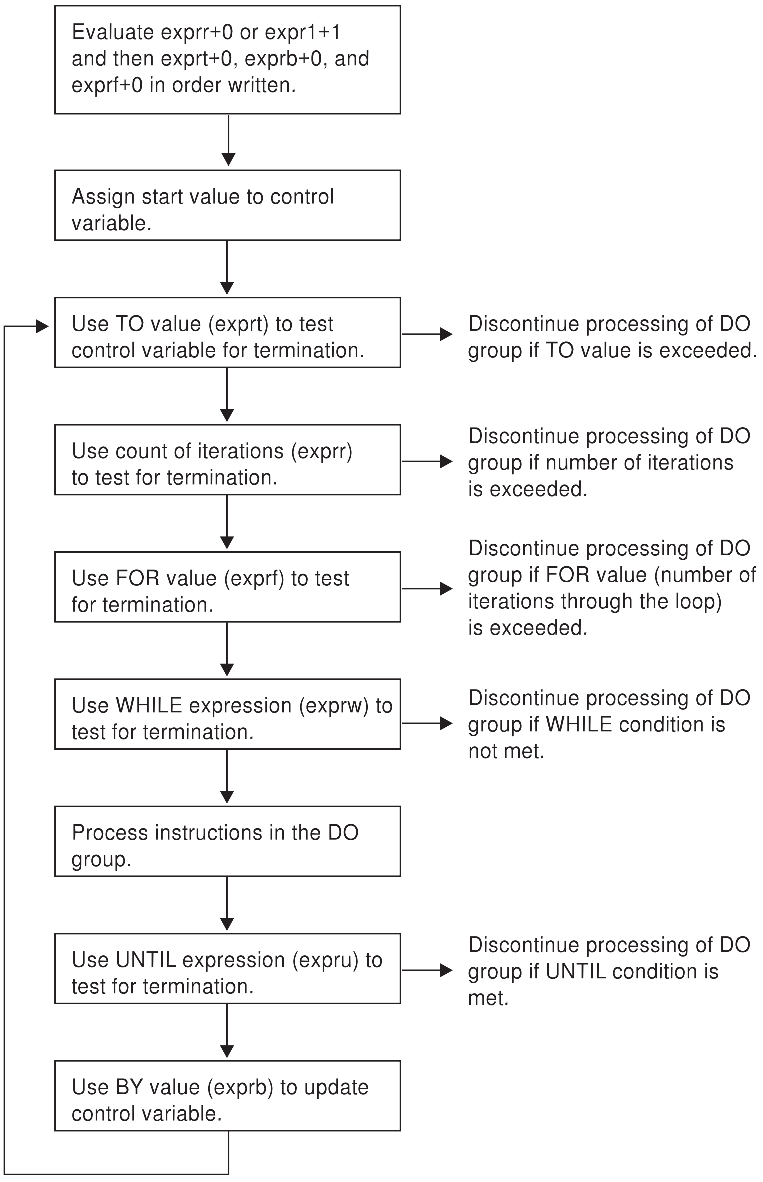

Evaluate exprr+0 or expr1+1 and then exprt+0, exprb+0, and exprf+0 in order written.
Assign start value to control variable.
Use TO value (exprt) to test control variable for termination.
Discontinue processing of DO group if TO value is exceeded.
Use count of iterations (exprr) to test for termination.
Discontinue processing of DO group if number of iterations is exceeded.
Use FOR value (exprf) to test for termination.
Discontinue processing of DO group if FOR value (number of iterations through the loop) is exceeded.
Use WHILE expression (exprw) to test for termination.
Discontinue processing of DO group if WHILE condition is not met.
Process instructions in the DO group.
Use UNTIL expression (expru) to test for termination.
Discontinue processing of DO group if UNTIL condition is met.
Use BY value (exprb) to update control variable.

Figure 24. Concept of a DO Loop

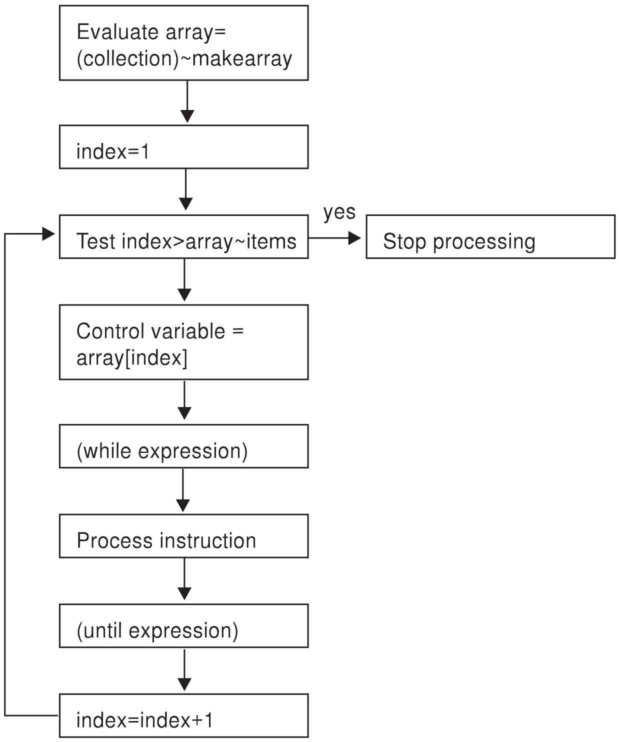


Figure 25. Concept of Repetitive Loop over Collection

Appendix B. Migration

This appendix lists some differences between Object REXX and earlier versions of REXX, and between Object REXX for OS/2 and Object REXX for Windows NT and Windows 95.

Error Codes and Return Codes

Some error codes have changed and some have been added. Also, for most errors you now receive two error messages. The first should be similar or identical to the message you would have seen previously. The second provides additional and more detailed information. So, for example, where you formerly received "Invalid Call to Routine", you now get further information on what is wrong with the call.

Also, the return codes of host commands might be different.

In Windows 95, you do not get return codes for external commands that:

- Are internal commands of the command interpreter COMMAND.COM, such as DIR, COPY, or MD
- Are 16–bit applications
- Redirect input or output

Error Detection and Reporting

Some errors are now detected earlier. Formerly, REXX would wait until it encountered an error during execution to report it to you. Now, some errors are reported before the first instruction in your REXX script is executed. In particular, syntax errors are reported after you have invoked the program, but before it starts execution.

Environment Variables

Environment variables set within an Object REXX program by the VALUE function or "SET" are not kept after the program termination.

Stems versus Collections

Stems are a general data structure that are powerful but abstract. In earlier releases of REXX, you could use stems to create data structures of all types, such as arrays, stacks, and queues. These data structures were semantically neutral. Because stems were the basis for all of them, the code itself gave no hint of which structure was implemented and for what purpose.

The best data structure job is not always the most powerful and abstract but the most specific and restrictive. Object REXX provides a variety of data structures in the collection classes. This helps reduce errors because you can select the data structure that best meets your requirements. It also helps eliminate the misuse of data structures and adds a semantic context that makes programs easier to maintain.

Input and Output Using Functions and Methods

Do not use a mixture of methods and functions for input and output because it can cause unpredictable results. For example, using the LINEOUT method and the LINEOUT function on the same persistent stream object can cause overlays.

When a REXX I/O function creates a stream object, the language processor maintains the stream object. When an I/O method creates a stream object, it is returned to the program to be maintained. Therefore, these two stream objects are separate stream objects with different read and write pointers. The program needs to synchronize the read and write pointers of both stream objects. Otherwise, overlays would occur.

.Environment

The .Environment directory in Windows is local and not system-global as in OS/2. This means that in Windows there is no difference between the scope of the .Local and .Environment directories.

Deleting Environment Variables

Value(envvar,"","ENVIRONMENT") does not delete an environment variable but sets the environment variable's value to "". Use Value(envvar,.nil,"ENVIRONMENT") to delete an environment variable.

Queuing

To improve performance it is recommended that you use the Queue class instead of RXQUEUE whenever the queued data is not to be shared among processes.

Trace in Macrospace

Functions in macrospace cannot be traced using the TRACE keyword. These functions are stored in an optimized format without source code. If you want to trace functions, do not load them into macrospace.

The RxMessageBox Function

In Windows, the RxMessageBox function does not support all the options available in OS/2. The following button styles are not available in Windows: CANCEL, ENTER, and ENTERCANCEL.

Appendix C. Error Numbers and Messages

The error numbers produced by syntax errors during the processing of REXX programs are all in the range 1 to 99. Errors are raised in response to conditions, for example, SYNTAX, NOMETHOD, and PROPAGATE. When the condition is SYNTAX, the value of the error number is placed in the variable RC when SIGNAL ON SYNTAX is trapped.

You can use the ERRORTEXT built-in function to return the text of an error message.

Some errors have associated subcodes. A subcode is a one- to three-digit decimal extension to the error number, for example, `115` in `40.115`. When an error subcode is available, additional information that further defines the source of the error is given. The ERRORTEXT built-in function cannot retrieve the secondary message, but it is available from the condition object created when SIGNAL ON SYNTAX traps an error.

Some errors are only or not displayed under certain conditions:

- Errors 3 and 5 cannot be trapped by SIGNAL ON SYNTAX.
- Error 4 can only be trapped by SIGNAL ON HALT or CALL ON HALT.
- Errors 6 and 30 can only be trapped by SIGNAL ON SYNTAX if they occur during the execution of an INTERPRET instruction.

Error List

Error 3 Failure during initialization

Explanation: The REXX program could not be read from the disk.

The associated subcodes are:

1	Failure during initialization: File "*filename*" is unreadable
901	Failure during initialization: Program "*program*" was not found
902	Error writing output file "*file*"
903	Program "*program_name*" cannot be run by this version of the REXX interpreter
904	Program "*program_name*" needs to be tokenized. To run untokenized scripts you need a full version of Object REXX. (This error occurs only with the RUNTIME version.)

Error 4 Program interrupted

Explanation: The system interrupted the execution of your REXX program because of an error or a user request.

The associated subcode is:

1	Program interrupted with *condition* condition

Error 5 System resources exhausted

Explanation: While trying to execute a program, the language processor was unable to get the resources it needed to continue. For example, it could not get the space needed for its work areas or variables. The program that called the language processor might itself have already used up most of the available storage. Or a request for storage might have been for more than the implementation maximum.

Error 6 Unmatched "/*" or quote

Explanation: A comment or literal string was started but never finished. This could be because the language processor detected:
- The end of the program (or the end of the string in an INTERPRET instruction) without finding the ending "*/" for a comment or the ending quotation mark for a literal string
- The end of the line for a literal string

The associated subcodes are:

1	Unmatched comment delimiter ("/*") on line *line_number*
2	Unmatched single quote (')
3	Unmatched double quote (")

Error 7 WHEN or OTHERWISE expected

Explanation: At least one WHEN construct (and possibly an OTHERWISE clause) is expected within a SELECT instruction. This message is issued if any other instruction is found or there is no WHEN construct before the OTHERWISE or all WHEN expressions are false and an OTHERWISE is not present. A common cause of this error is if you forget the DO and END around the list of instructions following a WHEN. For example:

```
WRONG                    RIGHT

Select                   Select
When a=c then            When a=c then DO
Say 'A equals C'         Say 'A equals C'
exit                     exit
Otherwise nop              end
end                      Otherwise nop
end
```

The associated subcodes are:

1	SELECT on line *line_number* requires WHEN
2	SELECT on line *line_number* requires WHEN, OTHERWISE, or END
3	All WHEN expressions of SELECT are false; OTHERWISE expected

Error 8 Unexpected THEN or ELSE

Explanation: A THEN or an ELSE clause was found that does not match a corresponding IF or WHEN clause. This often occurs because of a missing END or DO...END in the THEN part of a complex IF...THEN...ELSE construction. For example:

```
WRONG                      RIGHT

If a=c then do;            If a=c then do;
Say EQUALS                 Say EQUALS
exit                       exit
else                         end
Say NOT EQUALS             else
Say NOT EQUALS
```

The associated subcodes are:

1	THEN has no corresponding IF or WHEN clause
2	ELSE has no corresponding THEN clause

Error 9 Unexpected WHEN or OTHERWISE

Explanation: A WHEN or OTHERWISE was found outside of a SELECT construction. You might have accidentally enclosed the instruction in a DO...END construction by leaving out an END, or you might have tried to branch to it with a SIGNAL instruction (which does not work because the SELECT is then ended).

The associated subcodes are:

1 WHEN has no corresponding SELECT

2 OTHERWISE has no corresponding SELECT

Error 10 Unexpected or unmatched END

Explanation: More ENDs were found in your program than DO or SELECT instructions, or the ENDs did not match the DO or SELECT instructions.

This message also occurs if you try to transfer control into the middle of a loop using SIGNAL. In this case, the language processor does not expect the END because it did not process the previous DO instruction. Remember also that SIGNAL deactivates any current loops, so it cannot transfer control from one place inside a loop to another.

Another cause for this message is placing an END immediately after a THEN or ELSE subkeyword or specifying a *name* on the END keyword that does not match the *name* following DO. Putting the name of the control variable on ENDs that close repetitive loops can also help locate this kind of error.

The associated subcodes are:

1 END has no corresponding DO or SELECT

2 Symbol following END (“*symbol*”) must either match control variable of DO specification (“*control_variable*” on line *line_number*) or be omitted

3 END corresponding to DO on line *line_number* must not have a symbol following it because there is no control variable; found “*symbol*”

4 END corresponding to SELECT on line *line_number* must not have a symbol following; found “*symbol*”

5 END must not immediately follow THEN

6 END must not immediately follow ELSE

Error 11 Control stack full

Explanation: Your program exceeds the nesting level limit for control structures (for example, DO...END and IF...THEN...ELSE).

This could be because of a looping INTERPRET instruction, such as:

```
line='INTERPRET line'
INTERPRET line
```

These lines loop until they exceed the nesting level limit and the language processor issues this message. Similarly, a recursive subroutine or internal function that does not end correctly can loop until it causes this message.

The associated subcode is:

1 Insufficient control stack space; cannot continue execution

---

Error 13 Invalid character in program

Explanation: A character was found outside a literal (quoted) string that is not a blank or one of the following:

```
(Alphanumeric Characters)
A through Z, a through z, 0 through 9

(Name Characters)
! _ ? .

(Special Characters)
& * ( ) - + = ¬ ' " ; : < , > / \ | % ~ [ ]
```

The associated subcode is:

1 Incorrect character in program “*character*” ('*hex_character*'X)

Error 14 Incomplete DO/SELECT/IF

Explanation: At the end of the program or the string for an INTERPRET instruction, a DO or SELECT instruction was found without a matching END or an IF clause that is not followed by a THEN clause. Putting the name of the control variable on each END closing a controlled loop can help locate this kind of error.

The associated subcodes are:

1	DO instruction on line *line_number* requires matching END
2	SELECT instruction on line *line_number* requires matching END
3	THEN on line *line_number* must be followed by an instruction
4	ELSE on line *line_number* must be followed by an instruction
901	OTHERWISE on line *line_number* requires matching END

Error 15 Invalid hexadecimal or binary string

Explanation: Hexadecimal strings must not have leading or trailing blanks and blanks can only be embedded at byte boundaries. Only the digits `0`–`9` and the letters `a`–`f` and `A`–`F` are allowed. The following are valid hexadecimal strings:

```
'13'x
'A3C2 1c34'x
'1de8'x
```

Binary strings can have blanks only at the boundaries of groups of four binary digits. Only the digits `0` and `1` are allowed. These are valid binary strings:

```
'1011'b
'110 1101'b
'101101 11010011'b
```

You might have mistyped one of the digits, for example, typing a letter `O` instead of the number `0`. Or you might have used the one-character symbol `X` or `B` (the name of the variable `X` or `B`, respectively) after a literal string when the string is not intended as a hexadecimal or binary specification. In this case, use the explicit concatenation operator (`||`) to concatenate the string to the value of the symbol.

The associated subcodes are:

1	Incorrect location of blank in position *position* in hexadecimal string
2	Incorrect location of blank in position *position* in binary string
3	Only 0-9, a-f, A-F, and blank are valid in a hexadecimal string; found "*character*"
4	Only 0, 1, and blank are valid in a binary string; found "*character*"

Error 16 Label not found

Explanation: A SIGNAL instruction has been executed or an event for which a trap was set with SIGNAL ON has occurred, and the language processor could not find the label specified. You might have mistyped the label or forgotten to include it.

The associated subcode is:

1	Label "*label_name*" not found

Error 17 Unexpected PROCEDURE

Explanation: A PROCEDURE instruction was encountered at an incorrect position. This could occur because no internal routines are active or because the PROCEDURE instruction was not the first instruction processed after the CALL instruction or function call. One cause for this error is dropping through to an internal routine, rather than calling it with a CALL instruction or a function call.

The associated subcodes are:

1	PROCEDURE is valid only when it is the first instruction executed after an internal CALL or function invocation
901	INTERPRET data must not contain PROCEDURE

Error 18 THEN expected

Explanation: A THEN clause must follow each REXX IF or WHEN clause. The language processor found another clause before it found a THEN clause.

The associated subcodes are:

1	IF instruction on line *line_number* requires matching THEN clause

2	WHEN instruction on line *line_number* requires matching THEN clause

Error 19 String or symbol expected

Explanation: A symbol or string was expected after the CALL or SIGNAL keywords but none was found. You might have omitted the string or symbol or inserted a special character (such as a parenthesis).

The associated subcodes are:

1	String or symbol expected after ADDRESS keyword
2	String or symbol expected after CALL keyword
3	String or symbol expected after NAME keyword
4	String or symbol expected after SIGNAL keyword
6	String or symbol expected after TRACE keyword
7	String or symbol expected after PARSE keyword
901	String or symbol expected after ::CLASS keyword
902	String or symbol expected after ::METHOD keyword
903	String or symbol expected after ::ROUTINE keyword
904	String or symbol expected after ::REQUIRES keyword
905	String or symbol expected after EXTERNAL keyword
906	String or symbol expected after METACLASS keyword
907	String or symbol expected after SUBCLASS keyword
908	String or symbol expected after INHERIT keyword
909	String or symbol expected after tilde (~)
911	String or symbol expected after superclass colon (:)
912	String or symbol expected after STREAM keyword
913	String or symbol expected after MIXINCLASS keyword

Error 20 Symbol expected

Explanation: A symbol is expected after CALL ON, CALL OFF, END, ITERATE, LEAVE, NUMERIC, PARSE, SIGNAL ON, or SIGNAL OFF. Also, a list of symbols or variable references is expected after DROP, EXPOSE, and PROCEDURE EXPOSE. Either there was no symbol when one was required or the language processor found another token.

The associated subcodes are:

901	Symbol expected after DROP keyword
902	Symbol expected after EXPOSE keyword
903	Symbol expected after PARSE keyword
904	Symbol expected after PARSE VAR
905	NUMERIC must be followed by one of the keywords DIGITS, FORM, or FUZZ; found "*symbol*"
906	Symbol expected after "(" of a variable reference
907	Symbol expected after LEAVE keyword
908	Symbol expected after ITERATE keyword
909	Symbol expected after END keyword
911	Symbol expected after ON keyword
912	Symbol expected after OFF keyword
913	Symbol expected after USE ARG
914	Symbol expected after RAISE keyword
915	Symbol expected after USER keyword
916	Symbol expected after ::
917	Symbol expected after superclass colon (:)

Error 21 Invalid data on end of clause

Explanation: A clause such as SELECT or NOP is followed by a token other than a comment.

The associated subcodes are:

901 Data must not follow the NOP keyword; found "*data*"

902 Data must not follow the SELECT keyword; found "*data*"

903 Data must not follow the NAME keyword; found "*data*"

904 Data must not follow the condition name; found "*data*"

905 Data must not follow the SIGNAL label name; found "*data*"

906 Data must not follow the TRACE setting; found "*data*"

907 Data must not follow the LEAVE control variable name; found "*data*"

908 Data must not follow the ITERATE control variable name; found "*data*"

909 Data must not follow the END control variable name; found "*data*"

911 Data must not follow the NUMERIC FORM specification; found "*data*"

912 Data must not follow the GUARD OFF specification; found "*data*"

Error 22 Invalid character string

Explanation: A literal string contains character codes that are not valid. This might be because some characters are not possible, or because the character set is extended and certain character combinations are not allowed.

The associated subcode is:

1 Incorrect character string "*character_string*" ('*hex_string*'X)

Error 23 Invalid data string

Explanation: A data string (that is, the result of an expression) contains character codes that are not valid. This might be because some characters are not possible or because the character set is extended and certain character combinations are not allowed.

The associated subcode is:

1 Incorrect data string "*string*" ('*hex_string*'X)

Error 24 Invalid TRACE request

Explanation: This message is issued when:

- The option on a TRACE instruction or the argument to the built-in function does not start with A, C, E, F, I, L, N, O, or R.
- In interactive debugging, you entered a number that is not a whole number.

The associated subcodes are:

1 TRACE request letter must be one of "ACEFILNOR"; found "*value*"

901 Numeric TRACE requests are valid only from interactive debugging

Error 25 Invalid subkeyword found

Explanation: An unexpected token was found at this position of an instruction where a particular subkeyword was expected. For example, in a NUMERIC instruction, the second token must be DIGITS, FUZZ, or FORM.

The associated subcodes are:

1 CALL ON must be followed by one of the keywords ERROR, FAILURE, HALT, NOTREADY, USER, or ANY; found "*word*"

2 CALL OFF must be followed by one of the keywords ERROR, FAILURE, HALT, NOTREADY, USER, or ANY; found "*word*"

3 SIGNAL ON must be followed by one of the keywords ERROR, FAILURE, HALT, LOSTDIGITS, NOTREADY, NOMETHOD, NOSTRING, NOVALUE, SYNTAX, USER, or ANY; found "*word*"

4	SIGNAL OFF must be followed by one of the keywords ERROR, FAILURE, HALT, LOSTDIGITS, NOTREADY, NOMETHOD, NOSTRING, NOVALUE, SYNTAX, USER, or ANY; found "*word*"
11	NUMERIC FORM must be followed by one of the keywords SCIENTIFIC or ENGINEERING; found "*word*"
12	PARSE must be followed by one of the keywords ARG, LINEIN, PULL, SOURCE, VALUE, VAR, or VERSION; found "*word*"
15	NUMERIC must be followed by one of the keywords DIGITS, FORM, or FUZZ; found "*word*"
17	PROCEDURE must be followed by the keyword EXPOSE or nothing; found "*word*"
901	Unknown keyword on ::CLASS directive; found "*word*"
902	Unknown keyword on ::METHOD directive; found "*word*"
903	Unknown keyword on ::ROUTINE directive; found "*word*"
904	Unknown keyword on ::REQUIRES directive; found "*word*"
905	USE must be followed by the keyword ARG; found "*word*"
906	RAISE must be followed by one of the keywords ERROR, FAILURE, HALT, LOSTDIGITS, NOMETHOD, NOSTRING, NOTREADY, NOVALUE, or SYNTAX; found "*word*"
907	Unknown keyword on RAISE instruction; found "*word*"
908	Duplicate DESCRIPTION keyword found
909	Duplicate ADDITIONAL or ARRAY keyword found
911	Duplicate RETURN or EXIT keyword found
912	GUARD ON or GUARD OFF must be followed by the keyword WHEN; found "*word*"
913	GUARD must be followed by the keyword ON or OFF; found "*word*"
914	CALL ON condition must be followed by the keyword NAME; found "*word*"
915	SIGNAL ON condition must be followed by the keyword NAME; found "*word*"
916	Unknown keyword on FORWARD instruction; found "*keyword*"
917	Duplicate TO keyword found
918	Duplicate ARGUMENTS or ARRAY keyword found
919	Duplicate RETURN or CONTINUE keyword found
921	Duplicate CLASS keyword found
922	Duplicate MESSAGE keyword found

Error 26 Invalid whole number

Explanation: An expression was found that did not evaluate to a whole number or is greater than the limit (the default is 999 999 999):

- Positional patterns in parsing templates (including variable positional patterns)
- The operand to the right 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 *option* of the TRACE instruction.

This error is also raised if the value is not permitted (for example, a negative repetition count in a DO instruction), or the division performed during an integer divide or remainder operation does not result in a whole number.

The associated subcodes are:

2	Value of repetition count expression in DO instruction must be zero or a positive whole number; found "*value*"
3	Value of FOR expression in DO instruction must be zero or a positive whole number; found "*value*"
4	Positional pattern of PARSE template must be a whole number; found "*value*"
5	NUMERIC DIGITS value must be a positive whole number; found "*value*"
6	NUMERIC FUZZ value must be zero or a positive whole number; found "*value*"
7	Number used in TRACE setting must be a whole number; found "*value*"
8	Operand to the right of the power operator (**) must be a whole number; found "*value*"
11	Result of % operation did not result in a whole number
12	Result of // operation did not result in a whole number

Error 27 Invalid DO syntax

Explanation: A syntax error was found in the DO instruction. You probably used BY, TO, FOR, WHILE, or UNTIL twice, used a WHILE and an UTIL, or used BY, TO, or FOR when there is no control variable specified.

The associated subcodes are:

1	WHILE and UNTIL keywords cannot be used on the same DO loop
901	Incorrect data following FOREVER keyword on the DO loop; found "*data*"
902	DO keyword *keyword* can be specified only once

Error 28 Invalid LEAVE or ITERATE

Explanation: A LEAVE or ITERATE instruction was found at an incorrect position. Either no loop was active, or the name specified on the instruction did not match the control variable of any active loop.

Note that internal routine calls and the INTERPRET instruction protect DO loops by making them inactive. Therefore, for example, a LEAVE instruction in a subroutine cannot affect a DO loop in the calling routine.

You probably tried to use the SIGNAL instruction to transfer control within or into a loop. Because a SIGNAL instruction ends all active loops, any ITERATE or LEAVE instruction causes this message.

The associated subcodes are:

1	LEAVE is valid only within a repetitive DO loop
2	ITERATE is valid only within a repetitive DO loop
3	Symbol following LEAVE ("*symbol*") must either match the control variable of a current DO loop or be omitted
4	Symbol following ITERATE ("*symbol*") must either match the control variable of a current DO loop or be omitted

Error 29 Environment name too long

Explanation: The environment name specified on the ADDRESS instruction is longer than permitted for the system under which the interpreter is running.

The associated subcode is:

1	Environment name exceeds *limit* characters; found "*environment_name*"

Error 30 Name or string too long

Explanation: A variable name, label name, literal (quoted) string has exceeded the allowed limit of 250 characters.

The limit for names includes any substitutions. A possible cause of this error is if you use a period (.) in a name, causing an unexpected substitution.

Leaving off an ending quotation mark for a literal string, or putting a single quotation mark in a string, can cause this error because several clauses can be included in the string. For example, write the string `'don't'` as `'don''t'` or `"don't"`.

The associated subcodes are:

1 Name exceeds 250 characters: "*name*"

2 Literal string exceeds 250 characters: "*string*"

901 Hexadecimal literal string exceeds 250 characters: "*string*"

902 Binary literal string exceeds 250 characters: "*string*"

Error 31 Name starts with number or "."

Explanation: A variable was found whose name begins with a numeric digit or a period. You cannot assign a value to such a variable because you could then redefine numeric constants.

The associated subcodes are:

1 A value cannot be assigned to a number; found "*number*"

2 Variable symbol must not start with a number; found "*symbol*"

3 Variable symbol must not start with a "."; found "*symbol*"

Error 33 Invalid expression result

Explanation: The result of an expression was found not to be valid in the context in which it was used. Check for an illegal FUZZ or DIGITS value in a NUMERIC instruction. FUZZ must not become larger than DIGITS.

The associated subcodes are:

1 Value of NUMERIC DIGITS ("*value*") must exceed value of NUMERIC FUZZ ("*value*")

2 Value of NUMERIC DIGITS ("*value*") must not exceed *value*

901 Incorrect expression result following VALUE keyword of ADDRESS instruction

902 Incorrect expression result following VALUE keyword of SIGNAL instruction

903 Incorrect expression result following VALUE keyword of TRACE instruction

904 Incorrect expression result following SYNTAX keyword of RAISE instruction

Error 34 Logical value not 0 or 1

Explanation: An expression was found in an IF, WHEN, DO WHILE, or DO UNTIL phrase that did not result in a `0` or `1`. Any value operated on by a logical operator (¬, |, &, or &&) must result in a `0` or `1`. For example, the phrase

```
If result then exit rc
```

fails if `result` has a value other than `0` or `1`. Thus, it would be better to write the phrase:

```
If result¬=0 then exit rc
```

The associated subcodes are:

1 Value of expression following IF keyword must be exactly "0" or "1"; found "*value*"

2 Value of expression following WHEN keyword must be exactly "0" or "1"; found "*value*"

3 Value of expression following WHILE keyword must be exactly "0" or "1"; found "*value*"

4 Value of expression following UNTIL keyword must be exactly "0" or "1"; found "*value*"

5 Value of expression to the left of the logical operator "*operator*" must be exactly "0" or "1"; found "*value*"

901 Logical value must be exactly "0" or "1"; found "*value*"

902 Value of expression following GUARD keyword must be exactly "0" or "1"; found "*value*"

903	Authorization return value must be exactly “0” or “1”; found “*value*”

Error 35 Invalid expression

Explanation: An expression contains a grammatical error. Possible causes:
- An expression is missing when one is required
- You ended an expression with an operator
- You specified, in an expression, two operators next to one another with nothing in between them
- You did not specify a right parenthesis when one was required
- You used special characters (such as operators) in an intended character expression without enclosing them in quotation marks.

The associated subcodes are:

1	Invalid expression detected at “*token*”
901	Prefix operator “*operator*” is not followed by an expression term
902	Missing conditional expression following IF keyword
903	Missing conditional expression following WHEN keyword
904	Missing initial expression for DO control variable
905	Missing expression following BY keyword
906	Missing expression following TO keyword
907	Missing expression following FOR keyword
908	Missing expression following WHILE keyword
909	Missing expression following UNTIL keyword
911	Missing expression following OVER keyword
912	Missing expression following INTERPRET keyword
913	Missing expression following OPTIONS keyword
914	Missing expression following VALUE keyword of an ADDRESS instruction
915	Missing expression following VALUE keyword of a SIGNAL instruction
916	Missing expression following VALUE keyword of a TRACE instruction
917	Missing expression following VALUE keyword of a NUMERIC FORM instruction
918	Missing expression following assignment instruction
919	Operator “*operator*” is not followed by an expression term
921	Missing expression following GUARD keyword
922	Missing expression following DESCRIPTION keyword of a RAISE instruction
923	Missing expression following ADDITIONAL keyword of a RAISE instruction
924	Missing “(” on expression list of the ARRAY keyword
925	Missing expression following TO keyword of a FORWARD instruction
926	Missing expression following ARGUMENTS keyword of a FORWARD instruction
927	Missing expression following MESSAGE keyword of a FORWARD instruction
928	Missing expression following CLASS keyword of a FORWARD instruction

Error 36 Unmatched “(” or “[” in expression

Explanation: A matched parenthesis or bracket was found within an expression. There are more left parentheses than right parentheses or more left brackets than right brackets. To include a single parenthesis in a command, enclose it in quotation marks.

The associated subcodes are:

901 Left parenthesis "(" in position *position* on line *line_number* requires a corresponding right parenthesis ")"

902 Square bracket "[" in position *position* on line *line_number* requires a corresponding right square bracket "]"

Error 37 Unexpected ",", ")", or "]"

Explanation: Either a comma was found outside a function invocation, or there are too many right parentheses or right square brackets in an expression. To include a comma in a character expression, enclose it in quotation marks. For example, write the instruction:

```
Say Enter A, B, or C
```

as follows:

```
Say 'Enter A, B, or C'
```

The associated subcodes are:

1 Unexpected ","

2 Unexpected ")" in expression

901 Unmatched "]" in expression

Error 38 Invalid template or pattern

Explanation: A special character that is not allowed within a parsing template (for example, %) has been found, or the syntax of a variable pattern is incorrect (that is, no symbol was found after a left parenthesis). This message is also issued if you omit the WITH subkeyword in a PARSE VALUE instruction.

The associated subcodes are:

1 Incorrect PARSE template detected at "*column_position*"

2 Incorrect PARSE position detected at "*column_position*"

3 PARSE VALUE instruction requires WITH keyword

901 Missing PARSE relative position

Error 40 Incorrect call to routine

Explanation: An incorrect call to a routine was found. Possible causes are:

- You passed incorrect data (arguments) to the built-in or external routine.
- You passed too many arguments to the built-in, external, or internal routine.
- The external routine called was not compatible with the language processor.

If you did not try to call a routine, you might have a symbol or a string adjacent to a "(" when you meant it to be separated by a blank or other operator. The language processor would treat this as a function call. For example, write `TIME(4+5)` as follows: `TIME*(4+5)`.

The associated subcodes are:

1 External routine "*routine*" failed

3 Not enough arguments in invocation of *routine*; minimum expected is *number*

4 Too many arguments in invocation of *routine*; maximum expected is *number*

5 Missing argument in invocation of *routine*; argument *argument_number* is required

11 *function_name* argument *argument_number* must be a number; found "*value*"

12 *function_name* argument *argument_number* must be a whole number; found "*value*"

13 *function_name* argument *argument_number* must be zero or positive; found "*value*"

14 *function_name* argument *argument_number* must be positive; found "*value*"

19 *function_name* argument 2, "*value*", is not in the format described by argument 3, "*value*"

21	*function_name* argument *argument_number* must not be null
22	*function_name* argument *argument_number* must be a single character or null; found "*value*"
23	*function_name* argument *argument_number* must be a single character; found "*value*"
24	*function_name* argument *argument_number* must be a binary string; found "*value*"
25	*function_name* argument *argument_number* must be a hexadecimal string; found "*value*"
26	*function_name* argument *argument_number* must be a valid symbol; found "*value*"
27	STREAM argument 1 must be a valid stream name; found "*value*"
29	*function_name* conversion to format "*value*" is not allowed
32	RANDOM difference between argument 1 ("*argument*") and argument 2 ("*argument*") must not exceed 100000
33	RANDOM argument 1 ("*argument*") must be less than or equal to argument 2 ("*argument*")
34	SOURCELINE argument 1 ("*argument*") must be less than or equal to the number of lines in the program (*program_lines*)
35	X2D argument 1 cannot be expressed as a whole number; found "*value*"
43	*function_name* argument *argument_number* must be a single non-alphanumeric character or the null string; found "*value*"
44	*function_name* argument *argument_number*, "*value*", is a format incompatible with the separator specified in argument *argument_number*
901	Result returned by *routine* is longer than *length*: "*value*"
902	*function_name* argument *argument_number* must not exceed 999,999,999
903	*function_name* argument *argument_number* must be in the range 0-99; found "*value*"
904	*function_name* argument *argument_number* must be one of "*values*"; found "*value*"
905	TRACE setting letter must be one of ACEFILNOR; found "*value*"
912	*function_name* argument *argument_number* must be a single-dimensional array; found "*value*"
913	*function_name* argument *argument_number* must have a string value; found "*value*"
914	Unknown VALUE function variable environment selector; found "*value*"
915	Program "*program_name*" cannot be used with QUEUE:
916	Cannot read from a write-only property
917	Cannot write to a read-only property or typelib element.

Error 41 Bad arithmetic conversion

Explanation: A term in an arithmetic expression is not a valid number or has an exponent outside the allowed range of -999 999 999 to +999 999 999.

You might have mistyped a variable name, or included an arithmetic operator in a character expression without putting it in quotation marks.

The associated subcodes are:

1	Nonnumeric value ("*value*") used in arithmetic operation
3	Nonnumeric value ("*value*") used with prefix operator
4	Value of TO expression in DO instruction must be numeric; found "*value*"

5	Value of BY expression in DO instruction must be numeric; found "*value*"
6	Value of control variable expression in DO instruction must be numeric; found "*value*"
7	Exponent exceeds *number* digits; found "*value*"
901	Value of RAISE SYNTAX expression of DO instruction must be numeric; found "*value*"

Error 42 Arithmetic overflow/underflow

Explanation: The result of an arithmetic operation requires an exponent that is greater than the limit of nine digits (more than 999 999 999 or less than -999 999 999).

This error can occur during the evaluation of an expression (often as a result of trying to divide a number by 0) or while stepping a DO loop control variable.

The associated subcodes are:

1	Arithmetic overflow detected at: "*value operator value*"
2	Arithmetic underflow detected at: "*value operator value*"
3	Arithmetic overflow; divisor must not be zero
901	Arithmetic overflow; exponent ("*exponent*") exceeds *number* digits
902	Arithmetic underflow; exponent ("*exponent*") exceeds *number* digits
903	Arithmetic underflow; zero raised to a negative power

Error 43 Routine not found

Explanation: A function has been invoked within an expression or a subroutine has been invoked by a CALL, but it cannot be found. Possible reasons:

- The specified label is not in the program
- It is not the name of a built-in function
- The language processor could not locate it externally

Check if you mistyped the name.

If you did not try to call a routine, you might have put a symbol or string adjacent to a `(` when you meant it to be separated by a blank or another operator. The language processor then treats it as a function call. For example, write the string `3(4+5)` as `3*(4+5)`.

The associated subcodes are:

1	Could not find routine "*routine*"
901	Could not find routine "*routine*" for ::REQUIRES

Error 44 Function or message did not return data

Explanation: The language processor called an external routine within an expression. The routine seemed to end without error, but it did not return data for use in the expression.

You might have specified the name of a program that is not intended for use as a REXX function. Call it as a command or subroutine instead.

The associated subcode is:

1	No data returned from function "*function*"

Error 45 No data specified on function RETURN

Explanation: A REXX program has been called as a function, but returned without passing back any data.

The associated subcode is:

1	Data expected on RETURN instruction because routine "*routine*" was called as a function

Error 46 Invalid variable reference

Explanation: Within an ARG, DROP, EXPOSE, PARSE, PULL, or PROCEDURE instruction, the syntax of a variable reference (a variable whose value is to be used, indicated by its name being enclosed in parentheses) is incorrect. The right parenthesis that must immediately follow the variable name might be missing or the variable name might be misspelled.

The associated subcodes are:

1 Extra token (“*token*”) found in variable reference list; “)” expected

901 Missing “)” in variable reference

902 Extra token (“*token*”) found in USE ARG variable reference; “,” or end of instruction expected

Error 47 Unexpected label

Explanation: A label was used in the expression being evaluated for an INTERPRET instruction or in an expression entered during interactive debugging. Remove the label from the interpreted data.

The associated subcode is:

1 INTERPRET data must not contain labels; found “*label*”

Error 48 Failure in system service

Explanation: The language processor stopped processing the program because a system service, such as stream input or output or the manipulation of the external data queue, has failed to work correctly.

The associated subcode is:

1 Failure in system service: *service*

Error 49 Interpretation error

Explanation: A severe error was detected in the language processor or execution process during internal self-consistency checks.

The associated subcode is:

1 Interpretation error: unexpected failure initializing the interpreter

Error 90 External name not found

Explanation: An external class, method, or routine (specified with the EXTERNAL option on a ::CLASS, ::METHOD, or ::ROUTINE directive, or as a second argument on a NEW message to the Method class) cannot be found.

The associated subcodes are:

997 Unable to find external class “*class*”

998 Unable to find external method “*method*”

999 Unable to find external routine “*routine*”

Error 91 No result object

Explanation: A message term requires a result object, but the method did not return one.

The associated subcode is:

999 Message “*message*” did not return a result object

Error 92 OLE error

Explanation: An error occurred in the OLEObject class.

The associated subcodes are:

901 An unknown OLE error occurred (HRESULT=*errcode*).

902 The conversion of the VARIANT type *type* to a REXX object failed.

903 The conversion of REXX object *object* to a VARIANT type failed.

904 The number of elements provided to the method or property is different from the number of parameters accepted by it.

905 One of the parameters is not a valid VARIANT type.

906 The OLE object reported an exception "*text*". The text provides a code number, the source of the exception, and a description of the exception.

907 The requested method does not exist, or you tried to set the value of a read-only property.

908 One of the parameters could not be converted to the desired type.

909 One or more of the parameters could not be converted to the desired type. The first parameter with an incorrect type is argument *argnum*.

910 A required parameter was omitted.

911 The creation of an instance of the OLE object failed. Possible causes: the OLE object does not support an automation interface, or its initialization failed.

912 The object invoked has disconnected from its clients.

Error 93 Incorrect call to method

Explanation: The specified method or built-in or external routine exists, but you used it incorrectly.

The associated subcodes are:

901 Not enough arguments in method; *number* expected

902 Too many arguments in invocation of method; *number* expected

903 Missing argument in method; argument *argument* is required

904 Method argument *argument* must be a number; found "*value*"

905 Method argument *argument* must be a whole number; found "*value*"

906 Method argument *argument* must be zero or a positive whole number; found *value*

907 Method argument *argument* must be a positive whole number; found "*value*"

908 Method argument *argument* must not exceed *limit*; found "*value*"

909 Method argument *argument* must be in the range 0-99; found "*value*"

911 Method argument *argument* must not be null

912 Method argument *argument* must be a hexadecimal string; found "*value*"

913 Method argument *argument* must be a valid symbol; found "*value*"

914 Method argument *argument* must be one of *arguments*; found "*value*"

915 Method option must be one of *arguments*; found "*value*"

916 Method argument *argument* must have a string value

917 Method *method* does not exist

918 Incorrect list index "*index*"

919 Incorrect array position "*position*"

921 Argument missing on binary operator

922 Incorrect pad or character argument specified; found "*value*"

923 Incorrect length argument specified; found "*value*"

924 Incorrect position argument specified; found "*value*"

925 Not enough subscripts for array; *number* expected

926 Too many subscripts for array; *number* expected

927 Length must be specified to convert a negative value

928 D2X value must be a valid whole number; found "*value*"

929 D2C value must be a valid whole number; found "*value*"

931 Incorrect location of blank in position *position* in hexadecimal string

932 Incorrect location of blank in position *position* in binary string

933	Only 0-9, a-f, A-F, and blank are valid in a hexadecimal string; character found "*character*"
934	Only 0, 1, and blank are valid in a binary string; character found "*character*"
935	X2D result is not a valid whole number with NUMERIC DIGITS *digits*
936	C2D result is not a valid whole number with NUMERIC DIGITS *digits*
937	No more supplier items available
938	Method argument *argument* must have a string value
939	Method argument *argument* must have a single-dimensional array value
941	Exponent *exponent* is too large for *number* spaces
942	Integer part *integer* is too large for *number* spaces
943	*method* method target must be a number; found "value"
944	Method argument *argument* must be a message object
945	Missing argument in message array; argument *argument* is required
946	A message array must be a single-dimensional array with 2 elements
947	Method SECTION can be used only on single-dimensional arrays
948	Method argument *argument* must be of the *class* class
949	The index and value objects must be the same for PUT to an index-only collection
951	Incorrect alarm time; found "*time*"
952	Method argument *argument* is an array and does not contain all string values
953	Method argument *argument* could not be converted to type *type*
954	Method "*method*" can be used only on a single-dimensional array
956	Element *element* of the array must be a string
957	Element *element* of the array must be a subclass of the target object
958	Positioning of transient streams is not valid
959	An array cannot contain more than 99,999,999 elements
961	Method argument *argument* must have a string value or an array value

Error 97 Object method not found

Explanation: The object does not have a method with the given name. A frequent cause of this error is an uninitialized variable.

The associated subcode is:

1	Object "*object*" does not understand message "*message*"

Error 98 Execution error

Explanation: The language processor detected a specific error during execution.

The associated subcodes are:

902	Unable to convert object "*object*" to a double-float value
903	Unable to load library "*name*"
904	Abnormal termination occurred
905	Deadlock detected on a guarded method
906	Incorrect object reference detected
907	Object of type "*type*" was required
908	Metaclass "*metaclass*" not found
909	Class "*class*" not found
911	Cyclic inheritance in program "*program*"
913	Unable to convert object "*object*" to a single-dimensional array value

914	Unable to convert object “*object*” to a string value
915	A message object cannot be sent more than one SEND or START message
916	Message object “*object*” received an error from message “*message*”
917	Incorrect condition object received for RAISE OBJECT; found “*value*”
918	No active condition available for PROPAGATE
919	Unable to convert object “*object*” to a method
921	Could not retrieve “*value*” information for method “*method*”
931	No method description information for method “*method*” on class “*class*”
934	The number of OUT or INOUT type arguments cannot exceed *number*
935	REPLY can be issued only once per method invocation
936	RETURN cannot return a value after a REPLY
937	EXIT cannot return a value after a REPLY
938	Message search overrides can be used only from methods of the target object
939	Additional information for SYNTAX errors must be a single-dimensional array of values
941	Unknown error number specified on RAISE SYNTAX; found “*number*”
942	Class “*class*” must be a MIXINCLASS for INHERIT
943	Class “*class*” is not a subclass of “*class*” base class “*class*”
944	Class “*class*” cannot inherit from itself, a superclass, or a subclass (“*class*”)
945	Class “*class*” has not inherited class “*class*”
946	FORWARD arguments must be a single-dimensional array of values
947	FORWARD can only be issued in an object method invocation
948	Authorization failure: *value*
951	Concurrency not supported
952	*servername* class server not installed

Error 99 Translation error

Explanation: An error was detected in the language syntax.

The associated subcodes are:

901	Duplicate ::CLASS directive instruction
902	Duplicate ::METHOD directive instruction
903	Duplicate ::ROUTINE directive instruction
904	Duplicate ::REQUIRES directive instruction
905	CLASS keyword on ::METHOD directive requires a matching ::CLASS directive
907	EXPOSE must be the first instruction executed after a method invocation
908	INTERPRET data must not contain EXPOSE
909	GUARD must be the first instruction executed after EXPOSE or USE
911	GUARD can only be issued in an object method invocation
912	INTERPRET data must not contain GUARD
913	GUARD instruction did not include references to exposed variables
914	INTERPRET data must not contain directive instructions
915	INTERPRET data must not contain USE
916	Unrecognized directive instruction

917 Incorrect external directive name "*method*"

918 USE ARG requires a "," between variable names; found "token"

919 REPLY can only be issued in an object method invocation

921 Incorrect program line in method source array

922 ::REQUIRES directives must appear before other directive instructions

923 INTERPRET data must not contain FORWARD

924 INTERPRET data must not contain REPLY

925 An ATTRIBUTE method name must be a valid variable name; found "*name*"

926 Incorrect class external; too many parameters

927 "*classname*" is not a valid metaclass

928 Incorrect class external; class name missing or invalid

929 Incorrect class external; invalid class server "*servername*"

RXSUBCOM Utility Program

RXSUBCOM issues the following errors:

Error 115 **The RXSUBCOM parameters are incorrect.**

Explanation: You can use RXSUBCOM as follows:

RXSUBCOM REGISTER
Registers a dynamic link library subcommand handler. (See "RXSUBCOM REGISTER" on page 485.)

RXSUBCOM DROP
Deregisters a subcommand handler. (See "RXSUBCOM DROP" on page 486.)

RXSUBCOM QUERY
Checks the existence of a subcommand handler. (See "RXSUBCOM QUERY" on page 487.)

RXSUBCOM LOAD
Loads a subcommand handler dynamic link library. (See "RXSUBCOM LOAD" on page 487.)

Check the RXSUBCOM parameters and retry the command.

Error 116 **The RXSUBCOM parameter REGISTER is incorrect.**

Explanation: Check the parameters (see "RXSUBCOM REGISTER" on page 485) and retry the command.

Error 117 **The RXSUBCOM parameter DROP is incorrect.**

Explanation: Check the parameters (see "RXSUBCOM DROP" on page 486) and retry the command.

Error 118 **The RXSUBCOM parameter LOAD is incorrect.**

Explanation: Check the parameters (see "RXSUBCOM LOAD" on page 487) and retry the command.

Error 125 **The RXSUBCOM parameter QUERY is incorrect.**

Explanation: RXSUBCOM QUERY requires the environment name to be specified. Check the RXSUBCOM (see "RXSUBCOM QUERY" on page 487) and retry the command.

RXQUEUE Utility Program

RXQUEUE issues the following errors:

Error 119 The REXX queuing system is not initialized.

Explanation: The queuing system requires a housekeeping program to run. This program usually runs under the Presentation Manager® shell. The program is not running.

Report this message to your IBM service representative.

Error 120 The size of the data is incorrect.

Explanation: The data supplied to the RXQUEUE command is too long.

The RXQUEUE program accepts data records containing 0 - 65472 bytes. A record exceeded the allowable limits.

Use shorter data records.

Error 121 Storage for data queues is exhausted.

Explanation: The queuing system is out of memory. No more storage is available to store queued data.

Delete some queues or remove queued data from the system. Then retry your request.

Error 122 The name *name* is not a valid queue name.

Explanation: The queue name contains an invalid character. Only the following characters can appear in queue names:

```
'A'..'Z', '0'..'9', '.', '!', '?', '_'
```

Change the queue name and retry the command.

Error 123 The queue access mode is not correct.

Explanation: An internal error occurred in RXQUEUE.

The RXQUEUE program tried to access a queue with an incorrect access mode. Correct access modes are LIFO and FIFO.

Report this message to your IBM service representative.

Error 124 The queue *name* does not exist.

Explanation: The command attempted to access a nonexistent queue.

Create the queue and try again, or use a queue that has been created.

REXXC Utility Program

REXXC issues the following invocation errors:

Error 127 The REXXC command parameters are incorrect.

Explanation: The REXXC utility was invoked with zero or more than two parameters. REXXC accepts the following parameters:

- To check the syntax of a REXX program:

  ```
  REXXC
  ```

- To convert a REXX program into a sourceless executable file:

  ```
  REXXC InProgramName [OutProgramName] [/s]
  ```

For more details, refer to REXXC command ("Distributing Programs without Source" on page 490).

Error 128 Output file name must be different from input file name.

Error 131 The syntax of the command is incorrect.

Explanation: The RXQUEUE program was invoked with incorrect parameters. It can be started with the following parameters:

```
RXQUEUE [/FIFO | /LIFO | /CLEAR] [queuename]
```

For more details, refer to “The RXQUEUE Filter” on page 488.

Error 132 **System error occurred while processing the command.**

Explanation: An internal error occurred in RXQUEUE. RXQUEUE was unable to allocate the required system resources.

REXXRT Utility Program

REXXRT issues the following invocation errors:

Error 0 **Success.**

Explanation: The operation succeeded.

Error 1 **Input file is not in tokenized format!**

Explanation: The input file can be converted only if it is tokenized.

Error 2 **Input file is damaged!**

Explanation: The input file is in tokenized format, but is apparently corrupted.

Error 3 **Version conflict. The script was not tokenized with a previous version!**

Explanation: The input file is already in the new format and thus cannot be converted.

Appendix D. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will

Trademarks

Index

Special Characters

Readers' Comments — We'd Like to Hear from You

Object REXX for Windows
Reference
Version 2.1

Publication No. SH12-6725-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
SH12-6725-00

IBM®

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Deutschland Entwicklung GmbH
Information Development, Dept. 0446
Schoenaicher Str. 220
71032 Boeblingen
Germany

Fold and Tape **Please do not staple** Fold and Tape

SH12-6725-00

Cut or Fold
Along Line

Part Number: CT81CIE
Program Number: 5639-M68 Development Edition
5639-M69 Interpreter Edition

Printed in Denmark by IBM Danmark A/S

SH12-6725-00

(1P) P/N: CT81CIE