Object REXX for Linux

Programming Guide

Version 1.2

Object REXX for Linux

Programming Guide

Version 1.2

Note!
Before using this information and the product it supports, be sure to read the general information under "Appendix C. Notices" on page 157.

Second Edition, March 1999

This edition applies to Version 1.2 of IBM Object REXX for Linux, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters.

Contents

About This Book

This book describes the Object-Oriented REstructured eXtended eXecutor, or Object REXX programming language. In the following, it is called **REXX** unless compared to its traditional predecessor.

This book is aimed at developers familiar with Linux who want to use REXX for object-oriented programming, or a mix of traditional and object-oriented programming.

This book assumes you are already familiar with the techniques of traditional structured programming, and uses them as a springboard for quickly understanding REXX and, in particular, Object REXX. This approach is designed to help experienced programmers get involved quickly with the REXX language, exploit its virtues, and become productive fast.

Who Should Read This Book

Anyone interested in getting a basic understanding of object-oriented concepts should read this book. Experienced programmers can learn about the REXX language and how it is like and unlike other structured programming languages. Programmers who want to broaden their programming knowledge can learn object-oriented programming with REXX. Users already experienced with REXX can learn about object-oriented programming (OO) in general, and OO programming with REXX in particular.

Programmers who want to make their applications (typically coded in C) scriptable by REXX, extend the REXX language, or control REXX scripts from other applications should read the application programming interface (API) information.

What You Should Know before Reading This Book

To most effectively use the information contained in this book, you should know how to:

- Program with a traditional language like C, Basic, or Pascal.
- Use basic Linuxcommands for manipulating files, such as **cp**, **rm**, and **ls**. The more familiar you are with Linux, the better.

If you have little or no experience with the Linux operating system, refer to the documentation provided with it. You should also have the *Object REXX for Linux: Reference* at hand.

Except in examples, all commands, program names, file names, and library names that must be typed in lowercase characters, are printed in bold to improve readability.

Related Information

Object REXX for Linux: Reference

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://www2.hursley.ibm.com/orexx. There you will find the feedback page where you can enter comments and send them.
- Send your comments by e-mail to swsdid@de.ibm.com, or to the IBMMAIL address DEIBM3P3@IBMMAIL. 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)7031-16-6901.

Chapter 1. Meet Object REXX

REXX is a versatile, free-format language. Its simplicity makes it a good first language for beginners. For more experienced users and computer professionals, REXX offers powerful functions and the ability to issue commands to several environments.

The Main Attractions

The following aspects of REXX round out its versatility and functions.

Object-Oriented Programming

Object-oriented extensions have been added to traditional REXX, but its existing functions and instructions have not changed. The Object REXX interpreter is actually an enhanced version of its predecessor, but with new support for:

- Classes, objects, and methods
- Messaging and polymorphism
- Inheritance and multiple inheritance

Object REXX supplies the user with a base set of classes: ALARM, CLASS, ARRAY, LIST, QUEUE, TABLE, SET, DIRECTORY, RELATION, BAG, MESSAGE, METHOD, MONITOR, STEM, STREAM, STRING, and SUPPLIER. Object REXX is fully compatible with earlier versions of REXX that were not object-oriented.

An English-Like Language

To make REXX easier to learn and use, many of its instructions are meaningful English words. REXX instructions are common words such as SAY, PULL, IF...THEN...ELSE, DO...END, and EXIT.

Fewer Rules

REXX has relatively few rules about format. A single instruction can span many lines, and you can include several instructions on a single line. Instructions need not begin in a particular column and can be typed in uppercase, lowercase, or mixed case. You can skip spaces in a line or entire lines. There is no line numbering.

Interpreted, Not Compiled

REXX is an interpreted language. When a REXX program runs, its language processor reads each statement from the source file and runs it, one statement at a time. Languages that are not interpreted must be compiled into object code before they can be run.

Built-In Functions and Methods

REXX has built-in functions and methods that perform various processing, searching, and comparison operations for text and numbers and provide formatting capabilities and arithmetic calculations.

Typeless Variables

REXX regards all data as objects of various kinds. Variables can hold any kind of object, so you need not declare variables as strings or numbers.

String Handling

REXX includes capabilities for manipulating character strings. This allows programs to read and separate characters, numbers, and mixed input. REXX performs arithmetic operations on any string that represents a valid number, including those in exponential formats.

Clear Error Messages and Powerful Debugging

REXX displays messages with meaningful explanations when a REXX program encounters an error. In addition, the TRACE instruction provides a powerful debugging tool.

REXX and the Linux Operating System

The most important role REXX plays is as a programming language for Linux. A REXX program can serve as a script for the Linux operating system to follow. Using REXX, you can reduce long, complex, or repetitious tasks to a single command or program.

A Classic Language Gets Classier

Object-oriented extensions have been added to traditional REXX without changing its existing functions and instructions. So you can continue to use REXX's procedural instructions, and incorporate objects as you become more comfortable with the technology. In general, your current REXX programs will work without change. But because Object REXX catches more errors at translate time than traditional REXX, you may have to fix these.

In object-oriented technology, *objects* are used in programs to model the real world. Similar objects are grouped into *classes*, and the classes themselves are arranged in hierarchies.

As an object-oriented programmer, you solve problems by identifying and classifying objects related to the problem. Then you determine what actions or behaviors are required of those objects. Finally, you write the instructions to generate the classes, create the objects, and implement the actions. Your main program consists of instructions that send messages to objects.

A billing application, for example, might have an Invoice class and a Receipt class. These two classes might be members of a Forms class. Individual invoices are *instances* of the Invoice class.

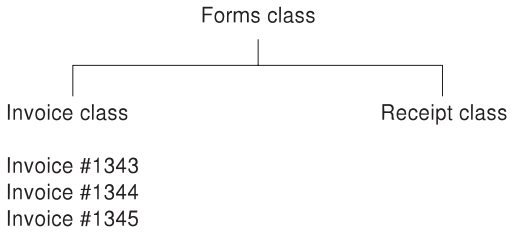


Figure 1. Objects in a Billing Application

Each instance contains all the data associated with it (such as customer name or descriptions and prices of items purchased). To get at the data, you write instructions that send *messages* to the objects. These messages activate coded actions called *methods*. For an invoice object, you might need CREATE, DISPLAY, PRINT, UPDATE, and ERASE methods.

From Traditional REXX to Object REXX

In traditional, or classic, REXX, all data was stored as *strings.* The strings represented character data as well as numeric data. From an object-oriented perspective, traditional REXX had one kind of objects: strings. In object-oriented terminology, each string variable was an *object* that was an *instance* of the String class.

In Object REXX, variables can now reference objects other than strings. In addition to a string class, REXX now includes classes for creating arrays, queues, streams, and many other useful objects. Objects in these new REXX classes are manipulated by *methods* instead of traditional functions. To activate a method, you just send the object a *message*.

For example, instead of using the SUBSTR function on a string variable Name, you send a SUBSTR message to the string object. Here is the old way:

```
s=substr(name,2,3)
```

And here is the new way:

```
s=name~substr(2,3)
```

The tilde (~) character is the REXX *message send* operator, called *twiddle*. The object receiving the message is to the left of the twiddle. The message sent is to the right. In this example, the Name object is sent the SUBSTR message. The numbers in parentheses (2,3) are arguments sent as part of the message. The SUBSTR method is run for the Name object, and the result is assigned to the *s* string object.

For the new classes (such as array or queue), methods are provided, but not equivalent functions. For example, suppose you want to use the new REXX array object instead of the traditional string-based stem variables (such as text.1 or text.2). To create an array object of five elements, you would send a NEW message to the array class as follows:

```
myarray=.array~new(5)
```

A new instance, named Myarray, of the Array class is created. The period is necessary because it precedes a built-in class name in an expression. The Myarray array object has five elements. Some of the other methods, besides NEW, for array objects are PUT, AT, REMOVE, SIZE, [], and []=.

By adding object technology to its repertoire of traditional programming techniques, REXX has evolved into an object-oriented language, like Smalltalk. Object REXX accommodates the programming techniques of traditional REXX while adding new ones. With Object REXX, you can use the new technology as much or as little as you like, at whatever pace you like. You can mix classic and object techniques. You can ease into the object world gradually, building on the REXX skills and knowledge you already have.

The Object Advantage

If you are unsure about whether to employ REXX's object-oriented features, here are some tips to help you decide.

Object-oriented technology reinforces sound programming practices, such as hiding your data from code that does not use it (encapsulation and polymorphism), partitioning your program in small, manageable units (classification and data abstraction), reusing code wherever possible and changing it in one place (inheritance and functional decomposition).

Other advantages often associated with object technology are:

- Simplified design through modeling with objects

- Greater code reuse
- Rapid prototyping
- The higher quality of proven components
- Easier and reduced maintenance
- Cost-savings potential
- Increased adaptability and scalability

With Object REXX, you get the user-friendliness of REXX in an object-oriented environment.

Object REXX provides a Sockets API for REXX and an FTP API. So you can script Object REXX clients and servers, and run them in the Internet.

The Next Step

If you already know traditional REXX and want to go straight to the basic concepts of object-oriented programming, continue with "Chapter 3. Into the Object World" on page 21.

If you are unfamiliar with traditional REXX, continue to read "Chapter 2. A Quick Tour of Traditional REXX" on page 7.

Chapter 2. A Quick Tour of Traditional REXX

Because this book is for Linux programmers, it is assumed that you are familiar with at least one other language. This chapter gives an overview of the basic REXX rules and shows you in which respects REXX is similar to, or different from, other languages you may already know.

What Is a REXX Program?

A *REXX program* is a text file, typically created using a text editor or a word processor that contains a list of instructions for your computer. REXX programs are interpreted, which means the program is, like a batch file, processed line by line. Consequently, you do not have to compile and link REXX programs. To run a REXX program, all you need is Linux, the Object REXX interpreter, and the ASCII text file containing the program.

REXX is similar to programming languages such as C, Pascal, or Basic. An important difference is that REXX variables have no data type and are not declared. Instead, REXX determines from context whether the variable is, for example, a string or a number. Moreover, a variable that was treated as a number in one instruction can be treated as a string in the next. REXX keeps track of the variables for you. It allocates and deallocates memory as necessary.

Another important difference is that you can execute Linux commands and other applications from a REXX program. This is similar to what you can do with a Linux Batch facility program. However, in addition to executing the command, you can also receive a return code from the command and use any displayed output in your REXX program. For example, the output displayed by an **ls** command can be intercepted by a REXX program and used in subsequent processing.

REXX can also direct commands to environments other than Linux. Some applications provide an environment to which REXX can direct subcommands of the application. Or they also provide functions that can be called from a REXX program. In these situations, REXX acts as a macrolanguage for the application.

Running a REXX Program

To ensure compatibility with OS/2® and Windows, REXX programs should have a file extension of **cmd**. Here is a typical REXX program named **greeting.cmd**. It prompts the user to type in a name and then displays a personalized greeting:

```
/* greeting.cmd - a REXX program to display a greeting.  */
say 'Please enter your name.'    /* Display a message    */
pull name                        /* Read response        */
say 'Hello' name                 /* Display greeting     */
exit 0                   /* Exit with a return code of 0 */
```

The program begins with a comment. This is not a requirement, but it is recommended to ensure compatibility with other operating systems, such as OS/2, where the first line in a REXX program must be a comment line.

The Object REXX interpreter is invoked by the command

```
rexx
```

To run the program **myrexx.cmd**, for example, use the command **rexx myrexx.cmd**.

SAY is a REXX instruction that displays a message (like PRINT in Basic or printf in C). The message to be displayed follows the SAY keyword. The single quotes are necessary to delimit a character string. In this case, the character string is `Please enter your name`. You can use double quotes (") instead of single quotes.

The PULL instruction reads a line of text from the standard input (the keyboard), and returns the text in the variable specified with the instruction. In our example, the text is returned in the variable *name*.

The next SAY instruction provides a glimpse of what can be done with REXX strings. It displays the word `Hello` followed by the name of the user, which is stored in variable *name*. REXX substitutes the value of *name* and displays the resulting string. You do not need a separate format string as you do with C or Basic.

The final instruction, EXIT, ends the REXX program. Control returns to Linux. EXIT can also return a value. In our example, 0 is returned. The EXIT instruction is optional.

You can terminate a running REXX program by pressing the Control (Ctrl)+Break keys. REXX stops running the program and control returns to Linux.

Elements of REXX

REXX programs are made up of *clauses*. Each clause is a complete REXX instruction.

REXX instructions include the obligatory program control verbs (IF, SELECT, DO, CALL, RETURN) as well as verbs that are unique to REXX (such as PARSE, GUARD, and EXPOSE). In all, there are about 30 instructions. Many REXX programs use only a small subset of the instructions.

A wide variety of built-in functions complements the instruction set. Many functions manipulate strings (such as SUBSTR, WORDS, POS, and SUBWORD). Other functions perform stream I/Os (such as CHARIN, CHAROUT, LINEIN, and LINEOUT). Still other functions perform data conversion (such as X2B, X2C, D2X, and C2D). A quick glance through the functions section of the *Object REXX for Linux* gives you an idea of the scope of capabilities available to you.

The built-in functions are also available in REXX implementations on other operating systems. In addition to these system-independent functions, REXX includes a set of functions for working with Linux itself. These functions, known as the *REXX Utilities*, let you work with resources managed by Linux, such as the display and the file system.

Instructions and functions are the building blocks of traditional REXX programs. To convert REXX into an object-oriented language, two more elements are needed: classes and methods. Classes and methods are covered in later chapters. This chapter continues with traditional building blocks of REXX.

Writing Your Program

You can create REXX programs using any editor that can write straight ASCII files without hidden format controls. **vi** is an editor that you can use.

REXX is a free-format programming language. You can indent lines and insert blank lines for readability if you wish. But even free-format languages have some rules about how language elements are used. REXX's rules center around its basic language element: the clause.

Usually, there is one clause on each line of the program, but you can put several and separate each clause with a semicolon (;):

```
say "Hello"; say "Goodbye"  /* Two clauses on one line */
```

To continue a clause on a second line, put a comma at the end of the line:

```
say,             /* Continuation */
"It isn't so"
```

If you need to continue a literal string, do it like this:

```
say,             /* Continuation of literal strings */
"This is a long string that we want to continue",
"on another line."
```

REXX automatically adds a blank after `continue`. If you need to split a string, but do not want to have a blank inserted when REXX puts the string back together, use the REXX concatenation operator (`||`):

```
say "I do not want REXX to in"||,   /* Continuation with concatenation */
"sert a blank!"
```

Testing Your Program

When writing your program, you can test statements as you go along using the **rexxtry** command from the Linux command prompt. **rexxtry** is a kind of REXX mini-interpreter that checks REXX statements one at a time. If you run **rexxtry** with no parameter, or with a question mark as a parameter, **rexxtry** also briefly describes itself.

From your current Linux window, open another window and type:

```
rexx rexxtry
```

rexxtry describes itself and asks you for a REXX statement to test. Enter your statement; **rexxtry** then runs it and returns any information available, or displays an error message if a problem is encountered. **rexxtry** remembers any previous statements you have entered during the session. To continue, type the next line in your program and **rexxtry** will check it for you.

Enter an equal sign (=) to repeat your previous statement, or a question mark (?) to invoke system-provided online information about the REXX language.

When you are done, type:

```
exit
```

and press Enter to leave **rexxtry**.

You can also enter a REXX statement directly on the command line for immediate processing and exit:

```
rexx rexxtry call show
```

In this case, entering **call show** displays the user variables provided by **rexxtry**.

Variables, Constants, and Literal Strings

Comprehensive rules for variables, constants, and literal strings are contained in the *Object REXX for Linux*.

REXX imposes few rules on variable names. A variable name can be up to 250 characters long, with the following restrictions:

- The first character must be A-Z, a-z, !, ?, or _ . REXX translates lowercase letters to uppercase before using them.
- The rest of the characters may be A-Z, a-z, !, ?, or _, ., or 0-9.
- The period (.) has a special meaning for REXX variables. Do not use it in a variable name until you understand the rules for forming compound symbols.

The variable name can be typed and queried in uppercase, mixed-case, or lowercase characters. A variable name in uppercase characters, for example, can also be queried in lowercase or mixed-case characters. If you query a variable name that has not yet been set, the name is returned in uppercase characters.

Literal strings in REXX are delimited by quotation marks (either ′ or ″). Examples of literal strings are:

```
'Hello'
"Final result:"
```

If you need to use quotation marks within a literal string, use quotation marks of the other type to delimit the string. For example:

```
"Don't panic"
'He said, "Bother"'
```

There is another way to do this. Within a literal string, a pair of quotation marks (the same type that delimits the string) is interpreted as one of that type. For example:

```
'Don''t panic'                (same as "Don't panic"      )
"He said, ""Bother"""         (same as 'He said, "Bother"')
```

Assignments

Assignments in REXX usually take this form:

```
name = expression
```

For *name*, specify any valid variable name. For *expression*, specify the information to be stored, such as a number, a string, or some calculation. Here are some examples:

```
a=1+2
b=a*1.5
c="This is a string assignment. No memory allocation needed!"
```

The PARSE instruction and its variants PULL and ARG also assign values to variables. PARSE assigns data from various sources to one or more variables according to the rules of parsing. PARSE PULL, for example, is often used to read data from the keyboard:

```
/* Using PARSE PULL to read the keyboard                          */
say 'Enter your first name and last name'   /* prompt user        */
parse pull response       /* read keyboard and put result in RESPONSE */
say response              /* possibly displays 'John Smith'       */
```

Other operands of PARSE indicate the source of the data. PARSE ARG, for example, retrieves command line arguments. PARSE VERSION retrieves the information about the version of the REXX interpreter being used.

The most powerful feature of PARSE, however, is its ability to parse data according to a template that you supply. The various pieces of data are assigned to variables that are part of the template. The following example prompts the user for a date, and assigns the month, day, and year to different variables. (In a real application, you would want to add instructions to verify the input.)

```
/* PARSE example using a template */
say 'Enter a date in the form MM/DD/YY'
parse pull month '/' day '/' year
say month
say day
say year
```

The template in this example contains two literal strings ('/'). The PARSE instruction uses these literals to determine how to split the data.

The PULL and ARG instructions are short forms of the PARSE instruction. See the *Object REXX for Linux: Reference* for more information on REXX parsing.

Using Functions

REXX functions can be used in any expression. In the following example, the built-in function WORD is used to return the third blank-delimited word in a string:

```
/* Example of function use                                    */
myname="John Q. Public"   /* assign a literal string to MYNAME */
surname=word(myname,3)    /* assign WORD result to SURNAME     */
say surname               /* display Public                    */
```

Literal strings can be supplied as arguments to functions, so the previous program can be rewritten as follows:

```
/* Example of function use                                        */
surname=word("John Q. Public",3) /* assign WORD result to SURNAME     */
say surname                      /* display Public                    */
```

Because an expression can be used with the SAY instruction, you can further reduce the program to:

```
/* Example of function use                                        */
say word("John Q. Public",3)
```

Functions can be nested. Suppose you want to display only the first two letters of the third word, Public. The LEFT function can return the first two letters, but you need to give it the third word. LEFT expects the input string as its first argument and the number of characters to return as its second argument:

```
/* Example of function use */

/* Here is how to do it without nesting */
thirdword=word("John Q. Public",3)
say left(thirdword,2)

/* And here is how to do it with nesting */
say left(word("John Q. Public",3),2)
```

Program Control

REXX has instructions such as DO, IF, and SELECT to control your program. Here is a typical REXX IF instruction:

```
if a>1 & b<0 then do
say "Whoops, A is greater than 1 while B is less than 0!"
say "I'm ending with a return code of 99."
exit 99
end
```

The REXX relational operator for a logical AND is different from the operator in C, which is &&. Other relational operators differ as well, so you may want to review the appropriate section in the *Object REXX for Linux: Reference*.

Here is a list of some REXX comparison operators and operations:

= True if the terms are equal (numerically, when padded, and so on)

\=, ¬= 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 equal to

<= Less than or equal to

== True if terms are strictly equal (identical)

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

Note: Throughout the language, the NOT character, ¬, is synonymous with the backslash (\). You can use both characters. The backslash can appear in the \ (prefix not), \=, and \== operators.

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.

Note: On ASCII systems, REXX recognizes the ASCII character encoding 124 as the logical OR character. Depending on the code page or keyboard you are using for your particular country, the logical OR character is shown as a solid vertical bar (|) or a split vertical bar (¦). The appearance of the character on your 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, make sure this character is ASCII character encoding 124.

Using the wrong relational or comparison operator is a common mistake when switching between C and REXX. The familiar C language braces { } are not used in REXX for blocks of instructions. Instead, REXX uses DO/END pairs. The THEN keyword is always required.

Here is an IF instruction with an ELSE:

```
if a>1 & b<0 then do
    say "Whoops, A is greater than 1 while B is less than 0!"
    say "I'm ending with a return code of 99."
    exit 99
end
else do
    say "A and B are okay."
    say "On with the rest of the program."
end  /* if */
```

You can omit the DO/END pairs if only one clause follows the THEN or ELSE keyword:

```
if words(myvar) > 5 then
    say "Variable MYVAR has more than five words."
else
    say "Variable MYVAR has fewer than six words."
```

REXX also supports an ELSE IF construction:

```
count=words(myvar)
if count > 5 then
    say "Variable MYVAR has more than five words."
else if count >3 then
    say "Variable MYVAR has more than three, but fewer than six words."
else
    say "Variable MYVAR has fewer than four words."
```

The SELECT instruction in REXX is similar to the SELECT CASE statement in Basic and the *switch* statement in C. SELECT executes a block of statements based on the value of an expression. REXX's SELECT differs from the equivalent statements in Basic and C in that the SELECT keyword is not followed by an expression. Instead, expressions are placed in WHEN clauses:

```
select
when name='Bob' then
    say "It's Bob!"
when name='Mary' then
    say "Hello, Mary."
otherwise
end /* select */
```

WHEN clauses are evaluated sequentially. When one of the expressions is true, the statement, or block of statements, is executed. All the other blocks are skipped, even if their WHEN clauses would have evaluated to true. Notice that statements like the *break* statement in C are not needed.

The OTHERWISE keyword is used without an instruction following it. REXX does not require an OTHERWISE clause. However, if none of the WHEN clauses evaluates to true and you omit OTHERWISE, an error occurs. Therefore, always include an OTHERWISE.

As with the IF instruction, you can use DO/END pairs for several clauses within SELECT cases. You do not need a DO/END pair if several clauses follow the OTHERWISE keyword:

```
select
when name='Bob' then
    say "It's Bob"
when name='Mary' then do
    say "Hello Mary"
    marycount=marycount+1
    end
```

```
otherwise
    say "I'm sorry.  I don't know you."
    anonymous=anonymous+1
end /* select */
```

Many Basic implementations have several different instructions for loops. REXX only knows the DO/END pair. All of the traditional looping variations are incorporated into the DO instruction:

```
do i=1 to 10         /* Simple loop            */
   say i
end

do i=1 to 10 by 2    /* Increment count by two */
   say i
end

b=3; a=0             /* DO WHILE - the conditional expression  */
do while a<b         /* is evaluated before the instructions   */
   say a             /* in the loop are executed.  If the      */
   a=a+1             /* expression isn't true at the outset,   */
end                  /* instructions are not executed at all.  */

a=5                  /* DO UNTIL - like many other languages,  */
b=4                  /* a REXX DO UNTIL block is executed at   */
do until a>b         /* least once.  The expression is         */
   say "Until loop"  /* evaluated at the end of the loop.      */
end
```

REXX also has a FOREVER keyword. Use the LEAVE, RETURN, or EXIT instructions to break out of the loop:

```
                 /* Program to emulate your five-year-old child */
num=random(1,10)  /* To emulate a three-year-old, move this inside the loop! */
do forever
  say "What number from 1 to 10 am I thinking of?"
  pull guess
  if guess=num then do
     say "That's correct"
     leave
  end
  say "No, guess again..."
end
```

REXX also includes an ITERATE instruction, which skips the rest of the instructions in that iteration of the loop:

```
do i=1 to 100
   /* Iterate when the 'special case' value is reached    */
   if i=5 then iterate

   /* Instructions used for all other cases would be here */

end
```

You can use loops in IF or SELECT instructions:

```
/* Say hello ten times if I is equal to 1 */
if i=1 then
   do j=1 to 10
      say "Hello!"
   end
```

There is an equivalent to the Basic GOTO statement: the REXX SIGNAL instruction. SIGNAL causes control to branch to a label:

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

As with GOTO, you need to be careful about how you use SIGNAL. In particular, do not place SIGNAL in the middle of a DO/END block or into a SELECT structure.

Subroutines and Procedures

In REXX you can write routines that make all variables accessible to the called routine. You can also write routines that hide the caller's variables.

The following shows an example of a routine in which all variables are accessible:

```
/* Routine example                    */
i=10               /* Initialize I    */
call myroutine     /* Call routine    */
say i              /* Displays 22     */
exit               /* End main program */

myroutine:         /* Label           */
i=i+12             /* Increment I     */
return
```

The CALL instruction calls routine MYROUTINE. A label (note the colon) marks the start of the routine. A RETURN instruction ends the routine. Notice that an EXIT instruction is required in this case to end the main program. If EXIT is omitted, REXX assumes that the following instructions are part of your main program and will execute those instructions. The SAY instruction displays 22 instead of 10 because the caller's variables are accessible to the routine.

You can return a result to the caller by placing an expression in the RETURN instruction, like this:

```
/* Routine with result                */
i=10               /* Initialize I     */
call myroutine     /* Call routine     */
say result         /* Displays 22      */
exit               /* End main program */

myroutine:         /* Label            */
return i+12        /* Increment I      */
```

The returned result is available to the caller in the special variable RESULT, as previously shown. If your routine returns a result, you can call it as a function:

```
/* Routine with result called as function  */
i=10               /* Initialize I         */
say myroutine()    /* Displays 22          */
exit               /* End main program     */

myroutine:         /* Label                */
return i+12        /* Increment I          */
```

You can pass arguments to this sort of routine, but all variables are available to the routine anyway.

You can also write routines that separate the caller's variables from the routine's variables. They eliminate the risk of accidentally writing over a variable used by the caller or by some other unprotected routine. To get protection, use the PROCEDURE instruction, as follows:

```
/* Routine example using PROCEDURE instruction                          */
headcount=0
tailcount=0
/* Toss a coin 100 times, report results */
do i=1 to 100
   call cointoss                                      /* Flip the coin      */
   if result='HEADS' then headcount=headcount+1       /* Increment counters */
   else tailcount=tailcount+1
                                                      /* Report results     */
say "Toss is" result ||".  Heads=" headcount  "Tails=" tailcount
end /* do */
exit                                                  /* End main program   */

cointoss: procedure             /* Use PROCEDURE to protect caller         */
   i=random(1,2)                /* Pick a random number: 1 or 2            */
   if i=1 then return "HEADS"   /* Return English string                   */
return "TAILS"
```

In this example, the variable i is used in both the main program and the routine. When the PROCEDURE instruction is placed after the routine label, the routine's variables become local variables. They are isolated from all other variables in the program. Without the PROCEDURE instruction, the program would loop indefinitely. On each iteration the value of i would be reset to some value less than 100, which means the loop would never end. If a

programming error causes your procedure to loop indefinitely, use Ctrl+Break or close the Linux session to end the procedure.

To access variables outside the routine, add an EXPOSE operand to the PROCEDURE instruction. List the desired variables after the EXPOSE keyword:

```
/* Routine example using PROCEDURE instruction with EXPOSE operand        */
headcount=0
tailcount=0
/* Toss a coin 100 times, report results                                  */
do i=1 to 100
   call cointoss                                    /* Flip the coin      */
   say "Toss is" result ||".  Heads=" headcount  "Tails=" tailcount
end /* do */
exit                                                /* End main program   */

cointoss: procedure expose headcount tailcount /* Expose the counter variables */
   if random(1,2)=1 then do                    /* Pick a random number: 1 or 2 */
      headcount=headcount+1                    /* Bump counter...              */
      return "HEADS"                           /* ...and return English string */
   end
   else
      tailcount=tailcount+1
return "TAILS"
```

To pass arguments to a routine, separate the arguments with commas:

```
call myroutine arg1, "literal arg", arg3   /* Call as subroutine */
myrc=myroutine(arg1, "literal arg", arg3)  /* Call as function   */
```

In the routine, use the PARSE ARG instruction to retrieve the argument.

Chapter 3. Into the Object World

Object REXX includes features typical of an object-oriented language—features like subclassing, polymorphism, and data encapsulation. Object REXX is an *extension* of the traditional REXX language, which has been expanded to include classes, objects, and methods. These extensions do not replace traditional REXX functions, or preclude the development or running of traditional REXX programs. You can program as before, program with objects, or mix objects with regular REXX instructions. The REXX programming concepts that support the object-oriented features are described in this chapter.

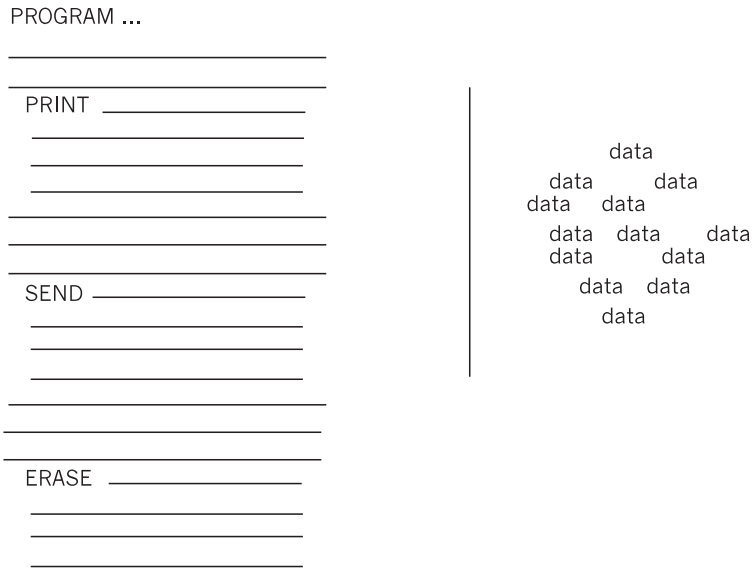
What Is Object-Oriented Programming?

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.

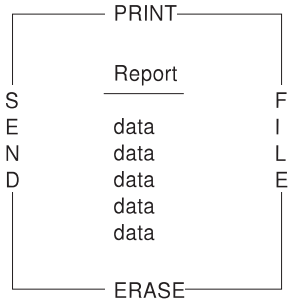


Figure 2. Modular Data—a Report Object

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:

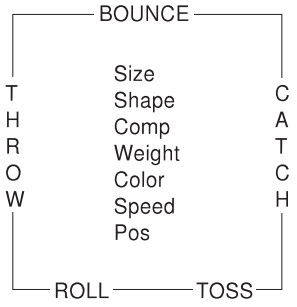


Figure 3. A Ball Object

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:

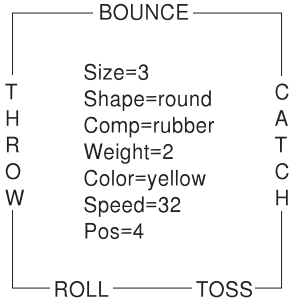


Figure 4. Ball Object with Variable Names and Values

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:

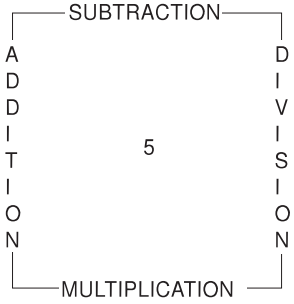


Figure 5. Encapsulated 5 Object

REXX comes with a basic set of classes for creating objects (see "Chapter 4. The Basics of Classes" on page 31). 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:

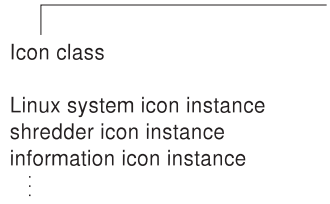


Figure 6. A Simple Class

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 Linux system icon, it might be position='20,20', while for the shredder it is '20,30' and for information it is '20,40':

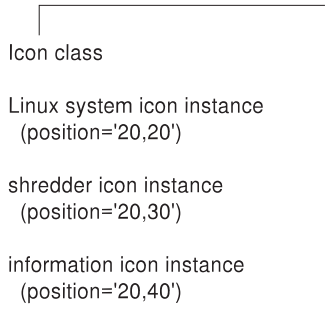


Figure 7. Icon Class

Objects that belong to a class are called *instances* of that class. As instances of the Icon class, the Linux 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.

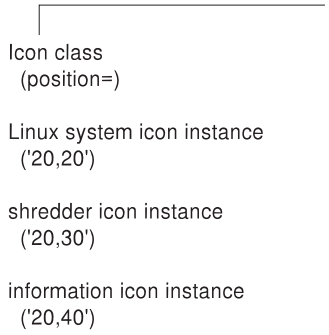


Figure 8. Instances of the Icon Class

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.

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):

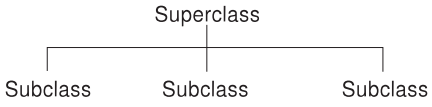


Figure 9. Superclass and Subclasses

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

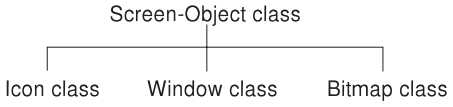


Figure 10. The Screen-Object Superclass

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*:

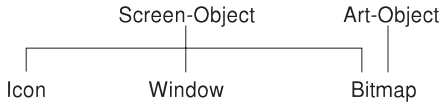


Figure 11. Multiple Inheritance

Chapter 4. The Basics of Classes

Similar objects in REXX are grouped into classes, forming a hierarchy. REXX gives you a basic class hierarchy to start with. All of the classes in the hierarchy are described in detail in the *Object REXX for Linux: Reference*. The following list shows the classes REXX provides (there may be others in the system). The classes indented are subclasses. Classes in the list that are described later in this chapter are printed in **bold**:

```
Object

    Alarm
    Class
    Collection classes
        Array
        List
        Queue
        Table
            Set
        Directory
        Relation
            Bag
    Message
    Method
    Monitor
    Stem
    Stream
    String
    Supplier
```

REXX Classes for Programming

The classes REXX supplies provide the starting point for object-oriented programming. Some key classes that you are likely to work with are described in the following sections.

The Alarm Class

The *Alarm class* is used to create objects with timing and notification capability. An alarm object is able to send a message to an object at any time in the future, and until then, you can cancel the alarm.

The Collection Classes

The *collection classes* are used to manipulate collections of objects. A *collection* is an object that contains a number of *items*, which can be any objects. These manipulations might include counting objects, organizing them, or assigning

them a supplier (for example, to indicate that a specific assortment of baked goods is supplied by the Pie-by-Night Bakery).

REXX includes classes, for example, for arrays, lists, queues, tables, and directories. Each 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 [] (left and right bracket) methods, and each collection defines its own acceptable index types:

Array A sequenced collection of objects ordered by whole-number indexes.

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.

Queue
A sequenced collection of 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.

Table A collection of 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 an index match by using the == comparison method to test for strict equality. A table contains no duplicate indexes.

Directory
A collection of character string indexes. Indexes are compared using the string == comparison method to test for strict equality.

Relation
A collection of indexes that can be any object (as with the Table class). A relation can contain duplicate indexes.

Set A collection where the indexes are equal to the values. Set indexes can be any object (as with the Table class) and each index is unique.

Bag A collection where the index is equal to the value. Bag indexes can be any object (as with the Table class) and each index can appear more than once.

The Message Class

To manipulate message objects, you use the *Message class*. Methods created for this class are used, for example, to send a message, to notify the sender object when an error occurs or when message processing is complete, to return the

results of that processing to the sender or to some other object, or to process the message object concurrently with the sender object.

The Monitor Class

The *Monitor class* provides a way to forward messages to a specified destination. Monitor methods let you initialize a monitor object, specify a destination object or use a previously specified one, and obtain the name of the current destination object.

The Stem Class

A stem is a symbol that must start with a letter and end with a period, like "FRED." or "A.". The value of a stem is a stem object by default. A stem object is a collection of unique indexes that are character strings. Stem objects are automatically created when a REXX stem variable or REXX compound variable is used. In addition to the items assigned to the collection indexes, a stem object also has a default value that is used for all uninitialized indexes of the collection. You can assign a default value to a stem object and later retrieve this value.

The Stream Class

Input and output streams let REXX communicate with external objects, such as people, files, queues, serial interfaces, displays, and networks. In programming there are many stream actions that can be coded as methods for manipulating the various stream objects. These methods and objects are organized in the *Stream class.*

The methods are used to open streams for reading or writing, close streams at the end of an operation, move the line-read or line-write position within a file stream, or get information about a stream. Methods are also provided to get character strings from a stream or send them to a stream, count characters in a stream, flush buffered data to a stream, query path specifications, time stamps, size, and other information from a stream, or do any other I/O stream manipulation (see "Chapter 7. Input and Output" on page 83 for examples).

The String Class

Strings are data values that can have any length and contain any characters. They are subject to logical operations like AND, OR, exclusive OR, and logical NOT. Strings can be concatenated, copied, reversed, joined, and split. When strings are numeric, there is the need to perform arithmetic operations on them or find their absolute value or convert them from binary to hexadecimal, and vice versa. All this and more can be accomplished using the *String class* of objects.

The Supplier Class

Some collections have suppliers: a bakery, for example, can supply certain breads, cookies, cakes and pies; a financial report can supply certain data, statistics, tables, and reports. The *Supplier class* is used to enumerate items that a collection contained when the supplier was created. For example, this class contains methods to verify if an item is available from a supplier (Does the Pie-by-Night Bakery sell chocolate cake?). Another method returns the index of the current item in a collection (What is the position of the apple pie record?). Others return the current collection item in a collection object, and the next item in the collection.

REXX Classes for Organizing Objects

REXX provides several key classes that form the basis for building class hierarchies.

The Object Class

Because the topmost class in the hierarchy, or the root class, is the *Object class,* everything below it is an object. To interact with each other, objects require their own actions, called methods. These methods that encode actions needed by all objects belong to the Object class.

Every other class in the hierarchy inherits the methods of the root class. *Inheritance* is the handing down of methods from a "parent" class—called a *superclass*—to all of its "descendent" classes—called *subclasses.* Finally, instances *acquire* methods from their classes. Any method created for the Object class is automatically made available to every other class in the hierarchy.

The Class Class

The *Class class* is used for generating new classes. If a class is like a factory for producing *instances,* Class is like a factory for producing *factories.* Class is the parent of every new class in the hierarchy, and these all inherit Class-like characteristics. Class-like characteristics are methods and related variables, which reside in Class, to be used by all classes.

A class that can be used to create another class is called a *metaclass.* The Class class is unique among REXX classes in that it is the only metaclass that REXX provides (see "Metaclasses" on page 45). As such, the Class's methods not only make new classes, they make methods for use by the new class and its instances. They also make methods that only the new class itself can use, but not its instances. These are called *class methods.* They give a new class some power that is denied to its instances.

Because each instance of Class is another class, that class inherits the Class's instance methods as class methods. Thus if Class generates a Pizza factory instance, the factory-running actions (Class's *instance* methods) become the *class* methods of the Pizza factory. Factory operations are class methods, and any new methods created to manipulate pizzas would be instance methods:

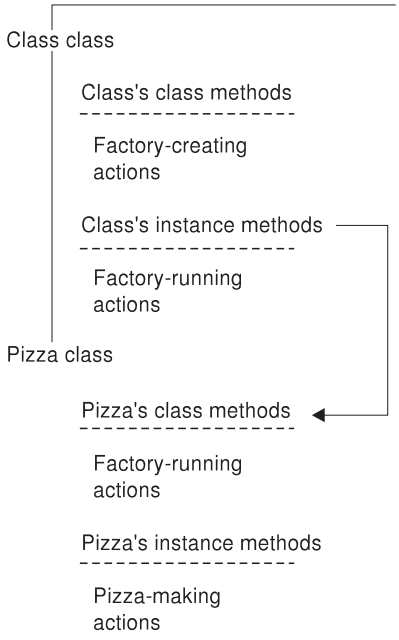
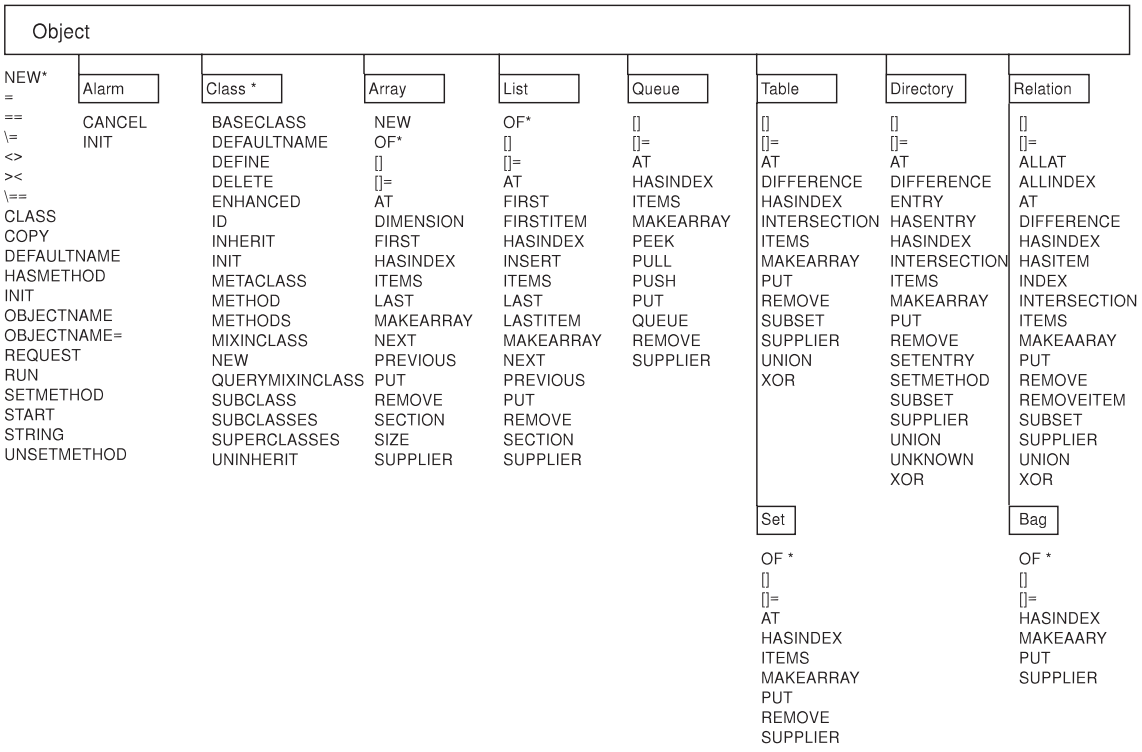


Figure 12. How Subclasses Inherit Instance Methods from the Class Class

As a programmer, you typically create classes by using *directives,* rather than the methods of the Class class. In particular, you'll use the ::CLASS directive, described later in this section. The ::CLASS directive is a kind of REXX clause that allows class definitions to be saved permanently, in a file, where they can be reused by other programs. Creating classes by using Class methods sent as messages is not recommended if permanency or reuse is required. At any rate, directives have class-creating powers similar to the Class methods.

REXX Classes: The Big Picture

The following figure diagrams the supplied REXX classes, along with their methods.



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

Figure 13. Classes and the Inheritance of Methods (Part 1 of 2)

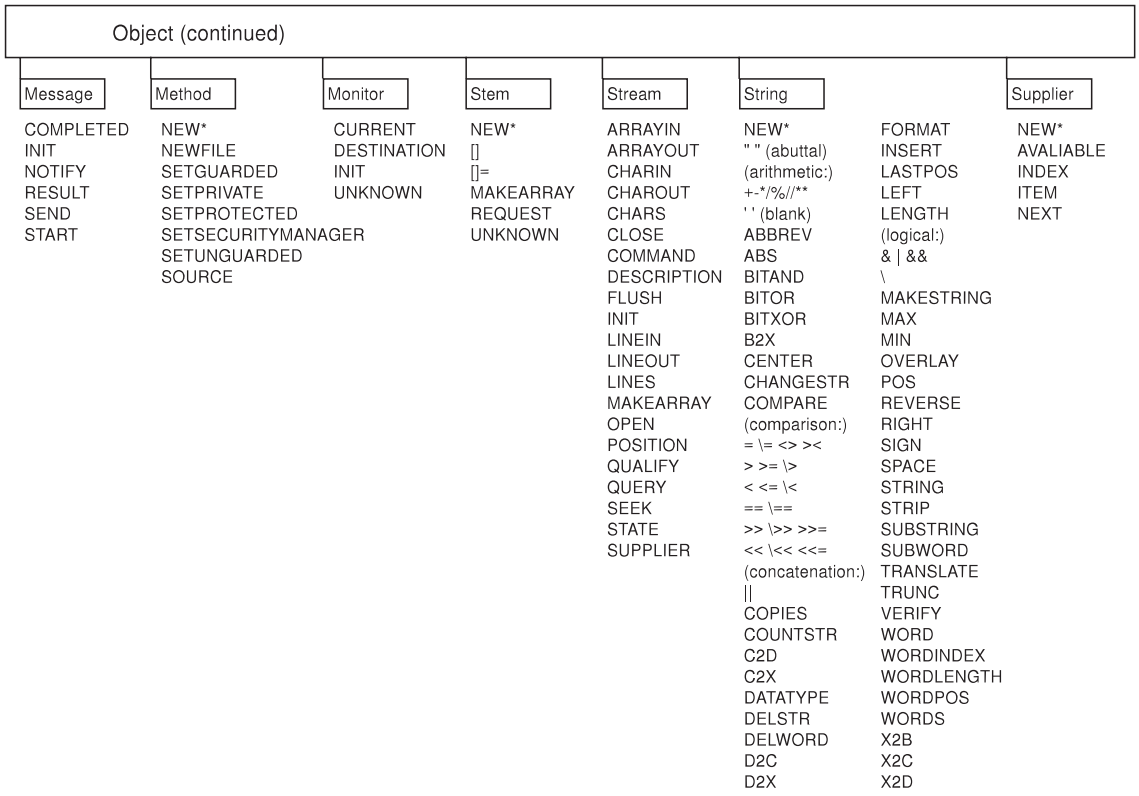


Figure 13. Classes and the Inheritance of Methods (Part 2 of 2)

Creating Your Own Classes Using Directives

By analyzing your problem in terms of objects, you can determine what classes need to be created. You can create a class using messages or directives. Directives are a new kind of REXX clause, and they are preferred over messages because the code is easier to read and understand, especially in large programs. They also provide an easy way for you to save your class definitions and share them with others using the PUBLIC option.

What Are Directives?

A REXX program is made up of one or more executable units. *Directives* separate these units, which themselves are REXX programs. REXX processes all directives first to set up any classes, methods, or routines needed by the program. Then it runs any code that exists before the first directive. The first directive in a program marks the end of the executable part of the program. A

directive is a kind of clause that begins with a double-colon (::) and is non-executable (a directive cannot appear in the expression of an INTERPRET instruction, for example).

The Directives REXX Provides

The following is a short summary of all the REXX directives. See the *Object REXX for Linux: Reference* for more details on, or examples of, any of these REXX directives.

The ::CLASS Directive

You use the ::CLASS directive to create a class. Programs can then use the new class by specifying it as a REXX environment symbol (the class name preceded by a period) in the program. For example, in "A Sample Program Using Directives" on page 40, the Savings class is created using the ::CLASS directive. A program can then use the new class by specifying it as an environment symbol, ".savings".

The new class that you create acquires any methods defined by subsequent ::METHOD directives within the program, until either another ::CLASS directive or the end of the program is reached.

You can use the ::CLASS directive's SUBCLASS option to make the new class the subclass of another. In "A Sample Program Using Directives" on page 40, the Savings class is made a subclass of the Account class. A subclass inherits instance and class methods from its specified superclass; in the sample, Savings inherits from Account.

Additional ::CLASS directive options are available for:

- Inheriting *instance* methods from a specified metaclass as *class* methods of the new class (the METACLASS option). For more information on metaclasses, see "Metaclasses" on page 45.
- Making the new class available to programs outside its containing REXX program (the PUBLIC option). The outside program must refer to the new class by using a ::REQUIRES directive.
- Subclassing the new class to a mixin class in order to inherit its instance and class methods (the MIXINCLASS option).
- Adding the instance and class methods of a mixin class to the new class, without subclassing it (the INHERIT option).

When you create a new class, it is always a subclass of an existing class. If you do not specify the SUBCLASS or MIXINCLASS option on the ::CLASS directive, the superclass for the new class is the Object class.

Your class definition can be in a file of its own, with no executable code preceding it. For example, when you define classes and methods to be shared by several programs, you put the executable code in another file and refer to the class file using a ::REQUIRES directive.

REXX processes ::CLASS directives in the order in which they appear, unless there is a dependency on some later directive's processing. You cannot create two classes that have the same class name in one program. If several programs contain classes with the same name, the last ::CLASS directive processed is used.

The ::METHOD Directive

The ::CLASS directive is usually followed by a ::METHOD directive, which is used to create a method for that class and define the method's attributes. The next directive in the program, or the end of the program, ends the method.

Some classes you define have an INIT method. INIT is called whenever a NEW message is sent to a class. The INIT method must contain whatever code is needed to initialize the object.

It is not required that a ::METHOD directive be preceded by a ::CLASS directive. However, without it the method is only accessible by the executable part of the program through REXX's .METHODS environment symbol. This symbol identifies a directory of methods that you can refer to by name. For each method name only one method directive can appear that is not associated with a class.

The ::METHOD directive can be used for:

- Creating a *class* method for the most-recent ::CLASS directive (the CLASS option).
- Creating a *private* method; that is, a method that works like a subroutine and can only be activated by the object it belongs to—otherwise the method is public by default, and any sender can activate it.
- Creating a method that can be called while other methods are active on the same object, as described in "Activating Methods" on page 69 (the UNGUARDED option).
- Creating the instance methods *method_name* and *method_name*= for the preceding ::CLASS directive (the ATTRIBUTE option).

The ::ROUTINE Directive

You use the ::ROUTINE directive to create a named routine within a program. The ::ROUTINE directive starts the named routine and another directive (or the end of the program) ends the routine.

The ::ROUTINE directive is useful for defining lower-level routines that are called by several methods. These methods might be in unrelated classes or in different applications. You use ::ROUTINE when you have a utility that you do not want to appear as a method.

The ::ROUTINE directive includes a PUBLIC option for making the routine available to programs outside its containing REXX program. The outside program must reference the routine by using a ::REQUIRES directive.

Only one ::ROUTINE directive can appear for a routine name within a program.

The ::REQUIRES Directive

You use the ::REQUIRES directive when a program needs access to the classes and objects of another program. This directive has the following form:

```
::REQUIRES program_name
```

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

Local routine or class definitions within a program override routines or classes of the same name in programs that are accessed through ::REQUIRES directives. Another directive (or the end of the program) must follow a ::REQUIRES directive.

How Directives Are Processed

You place a directive (and its method code) *after* the program code. When you run a program containing directives, REXX:

1. Processes the directives first, to set up the program's classes, methods, and routines.
2. Runs any program code preceding the first directive. This code can use any classes, methods, and routines set up by the directives.

   Once REXX has processed the code preceding the directive, any public classes and objects the program defines are available to programs having the appropriate ::REQUIRES directive.

A Sample Program Using Directives

Here is a program that uses directives to create new classes and methods:

```
asav = .savings~new             /* executable code begins */
say asav~type                   /* executable code        */
asav~name= 'John Smith'         /* executable code ends   */

::class Account                 /* directives begin ...   */
```

```
  ::method 'TYPE'
    return "an account"

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

::class Savings subclass Account

  ::method 'TYPE'
    return "a savings account"   /* ... directives end     */
```

The preceding program uses the ::CLASS directive to create two classes, the Account class and its Savings subclass. In the `::class Account` expression, the ::CLASS directive precedes the name of the new class, Account.

The example program also uses the ::METHOD directive to create TYPE and NAME= methods for Account. In the `::method 'TYPE'` expression, the ::METHOD directive precedes the method name, and is immediately followed by the code for the method. Methods for any new class follow its ::CLASS directive in the program, and precede the next ::CLASS directive.

In the `::method 'NAME='` method, the USE ARG instruction retrieves the argument. The EXPOSE instruction, which must immediately follow the ::METHOD directive, makes the value (here, "John Smith") available for use by other methods. A variable in an EXPOSE instruction is called an *object variable.*

You do not have to associate object variables with a specific object. REXX keeps track of object variables for you. Whenever you send a message to savings account Asav, which points to the Name object, REXX knows what internal object value to use. If you assign another value to Asav (such as "Mary Smith"), REXX deletes the object that was associated with Asav ("John Smith") as part of its normal garbage-collection operations.

In the Savings subclass, a second TYPE method is created that supersedes the TYPE method Savings would otherwise have inherited from Account. Note that the directives appear after the program code.

Another Sample Program

A directive is nonexecutable code that begins with a double colon (::) and follows the program code. The ::CLASS directive creates a class; in this example, the Dinosaur class. The sample provides two methods for the Dinosaur class, INIT and DIET. These are added to the Dinosaur class using the ::METHOD directives. After the line containing the ::METHOD directive,

the code for the method is specified. Methods are ended either by the start of the next directive or by the end of the program.

Because directives must follow the executable code in your program, you put that code first. In this case, the executable code creates a new dinosaur, Dino, that is an instance of the Dinosaur class. REXX then runs the INIT method. REXX runs any INIT method automatically whenever the NEW message is received. Here the INIT method is used to identify the type of dinosaur. Then the program runs the DIET method to determine whether the dinosaur eats meat or vegetables. REXX saves the information returned by INIT and DIET as variables in the Dino object.

In the example, the Dinosaur class and its two methods are defined following the executable program code:

```
dino=.dinosaur~new         /* Create a new dinosaur instance and
                                        /* initialize variables */
dino~diet                  /* Run the DIET method          */
exit
::class Dinosaur           /* Create the Dinosaur class  */

  ::method init            /* Create the INIT method     */
    expose type
    say "Enter a type of dinosaur."
    pull type
    return

  ::method diet            /* Create the DIET method     */
    expose type
    select
    when type="T-REX" then string="Meat-eater"
    when type="TYRANNOSAUR" then string="Meat-eater"
    when type="TYRANNOSAURUS REX" then string="Meat-eater"
    when type="DILOPHOSAUR" then string="Meat-eater"
    when type="VELICORAPTOR" then string="Meat-eater"
    when type="RAPTOR" then string="Meat-eater"
    when type="ALLOSAUR" then string="Meat-eater"
    when type="BRONTOSAUR" then string="Plant-eater"
    when type="BRACHIOSAUR" then string="Plant-eater"
    when type="STEGOSAUR" then string="Plant-eater"
    otherwise string="Type of dinosaur or diet unknown"
    end
    say string
    return 0
```

Creating Classes Using Messages

You can create a class using messages as well as directives. Though classes are available only to the program that creates them, there are occasions when this is useful and public availability is not required. The following sections demonstrate the message technique using the Savings Account example previously shown with directives.

Defining a New Class

To define a new class using messages, you send a SUBCLASS message to the new class's superclass. That is, you send the message to the class that precedes the new class in the hierarchy. To define a subclass of the Object class called Account, you enter:

```
account = .object~subclass('Account')
```

Here, `.object` is a reference to the REXX Object class. `.object` is an environment symbol indicating the intention to create a new class that is a subclass of the Object class. Environment symbols represent objects in a directory of public objects, called the *Environment object.* These public objects are available to all other objects, and include all the classes that REXX provides. Environment symbols begin with a period and are followed by the class name. Thus the Object class is represented by `.object`, the Alarm class by `.alarm`, the Array class by `.array`, and so on.

The twiddle (~) is the "message send" symbol, `subclass` is a method of Class, and the string identifier in parentheses is an argument of SUBCLASS that names the new class, Account.

Adding a Method to a Class

You use the DEFINE method to define methods for your new class. To define a TYPE method and a NAME= method, you enter:

```
account~define('TYPE', 'return "an account"')
account~define('NAME=', 'expose name; use arg name')
```

Defining a Subclass of the New Class

Using the SUBCLASS method, you can define a subclass for your new class and then a method for that subclass. To define a Savings subclass for the Account class, and a TYPE method for Savings, you enter:

```
savings = account~subclass('Savings Account')
savings~define('TYPE', 'return "a savings account"')
```

Defining an Instance

You use the NEW method to define an instance of the new class, and then call methods that the instance inherited from its superclass. To define an instance of the Savings class named "John Smith," and send John Smith the TYPE and NAME= messages to call the related methods, you enter:

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

Types of Classes

In REXX there are three class types:

- Object classes
- Abstract classes
- Mixin classes

An *object class* (the default) can create instances of the object in response to receiving a NEW or ENHANCED message. An *abstract class* serves mainly to organize other classes in the hierarchy and define their message interface. A *mixin class*, through multiple inheritance, is an additional superclass to a class. The mixin class typically possesses methods useful to the class that inherits it, but these must be *specifically added* because they lie outside the class's normal line of inheritance.

The following sections explain these class types in more detail.

Object Classes

An *object class* creates instances and provides methods that these instances can use. At the time of its creation, an instance acquires all instance methods of the class it belongs to. If a class adds new methods later, existing instances do not acquire them. Instances created after the new methods do acquire them.

Because classes define methods for their instances, and methods define the variables that instances use, object classes are factories for creating REXX instances. The Array class is an example of an object class.

Abstract Classes

An *abstract class* defines methods its subclasses can inherit, but typically has no instances. Rather, it serves to organize other classes in the hierarchy. An abstract class can be used to "filter out" a group of shared methods from a number of subclasses, so they do not have to exist in two places.

An abstract class pushes common elements further up the hierarchy, thus providing a higher level of organization. By filtering out and moving common methods upwards, the abstract class refines the message interface for its subclasses. This lays the groundwork for polymorphism, creating well-defined interfaces for users of the hierarchy. Abstract classes inherit the instance methods of the Class class.

You can create a new abstract class like an object class. You use a simple ::CLASS directive; no options are required. While abstract classes are not intended for creating instances, REXX does not prevent you from doing so.

Mixin Classes

The *mixin class* lets you optionally add a set of instance and class methods to one or more other classes using inheritance. You use mixins to extend the scope of a class beyond the usual lines of inheritance defined by the hierarchy. This is like widening a class's inheritance to accept methods from a sibling or cousin, as well as a parent. When a class inherits from more than just its parent superclass, it is called *multiple inheritance.*

You can add mixin methods to a class by using the INHERIT option on the ::CLASS directive. The class to be inherited must be a mixin class. During class creation and multiple inheritance, subclasses inherit both class and instance methods from their superclasses.

A mixin's first non-mixin superclass is its *base class.* Any subclass of a mixin's base class can directly or indirectly inherit a mixin; other classes cannot.

To create a new mixin class, you use the ::CLASS directive with the MIXINCLASS option. A mixin class is also an object class and can create its own instances.

Metaclasses

A *metaclass* is a class you can use to create another class. REXX provides just one metaclass, the Class class. A class is a factory for creating *instances,* and the Class class is a factory for creating *factories.* Whenever you create a new factory, or class, the new class is an instance of Class. The *instance* methods of Class provide the operations needed to run the new factories. These instance methods are inherited by the new factory as its *class* methods.

The classes REXX provides do not permit changes or additions to their method definitions. As a result, all new factories inherit these unchangeable actions from the Class class, and thus operate the same way. So if you want to create a new class—a new *factory*—that behaves differently from the others, you can do either of the following:

- Write additional class methods for the new class, using the ::METHOD directive with the CLASS option
- Use a metaclass

If you plan to create many factories with the same operational changes, you use the metaclass.

Any metaclass you create is a subclass of the Class class. To make your own metaclass, specify `class` as a SUBCLASS option in the ::CLASS directive:

```
/* Create a new metaclass  */
::class your_metaclass subclass class
```

The instance methods of *your_metaclass* becomes the class methods for any new class created using *your_metaclass.* For example, you could create a metaclass called InstanceCounter that includes instance methods for tracking how many instances the class creates:

```
/* Create a new metaclass that counts its instances        */
::class InstanceCounter subclass class
  ::method init

   ⋮
```

Instead of having to add instance-counting class methods to other new classes you write, you can make InstanceCounter their metaclass. When you create the new class, you specify InstanceCounter as a METACLASS option in the ::CLASS directive. Creating a Point class might look like this:

```
/* Create a public Point class using the InstanceCounter metaclass  */
::class point public metaclass InstanceCounter
  ::method init

   ⋮
```

The instance methods in your new InstanceCounter metaclass become the class methods of the Point class, and any other classes that you create in the future using a similar directive. Here is a complete example:

```
/* A metaclass example */

a = .point~new(1,1)                   /* Create point instances       */
say 'Created point instance' a        /* a, b, and c.                 */
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.'

                                      /* Create a new metaclass that  */
```

```
                                    /* counts its instances.        */
::class InstanceCounter subclass class
  ::method init                     /* Create an INIT method to     */
    expose instanceCount            /* initialize instanceCount.    */
    instanceCount = 0               /* Forward INIT to superclass.  */

    .message~new(self, .array~of('INIT',super), 'a', arg(1,'A'))~send

  ::method new                      /* Create a NEW instance method.*/
    expose instanceCount            /* Create a new instance.       */
    instanceCount = instanceCount + 1 /* Bump the count.            */

                                    /* Forward NEW to superclass.   */
    return .message~new(self, .array~of('NEW',super), 'a', arg(1,'A'))~send


  ::method instances                /* Create an INSTANCES method.  */
    expose instanceCount            /* Return the instance count.   */
    return instanceCount

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

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

Chapter 5. A Closer Look at Objects

This chapter covers the mechanics of using objects and methods in more detail. First, a quick refresher.

A REXX object consists of:

- Actions coded as *methods*
- Characteristics, coded as *variables,* and their values, sometimes referred to as "state data"

Sending a message to an object causes it to perform a related action. The method whose name matches the message performs the action. The message is the interface to the object, and with *information hiding*, only methods that belong to an object can access its variables.

Objects are grouped hierarchically into *classes.* The class at the top of the hierarchy is the Object class. Everything below it in the hierarchy belongs to the Object class and is therefore an object. As a result, all classes are objects.

In a class hierarchy, classes, superclasses, and subclasses are relative to one another. Unless designated otherwise, any class directly above a class in the hierarchy is a superclass. And any class below is a subclass.

From a class you can create *instances* of the class. *Instances* are merely similar objects that fit the template of the class; they belong to the class, but are not classes themselves. Instances are the most basic, most elemental of objects.

Both the classes and their instances contain variables and methods. The methods a class provides for use by its various instances are called *instance methods.* In effect, these define which messages an instance can respond to. Instance methods are the most common methods in REXX, and are therefore called *methods*.

The methods available to the class itself are called *class methods.* (They are actually the instance methods of the Class class.) They define messages that only the class—and not its instances—can respond to. Class methods generally exist to *create* instances and are much less common.

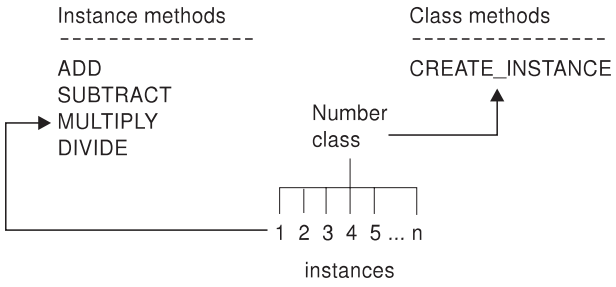


Figure 14. Instance Methods and Class Methods

Using Objects in REXX Clauses

The following examples with Myarray illustrate how to use new objects in REXX clauses.

```
myarray=.array~new(5)
```

creates a new instance of the Array class, Myarray. The period precedes a class name in an expression, to distinguish the class from other variables. The Myarray array object has five elements.

After Myarray is created, you can assign values to it. One way is with the PUT method. PUT has two arguments, which must be enclosed in parentheses. The first argument is the string to be placed in the element. The second is the number of the element in which to place the string. Here, the string object `Hello` is associated with the third element of Myarray:

```
myarray~put("Hello",3)
```

REXX dynamically adjusts the size of the array to accommodate the new element.

One way to retrieve values from an array object is by sending it an AT message. In the next example, the SAY instruction displays the third element of Myarray:

```
say myarray~at(3)

Results:

Hello
```

The SAY instruction expects the string object as input, which is what AT returns. If you put a nonstring object in the SAY instruction, SAY sends a STRING message to the object. The STRING method for REXX's built-in

objects returns a human-readable string representation for the object. In this example, the STRING method for Myarray returns the string `an array`:

```
say myarray  /* SAY sends STRING message to Myarray */

Results:

an array
```

Whenever a method returns a string, you can use it within expressions that require a string. Here, the element of the array that AT returns is a string, so you can put an expression containing the AT method inside a string function like COPIES():

```
say copies(myarray~at(3),4)

Results:
HelloHelloHelloHello
```

This example gets the same result using only methods:

```
say myarray~at(3)~copies(4)
```

Notice that the expression is evaluated from left to right. You can also use parentheses to enforce an order of evaluation.

Almost all messages are sent using the twiddle, but there are exceptions. The exceptions are to improve the reliability of the language. The following example uses the []= (left-bracket right-bracket equal-sign) and [] methods to set and retrieve array elements:

```
myarray[4]="the fourth element"
say myarray[4]
```

Although the previous instructions look like an ordinary array assignment and array reference, they are actually messages to the Myarray object. You can prove this by executing these equivalent instructions, which use the twiddle to send the messages:

```
myarray~"[]="("a new test",4)
say myarray~"[]"(4)
```

Similarly, expression operators (such as +, -, /, and *) are actually methods, but you do not have to use the twiddle to send them:

```
say 2+3      /* Displays 5 */
say 2~'+'(3) /* Displays 5 */
```

In the second SAY instruction, quotes are needed around the + message because it is a character not allowed in a REXX symbol.

When running your program, three methods that REXX looks for, and runs automatically when appropriate, are INIT, UNINIT, and STRING.

Initializing Instances Using INIT

Object classes can create instances. When these instances require initialization, you'll want to define an INIT method to set a particular starting value or initiate some startup processing. REXX looks for an INIT method whenever a new object is created and runs it.

The purpose of initialization is to ensure that the variable is set correctly before using it in an operation. If an INIT method is defined, REXX runs it after creating the instance. Any initialization arguments specified in the NEW or ENHANCED message are passed to the INIT method, which can use them to set the initial state of object variables.

If an instance has more than one INIT method (for example, INIT is defined in several classes), each INIT method must forward the INIT message up the hierarchy and run the topmost version of INIT, to properly initialize the instance. An example in the next section demonstrates the use of INIT.

Returning String Data Using STRING

The STRING method is a useful way to access object data and return it in string form for use by your program. When a SAY instruction is processed in REXX, REXX automatically sends a STRING message to the object specified in the expression. REXX uses the STRING method of the Object class and returns a human-readable string representation for the object. For example, if you instruct REXX to `say a`, and `a` is an array object, REXX returns an array. You can take advantage of this automatic use of STRING by overriding REXX's STRING method with your own, and extract the object data itself—in this case, the actual array data.

The following programs demonstrate STRING and INIT. In the first program, the Part class is created, and along with it, the two methods under discussion, STRING and INIT:

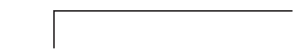
```
/* partdef.cmd - Class and method definition file */

/* Define the Part class as a public class */
::class part public

/* Define the INIT method to initialize object variables */
::method init
expose name description number
use arg name, description, number
```

```
/* Define the STRING method to return a string with the part name */
::method string
expose name
return "Part name:" name
```

In the ::CLASS directive, the keyword PUBLIC indicates that the class can be shared with other programs. The two ::METHOD directives define INIT and STRING. Whenever REXX creates a new instance of a class, it calls the INIT method for the class. The sample INIT method uses an EXPOSE instruction to make the name, description, and number variables available to other methods. These exposed variables are object variables, and are associated with a particular instance of a class:



```
Part class

part instance
  (name='Widget')
  (description='A small widge')
  (number=12345)

part instance
  (name='Framistat')
  (description='A device to control frams')
  (number=899)

part instance
  (name='Defragulator')
  (description='Removes frags from framistats')
  (number=01045)
```

Figure 15. Instances in the Part Class

INIT expects to be passed three arguments. The USE ARG instruction assigns these three arguments to the name, description, and number variables, respectively. Because those variables are exposed, the values are available to other methods.

The STRING method returns the string `Part name:`, followed by the name of a part. The STRING method does not expect any arguments. It uses the EXPOSE instruction to tap the object variables. The RETURN instruction returns the result string.

The following example shows how to use the Part class:

```
/* usepart.cmd  - use the Part class */
myparta=.part~new('Widget','A small widge',12345)
mypartb=.part~new('Framistat','Device to control frams',899)
```

```
say myparta
say mypartb
exit
::requires 'partdef'
```

The **usepart** program creates two parts, which are instances of the Part class. It then displays the names of the two parts.

REXX processes all directives before running your program. The ::REQUIRES directive indicates that the program needs access to public class definitions that are in another program. In this case, the ::REQUIRES directive refers to the **partdef** program, which contains the Part definition.

Note: Linux is case-sensitive. If you specify an external program name referring to a file in lowercase characters, REXX translates it into uppercase characters and searches for the program under its uppercase-character name. Therefore, always specify a program name that contains lowercase or mixed-case characters, in single quotes, for example 'myprogram'.

The assignment instructions for Mypart A and Mypart B create two objects that are instances of the Part class. The objects are created by sending a NEW message to the Part class. The NEW message causes the INIT method to be invoked as part of object creation. The INIT method takes the three arguments you provide and makes them part of the object's own exclusive set of variables, called a *variable pool.* Each object has its own variable pool (name, description, and number).

The SAY instruction sends a STRING message to the object. In the first SAY instruction, the STRING message is sent to MypartA. The STRING method accesses the Name object variable for MypartA and returns it as part of a string. In the second SAY instruction, the STRING message is sent again, but to a different object: MypartB. Because the STRING method is invoked for MypartB, it automatically accesses the variables for MypartB. You do not need to pass the name of the object to the method in order to distinguish different sets of object variables; REXX keeps track of them for you.

Another way to define classes is by using the SUBCLASS method. You can send a SUBCLASS method to any desired superclass to create a class.

Uninitializing and Deleting Instances Using UNINIT

Object classes can create instances but have no direct control over their deletion. If you assign a new value to a variable, REXX automatically reclaims the storage for the old value in a process called *garbage collection.*

If variables of other objects no longer reference an instance, REXX automatically reclaims that instance. If the instance has allocated other system resources, you must release them at this time using an UNINIT method. REXX cannot automatically release these resources because it is unaware that the instance has allocated them.

In the following example, the value passed to *text* is initialized by REXX using INIT and deleted by REXX using UNINIT. This program makes visible REXX's automatic invocation of INIT and UNINIT by revealing its processing on the screen using the SAY instruction:

```
/* uninit.cmd - example of UNINIT processing */

a=.scratchpad~new('Of all the things I've lost')
a=.scratchpad~new('I miss my mind the most')
say 'Exiting program.'
exit

::class scratchpad

  ::method init
    expose text
    use arg text
    say 'Remembering' text

  ::method uninit
    expose text
    say 'Forgetting' text
    return
```

Whether uninitialization processing is needed depends on the circumstances, for example when a message object holds an unreported error that should be reported and cleared. If an object requires uninitialization, define an UNINIT method to specify the processing you want.

If UNINIT is defined, REXX runs it before reclaiming the object's storage. If an instance has more than one UNINIT method (for example, UNINIT is defined in several classes), each UNINIT method is responsible for sending the UNINIT message up the hierarchy, using the SUPERCLASS overrides, to run the topmost version of UNINIT.

Special Variables

When writing methods, you can use special variables available in REXX. A special variable can be set automatically during the processing of a REXX program. REXX supports the following 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: Tracing interactively does not change the value of RC.

RESULT
is set by a RETURN instruction[1] in a subroutine that has been called, or a method that was activated by a message instruction, if the RETURN instruction specifies an expression. If the RETURN instruction has no expression on it, RESULT is dropped (becomes uninitialized).

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 regarding 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).

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.

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
```

1. RESULT is also set by an EXIT or REPLY instruction.

```
    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. For example:

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

This instruction 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. For example:

```
self~init:super
```

This instruction calls the INIT method the Object class defines.

You can alter these variables, just like any other variable, but REXX continues to set RC, RESULT, and SIGL automatically when appropriate. EXPOSE, PROCEDURE, USE, and DROP instructions affect these variables in their usual way.

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.

Public, Local, and Built-In Environment Objects

In addition to the special variables, REXX provides a unique set of objects, called *environment objects.* Environment objects are members of the Object class only. REXX makes the following environment objects available:

The Public Environment Object (.environment)

The Environment object is a directory of public objects that are always accessible throughout the whole process. To place something in the Environment directory, you use the form:

```
.environment~your.object =  value
```

Include a period (.) in any object name you use, to avoid conflicts with current or future REXX entries to the Environment directory. To retrieve your object, you use the form:

```
say .environment~your.object
```

The scope of `.environment` is the current process.

You use an *environment symbol* to access the entries of this directory. An environment symbol starts with a period and has at least one other character, which must not be a digit. You have seen environment symbols earlier; for example in:

```
asav = .savings~new
```

`.Savings` is an environment symbol, and refers to the Savings class. The classes you create can be referenced with an environment symbol. There is an environment symbol for each REXX-defined class, as well as for each of the unique objects this section describes, such as the NIL object.

The NIL Object (.nil)

The NIL object is a special environment object that does not contain any data. It represents the absence of an object, the way a null string represents a string with no characters. Its only methods are those of the Object class. You use the NIL object (rather than the null string) to test for the absence of data in an array entry:

```
if board[row,column] = .nil
then ...
```

All the environment objects REXX provides are single symbols. Use *compound symbols* when you create your own, to avoid conflicts with future REXX-defined entries.

The Local Environment Object (.local)

The Local environment object is a directory of process-specific objects that are always accessible. To place something in the Local environment directory, you use the form:

```
.local~your.object = value
```

Be sure to include a period (.) in any object name you use, to avoid conflicts with current or future REXX entries to the Local directory. To retrieve your object, you use the form:

```
say .local~your.object
```

The scope of `.local` is the current process.

You access objects in the Local environment object like in the Environment object. REXX provides the following objects in the Local environment:

.error is the Error object (the default error stream) to which REXX writes error messages and trace output to.

.input is the Input object (the default input stream), which is the source for the PARSE LINEIN instruction, the LINEIN method of the Stream class, and (if you do not specify a stream name) the LINEIN built-in function. It is also the source of the PULL and PARSE PULL instructions if the external data queue is empty.

.output
is the Output object (the default output stream), which is the destination of output from the SAY instruction, the LINEOUT method (.OUTPUT~LINEOUT), and (if you do not specify a 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.

Built-In Environment Objects

REXX provides environment objects that all programs can use. To access these *built-in objects,* you use the special environment symbols whose first character is a period (.).

.methods
The .methods environment symbol identifies a directory 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. 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'. Here is an 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: Tracing interactively does not change the value of .rs. The initial value of .rs is 0.

The Default Search Order for Environment Objects

When you use an environment symbol, REXX 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 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 program local environment directory, which 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.
4. The global environment directory, which includes all “permanent” REXX objects such as the REXX-supplied classes (for example, .ARRAY) and constants such as .TRUE and .FALSE. You can directly access the global environment by using the .environment symbol or using the VALUE built-in function with a null string for the *selector* argument.
5. REXX defined symbols. Other simple environment symbols are reserved for use by REXX for built-in objects. The currently defined built-in objects are .RS and .METHODS.

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

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

Example:

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

Determining the Scope of Methods and Variables

Methods interact with variables and their associated data. But a method cannot interact with any variable. Certain methods and variables are designed to work together. A method designates the variables it wants to work with by exposing them with an EXPOSE instruction. The exposed methods are called object variables. Exposing variables confines them to an object; in object-oriented terms, they are *encapsulated.* This protects the object variables' data from being changed by "unauthorized" methods belonging to other objects.

Objects with a Class Scope

Encapsulation usually takes place at the class level. The class is designed as a template of methods and variables. The instances themselves retain only the values of their variables.

Within the hierarchy, the class structure ensures the integrity of a class's variables, controlling the methods allowed to operate on them. The class structure also provides for easy updating of the method code. If a method requires a change, you only have to change it once, at the class level. The change then is acquired by all the instances sharing the method.

Associated methods and variables have a certain *scope,* which is the *class* to which they belong:

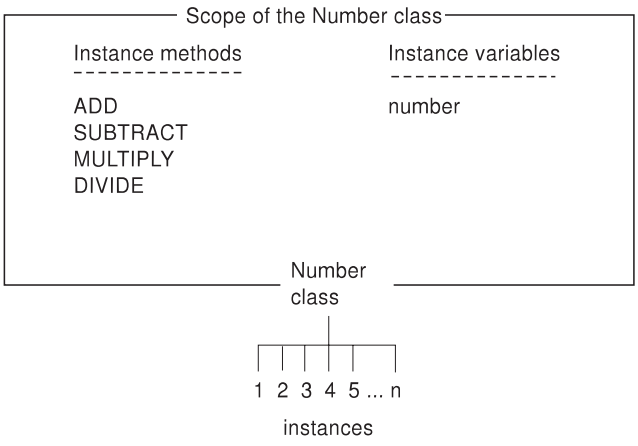


Figure 16. Scope of the Number Class

Each class in a class hierarchy has a scope different from any other class. This is what allows a variable in a subclass to have the same name as a variable in a superclass, even though the methods that use the variables may be completely different.

Objects with Their Own Unique Scope

The methods and variables used by instances in a class are usually found at the class level. But sometimes an instance differs in some respect from the others in its class. It might perform an additional action or require some unique handling. In this case one or more methods and related variables can be added directly to the instance. These additional methods and variables form a separate scope, independent of the class scopes found throughout the rest of the hierarchy.

Methods can be added directly to an instance's collection of object methods using SETMETHOD, a method of the Object class. All subclasses of the Object class inherit SETMETHOD. Alternately, the Class class provides an ENHANCED method that lets you create new instances of a class, whose object methods are the instance methods of its class, but enhanced with the additional collection methods.

More about Methods

A *method name* can be any character string. When an object receives a message, REXX searches for a method whose name matches the message name.

You must surround a method name with quotation marks when it is the same as an operator. The following example illustrates how to do this correctly. It

creates a new class (Cost), defines a new method (%), creates an instance of the Cost class (Mycost), and sends a % message to Mycost:

```
mycost = Cost~new           /* Creates new instance mycost.*/
mycost~'%'                  /* Sends % message to mycost.  */

::class Cost subclass 'Retail' /* Creates Cost, a sub-     */
                               /* class of 'Retail' class. */
  ::method "%"                 /* Creates % method.        */
    expose p                   /* Produces: Enter a price. */
    say "Enter a price"        /* If the user specifies a  */
    pull p                     /* price of 100,            */
    say p*1.07                 /* produces: 107            */
    return 0
```

The Default Search Order for Selecting a Method

When a message is sent to an object, REXX looks for a method whose name matches the message string. If the message is ADD, for example, REXX looks for a method named ADD. Because, in the class hierarchy, there may be more than one method with the same name, REXX begins its search at the object specified in the message. If the sought method is not found there, the search continues up the hierarchy. REXX searches in the following order:

1. A method the object defines itself (with SETMETHOD or ENHANCED).
2. A method the object's class defines.

   An object acquires the methods of its parent class; that is, the class for which the object was created. If the class subsequently receives new methods, objects predating the new methods *do not* acquire them.
3. A method an object's superclasses define.

   As with the object's class, only methods that existed in the superclass when the object was created are valid. REXX searches the superclass method definitions in the order that INHERIT messages were sent to an object's class.

If REXX does not find a match for the message name, REXX checks the object for method name UNKNOWN. If it exists, REXX calls the UNKNOWN method, and returns whatever the UNKNOWN method returns. For more information on the UNKNOWN method, see "Defining an UNKNOWN Method" on page 66. If the object does not have an UNKNOWN method, REXX raises a NOMETHOD condition. Any trapped information can then be inspected using REXX's CONDITION built-in function.

REXX searches *up* the hierarchy so that methods existing in higher levels can be supplemented or overridden by methods existing in lower levels.

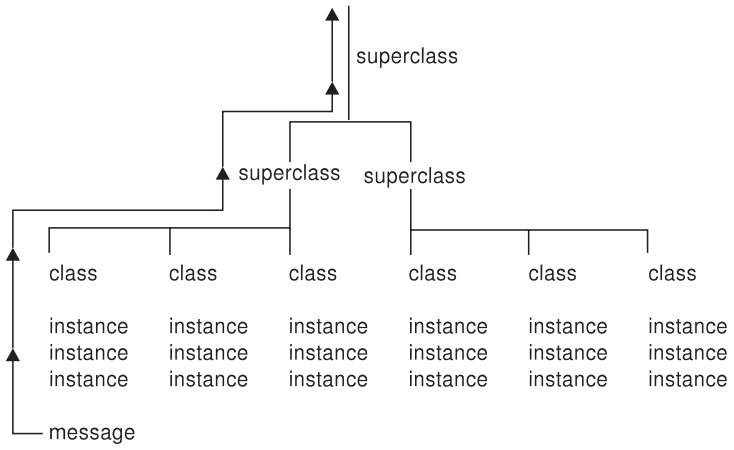


Figure 17. Searching the Hierarchy for a Method

For example, suppose you wrote a program that allows users to look up other users' phone numbers. Your program includes a class called Phone_Directory, and all its instances are users' names with phone numbers. You have included a method in Phone_Directory called NOTIFY that reports some data to a file whenever someone looks up a number. All instances of Phone_Directory use the NOTIFY method.

Now you decide you want NOTIFY, in addition to its normal handling, to personally inform you whenever anyone looks up *your* number. To accommodate this special case for your name only, you create your own NOTIFY method that adds the new task and replicates the file-handling task. You save the new method as part of your own name instance, retaining the same name, NOTIFY.

Now, when a NOTIFY message is sent to your name instance, the new version of NOTIFY is found first. REXX does not look further up the class hierarchy. The instance-level version *overrides* the version at the class level. This technique of overriding lets you change a method used by one instance without disturbing the common method used by all the other instances. It is very powerful for that reason.

Changing the Search Order for Methods

When composing a message, you can change the default search order for methods by doing both of the following:

1. Making the receiver object the sender object. You usually do this by specifying the special variable SELF. SELF holds the value of the object in which a method is running. You can use SELF to run another method in

an object where a method is already running or pass references about an object to the methods of other objects.

2. Specifying a colon and a class symbol after the message name. The class symbol identifies the class object to use as the starting point for the search. This class object must be:
   - A direct superclass of the class that defines the active method
   - The object's own class, if you used SETMETHOD to define the active method

     The class symbol is usually the special variable SUPER, but it can be any environment symbol or variable name whose value is a valid class.

In "A Sample Program Using Directives" on page 40, an Account subclass of the Object superclass is created. It defines a TYPE method for Account, and creates the Savings subclass of Account. The example defines a TYPE method for the Savings subclass, as follows:

```
::class Savings subclass Account

  ::method 'TYPE'
    return "a savings account"
```

To change the search order so REXX searches for TYPE in the Account rather than Savings subclass, enter this instead:

```
  ::method 'TYPE'
    return self~type:super '(savings)'
```

When you create an `asav` instance of the Savings subclass and send a TYPE message to `asav`:

```
say asav~type
```

REXX displays:

```
an account
```

rather than:

```
a savings account
```

because REXX searches for TYPE in the Account class first.

Public versus Private Methods

A method can be public or private. Any object can send a message that runs a *public* method. Only a message an object sends to itself, using the special variable SELF as the message receiver, can run a *private* method. Private methods include methods at different scopes within the same object. This allows superclasses to make methods available to their subclasses while

hiding those methods from other objects. A private method is like an internal subroutine. It shields the internal information of an object.

Defining an UNKNOWN Method

When an object that receives a message has no matching message name, REXX checks if the object has a method named UNKNOWN. If it does, REXX calls UNKNOWN, passing two arguments. The first is the name of the method that was not located. The second is an array containing the arguments passed with the original message.

To define an UNKNOWN message, you specify:

```
UNKNOWN(message_name,message_args)
```

Concurrency

In object-oriented programming, as in the real world, objects interact with each other. Assume, for example, throngs of people interacting at rush hour in the business district of a big city. A program that aspires to simulate the real world would have to enable many objects to interact at any given time. That could mean thousands of objects sending messages to each other, thousands of methods running at once. In REXX, this simultaneous activity is called *concurrency.* To be precise, the concurrency is *object-oriented* concurrency because it involves objects, as opposed to, for example, processes or threads.

REXX objects are inherently concurrent, and this concurrency takes the following forms:

- *Inter-object concurrency,* where several objects are active—exchanging messages, synchronizing, running their methods—at the same time
- *Intra-object concurrency,* where several methods are able to run on the same object at the same time

The default settings in REXX allow full inter-object concurrency but limited intra-object concurrency. Some situations, however, call for full intra-object concurrency.

Inter-Object Concurrency

REXX provides for inter-object concurrency, where several objects in a program can run at the same time, in the following ways:

- By early reply, by means of the REPLY instruction
- Using message objects

Early reply allows the object that sends a message to continue processing after the message is sent. Meanwhile, the receiving object runs the method

corresponding to the message. This method contains the REPLY instruction, which returns any results to the sender, interrupting the sender just long enough to reply. The sender and receiver continue operating simultaneously.

Alternatively, an independent *message object* can be created and sent to a receiver. One difference in this approach is that any reply returned does not interrupt the sender. The reply waits until the sender asks for it. In addition, message objects can notify the sender about the completion of the method it sent, and even specify synchronous or asynchronous method activation.

The chains of execution represented by the sender and receiver methods are called *activities*. An activity is a thread of execution that can run methods concurrently with methods on other activities. In other words, activities can run at the same time.

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
- A message invocation

An invocation is pushed onto an activity when an executable unit is invoked. It is removed (or popped) when execution completes.

Object Variable Pools

Every object has its own set of variables, called its *object variable pool.* These are variables associated solely with the object. When an object's method runs, it first identifies the object variables it intends to work with. Technically, it "exposes" these variables, using the REXX instruction EXPOSE. Exposing the object's variables distinguishes them from variables used by the method itself, which are not exposed. Every method an object owns—that is, all the *instance methods* in the object's class—can expose variables from the object's variable pool.

Therefore, an object variable pool includes variables:

- Exposed by methods in the object's class. This set of variables is called a *sub*pool.
- Inherited from classes elsewhere in the hierarchy (in the form of additional subpools).

All of a class's variables, together with the methods that expose them, are called a *class scope.* REXX exploits this class scope to achieve concurrency. To explain, the object variable pool is a collection of variable subpools. Each subpool is at a different scope in the object's inheritance chain. As long as the methods running on the object are at different scopes, they can run simultaneously.

Scopes, like objects, hide and protect data from outside manipulation. Methods of the same scope share the variable subpools of that scope. The scope shields the variable subpools from methods operating at other scopes. Thus, you can reuse variable names from class to class, without the variables being accessed and possibly corrupted by a method outside their own class. So class scopes divide an object's variable pool into subpools that can operate independently of one another. Several methods can use the same variable pool concurrently, as long as they confine themselves to variables in their own subpools.

Prioritizing Access to Variables

Even with class scopes and subpools, a variable is vulnerable if several methods within the scope try to access it at the same time. To handle this, REXX ensures that when a particular method is activated and exposes variables from its subpool, that method has exclusive use of the subpool until processing is complete. Until then, REXX delays the execution of any other method that needs the same subpool.

Thus if different activities send several messages to the same object, REXX forces the methods to run sequentially within a single scope. This "first-in, first-out" processing of methods in a scope prevents them from simultaneously accessing one variable, and possibly corrupting the data.

Sending Messages within an Activity

REXX makes one exception to sequential processing—when a method sends a message to itself. Assume that method M1 has exclusive access to object O, and then tries to run a second, *internal* method M2, also belonging to O. Internal method M2 would try to run, but REXX would delay it until the original method M1 finished. Yet M1 would be unable to proceed until M2 ran. The two methods would become deadlocked. In actual practice REXX intervenes by treating internal method M2 like a subroutine call. In this case, REXX runs method M2 immediately, then continues processing method M1.

The mechanism controlling this is the activity. Typically, whenever a message is invoked on an object, the activity acquires exclusive access by *locking* the object's scope. Any other activity sending a message to the object whose scope is locked must wait until the first activity releases the lock. The situation is

different, however, if the messages originate *from the same activity.* When an invocation running on an activity sends *another* message to the same object, the method is allowed to run because the activity has already acquired the lock for the scope. Thus, REXX permits nested, nonconcurrent method invocations on a single activity. No deadlocks occur because REXX treats these additional messages as subroutine calls.

Intra-Object Concurrency

Several methods can access the same object at the same time only if they are operating at different scopes. That is because they are working with separate variable subpools. If two methods in the same scope try to run on the object, REXX by default processes them on a "first-in, first-out" basis, while treating internal methods as subroutines. You can, however, achieve full intra-object concurrency. REXX offers several mechanisms for this, including:

- The SETUNGUARDED method of the Method class and the UNGUARDED option of the ::METHOD directive, which provide unconditional intra-object concurrency.
- The GUARD OFF and GUARD ON instructions, which permit switching between intra-object and default concurrency.

When intra-object concurrency at the scope level is needed, you must specifically employ these mechanisms (see the following section). Otherwise, REXX sequentially processes the methods when they are competing for the same object variables.

Activating Methods

By default, REXX assumes that an active method requires exclusive use of its object variable pool. If another method attempts access at that time, it is locked out until the first method is finished with the variable pool. This default intra-object concurrency maintains the integrity of the variable pool and prevents unexpected results. REXX manages queues for incoming requests that result in messages being sent to the same object.

Some methods can run concurrently without affecting variable pool integrity or yielding unexpected results. When a method does not need exclusive use of its object variable pool, you can use the SETUNGUARDED method or the UNGUARDED option of the ::METHOD directive to provide unconditional intra-object concurrency. These mechanisms control the locking of an object's scope when a method is invoked.

Many methods cannot use SETUNGUARDED and UNGUARDED because they sometimes require exclusive use of their variable pool. At other times, they must perform some action that involves the concurrent use of the same pool by a method on another activity. In this case, you can use the GUARD

built-in function. When the method reaches the point in its processing where it requires concurrent use of the variable pool, this function calls the GUARD OFF function. GUARD OFF lets another method that runs on a different activity become active on the same object. If the method needs to regain exclusive use, it calls GUARD ON.

For more flexibility when activating methods, you can use GUARD ON/OFF with the "WHEN *expression*" option. Add this instruction to the method code at the point where exclusive use of the variable pool becomes conditional. When processing reaches this point, REXX evaluates *expression* to determine if it is true or false.

For example, if you specify "GUARD OFF WHEN *expression*," the active method keeps running until *expression* becomes true. To become true, another method must assign or drop an object variable that is named in *expression.* Whenever an object variable changes, REXX reevaluates *expression.* If *expression* becomes true, GUARD is turned off, exclusive use of the variable pool is released, and other methods needing exclusive use can begin running. If *expression* becomes false again, GUARD is turned on and the active method regains exclusive use.

Note: If *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. A second activity can make *expression* invalid before GUARD ON WHEN can use it.

Chapter 6. Commands

From a REXX program you can pass commands to Linux or to applications designed to work with REXX. When used to run Linux commands, REXX becomes a powerful substitute for the Linux scripting facility. You can use variables, control structures, mathematics, and parsing to create procedures much more easily than with any shell script.

Applications that are designed to work with REXX are often referred to as *scriptable* applications. To work with REXX, a scriptable application registers an *environment* with REXX. An environment serves as a kind of workspace shared between REXX and the application. Environments accept application subcommands issued from REXX programs.

For example, many editors provide a command prompt or dialog box from which you can issue subcommands to set margins or add lines. If the editor is scriptable from REXX, you can issue editor subcommands from a REXX program. These REXX programs are referred to as *macros.*

When an application runs a REXX macro, REXX directs commands to the application's environment. If you issue a subcommand that the application does not recognize, it might pass the command to Linux, depending on the application.

To let you specify which environment processes a command, REXX includes an ADDRESS instruction. Starting your REXX programs from the Linux command line makes Linux the default environment for REXX commands.

How to Issue Commands

REXX makes it easy to issue commands. The basic rule is that whatever REXX cannot process it passes to the default environment. You can:

- Allow REXX to evaluate part or all of a clause as an expression. REXX automatically passes the resulting string to the default environment.
- Enclose the entire clause in quotation marks. This makes it a literal string for REXX to pass to the default environment.
- Send a command explicitly to Linux by using the ADDRESS instruction.

REXX processes your program one clause at a time. It examines each clause to determine if it is:

- A directive, such as ::CLASS or ::METHOD

- A message instruction, such as:

  ```
  .array~new
  ```

- A keyword instruction, such as:

  ```
  say 'Type total number'
  ```

  or

  ```
  pull input
  ```

- A variable assignment (any valid symbol followed by an equal sign), such as:

  ```
  price = cost * 1.2
  ```

- A label for calling other routines
- A null (empty) clause

If the clause is none of the above, REXX evaluates the entire clause as an expression and passes the resulting string to Linux.

If the string is a valid Linux command, Linux processes it as though you had typed the string at the command prompt and pressed the Enter key.

The following example shows a REXX clause that uses the **ls** command to display a list of files in the current directory.

```
/* display current directory */
say 'ls command using REXX'
'ls'
```

The clause `'ls'` is not a REXX instruction or a label, so REXX evaluates it and passes the resulting string to Linux. Linux recognizes the string **ls** as one of its commands and processes it.

Note: Always specify commands in single quotes. Otherwise, they are translated into uppercase characters and are not executed.

The following example shows how to add the PATH environment variable:

```
/* display current path setting      */
say 'example of showing the value of the PATH environment variable using REXX'
'echo $PATH'
```

The following example, **lp.cmd**, shows a program using the **ls** command and the PATH environment variable:

```
/* lp.cmd - Issue the ls command and the PATH environment variable to Linux */

say '='~copies(40)    /* display line of equal   */
                      /* signs (=) for a border  */

'ls'                  /* display listing of      */
```

```
                    /* the current directory   */

say '='~copies(40)  /* display line of =       */
'echo $PATH'        /* display the current     */
                    /* path setting            */
```

When you specify the following:

```
/home/user/tmp>rexx lp.cmd
```

you get the following output:

```
========================================

lp.cmd test demo.txt

========================================
.:/home/user/tmp:/usr/bin:/usr/lbin:/etc:/usr/sbin:/usr/ucb
/home/user/tmp
```

Note: For host commands passed to Linux using the address **cmd** previously described, a new process is created in which the system command handler is executed. However, the host commands **cd**, **export**, and **unset** are always executed in the process that runs REXX and any changes are returned to this process.

Command Echo

When your REXX program issues a Linux command, REXX passes the command to the Linux command handler for processing. This processing does not include displaying, or *echoing,* the command on the screen.

Therefore using the ECHO OFF command in your REXX program or putting an at sign (@) in front of the command has no effect.

REXX and Shell Scripts

You can use a REXX program whenever you use Linux shell scripts. REXX provides the following advantages over shell scripts:

- Easy readability (not cryptical)
- Simple but powerful arithmetic
- OO features, such as polymorphism, classes, subclasses, instances, and methods
- A large number of powerful built-in functions
- Concurrency
- Queuing

- A large number of utilities
- Debugging facilities
- Portability between different platforms

The following example shows how powerful REXX is when you make a calculation. Assume that your sample REXX program **sample.cmd** looks as follows:

```
/* REXX SCRIPT to                                  */
/* calculate the frequency from the time given.  */
/* Example of a simple calculation done by REXX. */
numeric digits 20
arg parm1 parm2
parse source . . scrname

  /* print help ------------------------------------- */
  if ((parm1 = '') | (parm1~datatype <> 'NUM')) then
   do
    say '-------------------------------------------------- '
    say ' Enter: 'scrname'  duration  unit'
    say '     o  duration - enter number of second(s); '
    say '     o  unit     - enter >s< for seconds (default);'
    say '        unit     - enter >m< for milliseconds; '
    say '        unit     - enter >u< for microseconds;  '
    say '        unit     - enter >n< for nano-seconds;  '
    say '        unit     - enter >p< for pico-seconds;  '
    say '-------------------------------------------------- '
    exit 0;
  end;

  /* calculate frequency and wavelength --------------- */
  select
    when (parm2 = 'S'  ) then nop;
    when (parm2 = 'M'  ) then parm1 = parm1 / 1000;
    when (parm2 = 'U'  ) then parm1 = parm1 / 1000000;
    when (parm2 = 'N'  ) then parm1 = parm1 / 1000000000;
    when (parm2 = 'P'  ) then parm1 = parm1 / 1000000000000;
    otherwise nop;
  end;

  frequency = 1 / parm1;

  wavelength = 299780000000 * parm1;

  /* format result ----------------------------------- */
  parse var frequency int '.' dec
  do i=length(int)-3 to 1 by -3
    int = insert(',',int,i);
  end;
  frequency = int||'.'||dec;

  parse var  wavelength  int '.' dec
  do i=length(int)-3 to 1 by -3
    int = insert(',',int,i);
```

```
  end;
  wavelength = int||'.'||dec;

  /* print result ------------------------------------ */
  if parm1 > 1 then Unit4Sec = 'seconds';
               else Unit4Sec = 'of a second';

  say copies('=',79)
  say ' A frequency of 'frequency' Hz '
  say '              has a duration of 'parm1' 'Unit4Sec';'
  say '              has a wavelength of 'wavelength' mm.'
  say copies('=',79)

exit 0;
```

If you then enter at the command prompt:

```
rexx sample.cmd 10 n
```

you get the following result:

```
================================================================================
  A frequency of 100,000,000. Hz
               has a duration of 0.00000001 of a second;
               has a wavelength of 2,997.80000000 mm.
================================================================================
```

Issuing a Command to Call a REXX Program

If you issue a command to have the system run one of its built-in commands or other programs, you can call it by name. However, to run another REXX program from your REXX program, you must call it using a CALL instruction instead of its name.

The REXX CALL instruction calls other REXX programs. To call a REXX program named **mysub1**, your CALL instruction could look like this:

```
call 'mysub1'
```

REXX recognizes the CALL instruction, handles the call, and processes **mysub1** as a REXX program.

The REXX CALL instruction does not call a non-REXX program. To call, for example, a non-REXX executable named **mysub2**, you can specify:

```
'mysub2'
```

REXX evaluates the expression and passes it to the Linux command handler for processing. The command handler processes **mysub2** as an executable.

You can also execute another REXX program within a new process by invoking the interpreter followed by the REXX program name like this:

```
"rexx mysub2"
```

However, remember that running the interpreter concurrently requires additional startup time and system resources.

Using Variables to Build Commands

You can use variables to build commands. The **shofil.cmd** program is an example. **shofil** types a file that the user specifies. It prompts the user to enter a file name and then builds a variable containing the **cat** command and the input file name.

To have REXX issue the command to the operating system, put the variable containing the command string on a line by itself. REXX evaluates the variable and passes the resultant string to Linux:

```
/* shofil.cmd - build command with variables  */

/* prompt the user for a file name            */
say "Type a file name:"

/* assign the response to variable filename;  */
/* specify parse to prevent the file name to  */
/* be translated into uppercase               */
parse pull filename

/* build a command string by concatenation    */
commandstr = 'cat' filename

/* Assuming the user typed "demo.txt,"        */
/* the variable commandstr contains           */
/* the string "cat demo.txt" and so...        */

commandstr           /* ...REXX passes the    */
                     /* string on to AIX      */
```

REXX displays the following on the screen when you run the program:

```
/home/user>rexx shofil
Type a file name:
demo.txt

This is a sample text file. Its sole
purpose is to demonstrate how
commands can be issued from REXX
programs.

/home/user>
```

Using Quotation Marks

The rules for forming a command from an expression are the same as those for forming expressions. Be careful with symbols that are used in REXX and Linux programs. The **lsrexx** program below shows how REXX evaluates a command when the command name and a variable name are the same:

```
/* lsrexx - assign a value to the symbol ls  */
say "ls command using REXX"
ls = "echo This is not a directory."

/* pass the evaluated variable to AIX        */
ls
```

Because ls is a variable that contains a string, the string is passed to the system. The **ls** command is not executed. Here are the results:

```
/home/user>rexx lsrexx
ls command using REXX
This is not a directory.
/home/user>
```

REXX evaluates a literal string—a string enclosed in matching quotation marks—exactly as it is. To ensure that a symbol in a command is not evaluated as a variable, enclose it in matching quotation marks as follows:

```
/* assign a value to the symbol ls      */
say "ls command using REXX"
ls = "echo This is another string now."

/* pass the literal string 'ls' to Linux  */
'ls'
```

REXX displays a directory listing.

The best way to ensure that REXX passes a string to the system as a command is to enclose the entire clause in quotation marks. This is especially important when you use symbols that REXX uses as operators.

If you want to use a variable in the command string, leave the variable outside the quotation marks. For example:

```
extension = "BAK"
"rm *."||extension

option = ' -al'
'ls' option
```

ADDRESS Instruction

To send a command to a specific environment, use this format of the ADDRESS instruction:

```
ADDRESS environment expression
```

For *environment*, specify the destination of the command. To address the Linux environment, use the symbol **bash**. For *expression*, specify an expression that results in a string that REXX passes to the environment. Here are some examples:

```
address bash 'ls'     /* pass the literal string  */
                      /* 'ls' to Linux             */

cmdstr = "ls *.txt"   /* assign a string           */
                      /* to a variable             */

address bash cmdstr   /* REXX passes the string    */
                      /* "ls *.txt" to Linux       */
address edit "rain"   /* REXX passes the "rain"    */
                      /* command to a fictitious   */
                      /* environment named edit    */
address bash 'ls'     /* pass the literal string   */
                      /* 'ls' to Linux             */

cmdstr = "ls *.txt"   /* assign a string           */
                      /* to a variable             */

address bash cmdstr   /* REXX passes the string    */
                      /* "ls *.txt" to Linux       */
address edit "rain"   /* REXX passes the "rain"    */
                      /* command to a fictitious   */
                      /* environment named edit    */
```

Notice that the ADDRESS instruction lets a single REXX program issue commands to two or more environments.

Using Return Codes from Commands

With each command it processes, Linux produces a number called a *return code*. When a REXX program is running, this return code is automatically assigned to a special built-in REXX variable named RC.

If the command was processed without problems, the return code is almost always 0. If something goes wrong, the return code issued is a nonzero number. The number depends on the command itself and the error encountered.

This example shows how to display a return code:

```
/* GETRC.CMD report */
"cat nosuch.fil"
say 'the return code is' RC
```

The special variable RC can be used in expressions like any other variable. In the next example, an error message is displayed when the **cat** command returns a nonzero value in RC:

```
/* Simple if/then error handler */
say "Type a file name:"
parse pull filename
'cat' filename
if RC \= 0
then say "Could not find" filename
```

This program tells you only that the system could not find a nonexistent file.

A system error does not stop a REXX program. Without some provision to stop the program, in this case a *trap*, REXX continues running. You might have to press the Control (Ctrl)+Break keys to stop processing. REXX includes the following instructions for trapping and controlling system errors:

- CALL ON ERROR
- CALL ON FAILURE
- SIGNAL ON ERROR
- SIGNAL ON FAILURE

Subcommand Processing

REXX programs can issue commands or subcommands to programs other than Linux. To determine what subcommands you can issue, refer to the documentation for the application.

To make your own applications scriptable from REXX, see "Appendix A. REXX Application Programming Interfaces" on page 97.

Trapping Command Errors

The most efficient way to detect errors from commands is by creating *condition traps*, using the SIGNAL ON and CALL ON instructions, with either the ERROR or the FAILURE condition. When used in a program, these instructions enable, or switch on, a detector in REXX that tests the result of every command. Then, if a command signals an error, REXX stops usual program processing, searches the program for the appropriate label (ERROR:, or FAILURE:, or a label that you created), and resumes processing there.

SIGNAL ON and CALL ON also tell REXX to store the line number (in the REXX program) of the command instruction that triggered the condition. REXX assigns that line number to the special variable SIGL. Your program can get more information about what caused the command error through the built-in function CONDITION.

Using the SIGNAL and CALL instructions to handle errors has several advantages; namely, that programs:

- Are easier to read because you can confine error-trapping to a single, common routine
- Are more flexible because they can respond to errors by clause (SIGL), by return code (RC), or by other information (CONDITION method or built-in function)
- Can catch problems and react to them before the environment issues an error message
- Are easier to correct because you can turn the traps on and off (SIGNAL OFF and CALL OFF)

For other conditions that can be detected using SIGNAL ON and CALL ON, see the *Object REXX for Linux: Reference.*

Instructions and Conditions

The instructions to set a trap for errors are SIGNAL and CALL. Example formats are:

```
SIGNAL ON condition NAME trapname
CALL   ON condition NAME trapname
```

The SIGNAL ON instruction initiates an exit subroutine that ends the program. The CALL ON instruction initiates a subroutine that returns processing to the clause immediately following the CALL ON instruction. You use CALL ON to *recover* from a command error or failure.

The command conditions that can be trapped are:

ERROR
Detects *any* nonzero error code the default environment issues as the result of a REXX command.

FAILURE
Detects a severe error, preventing the system from processing the command.

A failure, in this sense, is a particular category of error. If you use SIGNAL ON or CALL ON to set a trap only for ERROR conditions, then it traps failures as well as other errors. If you also specify a FAILURE condition, then the ERROR trap ignores failures.

With both the SIGNAL and the CALL instructions, you can specify the name of the trap routine. Add a NAME keyword followed by the name of the subroutine. If you do not specify the name of the trap routine, REXX uses the value of *condition* as the name (REXX looks for the label ERROR:, FAILURE:, and so on).

For more information about other conditions that can be trapped, see the *Object REXX for Linux: Reference.*

Disabling Traps

To turn off a trap for any part of a program, use the SIGNAL or CALL instructions with the OFF keyword, such as:

```
SIGNAL OFF ERROR
SIGNAL OFF FAILURE
CALL OFF ERROR
CALL OFF FAILURE
```

Using SIGNAL ON ERROR

The following example shows how a program can use SIGNAL ON to trap a command error in a program that copies a file. In this example, an error occurs because the name of a nonexistent file is stored in the variable file1. Processing jumps to the clause following the label ERROR:

```
/* example of error trap                                  */
signal on error                 /* Set the trap.          */
     .
     .
     .
'cp'  file1 file2               /* When an error occurs... */
     .
     .
exit .
error:                          /* ...REXX jumps to here   */
say "Error" rc "at line" sigl
say "Program cannot continue."
exit                            /* and ends the program.   */
```

Using CALL ON ERROR

If there were a way to recover, such as by typing another file name, you could use CALL ON to recover and resume processing:

```
/* example of error recovery */
call an error
     .
     .
     .
'cp'  file1 file2
say "Now I copied",
 "file1 to file2"
     .
     .
      exit
error:
say "Cannot find" file1
say "Type Y to continue anyway."
parse pull ans
if (ans = "Y" | ans = "Yes") then
   do
     parse pull file1
     'cp' file1 file2
     return
   end
else exit
```

A Common Error-Handling Routine

The following example shows a simple error trap that you can use in many programs:

```
/* Here is a sample "main program" with an error           */
signal on error        /* enable error handling            */
"rn myfiles.*"         /* mistyped 'rm' instruction        */
exit

/* And here is a fairly generic error-handler for this     */
/* program (and many others...)                            */
error:
say 'error' rc 'in system call.'
say
say 'line number =' sigl
say 'instruction = '  sourceline(sigl)
exit
```

Chapter 7. Input and Output

Object REXX supports a stream I/O model. In the stream model, your program reads data from various devices (such as hard disks, CD ROMs, and keyboards) as a continuous stream of characters. Your program also writes data as a continuous stream of characters.

In the stream model, a text file is represented as a stream of characters with special new-line characters marking the end of each line of text in the stream. A binary file is a stream of characters without an inherent line structure. REXX lets you read streams as lines or as characters.

To support the stream model, Object REXX includes a Stream class and many methods to use on stream objects. To input or output data, you first create an instance of the Stream class that represents the device or file you want to use. For example, the following clause creates a stream object for the file **/home/user/.profile**:

```
/* Create a stream object for .profile */
file=.stream~new('/home/user/.profile')
```

Then you send the stream object messages that are appropriate for the device or data. **.profile** is a text file, so you would normally use methods that read or write data as lines. Some of these methods are LINES, LINEIN, and LINEOUT.

If the stream represents a binary file, you can use methods that read and write data as characters. Some of these methods are CHARS, CHARIN, and CHAROUT.

The Stream class includes other methods for opening and closing streams, flushing buffers, seeking, retrieving stream status, and other input/output operations.

Many of the methods of the Stream class are also available as REXX built-in functions. Although you can use the functions, it is preferable to use the Stream class. In any case, do not mix functions and methods *on the same stream* to avoid unpredictable results.

More about Stream Objects

To use streams in Object REXX, you create new instances of the Stream class. These stream objects represent the various data sources and destinations available to your program, such as hard disks, CD ROMs, keyboards, displays, printers, serial interfaces, network. Because these sources are represented as objects, you can work with them in similar (but not identical) ways.

Stream objects can be transient or persistent. An example of a transient (or dynamic) stream object is a serial interface. Data can be sent or received from serial interfaces, but the data is not stored permanently by the serial interface itself. Consequently, you cannot, for example, search a position in the data stream. Once you read or write it, the data cannot be read again.

A disk file is an example of a persistent stream object. Because the data is stored on disk, you can search forward and backward in the stream and read data that you have previously read. REXX maintains separate read and write pointers to a stream. You can move the pointers independently using arguments on methods such as LINEIN, LINEOUT, CHARIN, and CHAROUT. REXX also provides SEEK and POSITION methods for setting the read and write positions.

Reading a Text File

The following shows an example of reading a file. Program **count.cmd** counts the words in a text file. To run it, enter **rexx count** followed by the name of the file to be processed:

```
rexx count myfile.txt
rexx count /home/user/devcon7.scr
```

count uses the String method WORDS to count the words, so **count** actually counts blank-delimited tokens:

```
/* count.cmd - counts the words in a file                          */
parse arg path              /* Get file name from command line     */
count=0                     /* Initialize a counter                */
file=.stream~new(path)      /* Create a stream object for the file */
do while file~lines<>0      /* Loop as long as there are lines     */
  text=file~linein          /* Read a line from the file           */
  count=count+(text~words)  /* Count words and add to counter      */
end
say count                   /* Display the count                   */
```

To read a file, **count** first creates a stream object for the file by sending the NEW message to the Stream class. The file name (with or without a path) is specified as an argument on the NEW method.

Within the DO loop, **count** reads the lines of the file by sending LINEIN messages to the stream object (pointed to by the variable File). The first LINEIN message causes REXX to open the file (the NEW method does not open the file). LINEIN, by default, reads one line from the file, starting at the current read position.

REXX returns only the text of the line to your program, but no new-line character.

The DO loop is controlled by the expression "file~lines<>0". The LINES method returns the number of lines remaining to be read in the file, so REXX processes the loop until no lines remain to be read.

In the **count** program, the LINEIN request forces REXX to open the file, but you can also open the file yourself using the OPEN method of the Stream class. By using the OPEN method, you control the mode in which REXX opens the file. When REXX implicitly opens a file because of a LINEIN request, it tries to open the file for both reading and writing. If that fails, it opens the file for reading. To ensure that the file is opened only for reading, you can modify **count** as follows:

```
/* count.cmd - counts the words in a file                          */
parse arg path              /* Get file name from command line     */
count=0                     /* Initialize a counter                */
file=.stream~new(path)      /* Create a stream object for the file */
openrc=file~open('read')    /* Open the file for reading           */
if openrc<>'READY:' then do /* Check the return code               */
  say 'Could not open' path||'~ RC='||openrc
  exit openrc               /* Bail out                            */
end
do while file~lines<>0      /* Loop as long as there are lines     */
  text=file~linein          /* Read a line from the file           */
  count=count+(text~words)  /* Count words and add to counter      */
end
file~close                  /* Close the file                      */
say count                   /* Display the count                   */
```

The CLOSE method, used near the end of the previous example, closes the file. A CLOSE is not required. REXX closes the stream for you when the program ends. However, it is a good idea to CLOSE streams to make the resource available for other uses.

Reading a Text File into an Array

REXX provides a Stream method, named MAKEARRAY, that reads the contents of a stream into an array object. MAKEARRAY is convenient when you need to read an entire file into memory for processing. You can read the entire file with a single REXX clause—no looping is necessary.

The following example (**cview.cmd**) uses the MAKEARRAY method to read the entire **.profile** file into an array object. **cview** displays selected lines from **.profile**. A search argument can be specified when starting **cview**:

```
rexx cview libpath
```

cview prompts for a search argument if you do not specify one.

If **cview** finds the string, it displays the line on which the string is found. **cview** continues to prompt for new search strings until you enter **q** in response to the prompt.

```
/* cview.cmd - display lines from .profile                          */
parse upper arg search_string          /* Get any command line argument */
file=.stream~new('/home/user/.profile')/* Create stream object          */
lines=file~makearray(line)             /* Read file into an array object */
                                       /* LINES points to the array obj. */
do forever
   if search_string='' then do         /* Prompt for user input          */
      say 'Enter a search string or Q to quit:'
      parse upper pull search_string
      if search_string='Q' then exit
   end  /* Do */
   do i over lines                      /* Scan the array                */
      if pos(search_string,translate(i))>0 then do
         say i                          /* Display any line that matches */
         say '='~copies(20)
      end  /* Do */
   end /* do */
   search_string=''                     /* Reset for next search         */
end /* do */
```

Reading Specific Lines of a Text File

You can read a specific line of a text file by entering a line number as an argument on the LINEIN method. In this example, line 3 is read from **.profile**:

```
/* Read and display line 3 of .profile */
infile=.stream~new('/home/user/.profile')
say infile~linein(3)
```

You do not reduce file I/O by using specific line numbers. Because text files do not have a specific record length, REXX must read through the file counting line-end characters to find the line you want.

Writing a Text File

To write lines of text to a file, you use the LINEOUT method. By default, LINEOUT appends to an existing file. The following example adds an item to a to-do list that is maintained as a simple text file:

```
/* todo.cmd - add to a todo list                              */
parse arg text
file=.stream~new('todo.dat')      /* Create a stream object    */
file~lineout(date() time() text)  /* Append a line to the file */
exit
```

In **todo**, a text string is provided as the only argument on LINEOUT. REXX writes the line of text to the file and then writes a new-line character. You do not have to provide a new-line character in the string to be written.

If you want to overwrite a file, specify a line number as a second argument to position the write pointer:

```
file~lineout('13760-0006',35)  /* Replace line 35 */
```

REXX does not prevent you from overwriting existing new-line characters in the file. Consequently, if you want to replace a line of the file without overlaying the following lines, the line you write must have the same length as the line you are replacing. Writing a line that is shorter than an existing line leaves part of the old line in the file.

Also, positioning the write pointer to line 1 does not replace the file. REXX starts writing over the existing data starting at line 1, but if you happen to write fewer bytes than previously existed in the file, your data is followed by the remainder of the old file.

To replace a file, use the OPEN method with WRITE REPLACE or BOTH REPLACE as an argument. In the following example, a file named **temp.dat** is replaced with a random number of lines. **temp.dat** is then read and displayed. You can run the example repeatedly to verify that **temp.dat** is replaced on each run.

```
/* repfile.cmd - demonstrates file replacement */
testfile=.stream~new('temp.dat') /* Create a new stream object        */
testfile~open('both replace')    /* Open for read, write, and replace */
numlines=random(1,100)           /* Pick a number from 1 to 100       */
runid=random(1,9999)             /* Pick a run identifier             */
do i=1 to numlines               /* Write the lines                   */
   testfile~lineout('Run ID:'||runid 'Line number' i)
end

/*
   Now read and display the file.  The read pointer is already at the
   beginning of the file.  MAKEARRAY reads from the read position to
   the end of the file and returns an array object containing the
   lines.
*/
filedata=testfile~makearray('line')
do i over filedata
   say i
end
testfile~close
```

The **repfile** example also demonstrates that REXX maintains separate read and write pointers to a stream. The read pointer is still at the beginning of the file while the write pointer is at the end of it.

Reading Binary Files

A binary file is a file whose data is not organized into lines using new-line characters. In most cases, you use the character I/O methods (such as CHARS, CHARIN, CHAROUT) on these files.

Suppose, for example, that you want to read the data in the **a.out** file into a variable:

```
/* geta - reads a.out into a variable                  */
infile=.stream~new('/home/user/a.out')
say 'Number of characters in the file=' infile~chars
/* Read the whole file into a single REXX variable.    */
/* REXX variables are limited by available memory.     */
myinfile=infile~charin(1,infile~chars)
say 'Number of characters read into variable' myinfile~length
```

The CHARIN method returns a string of characters from the stream, which in this case is **a.out**. CHARIN accepts two optional arguments. If no arguments are specified, CHARIN reads one character from the current read position and then advances the read pointer.

The first argument is a start position for reading the file. In the example, 1 is specified so that CHARIN begins reading with the first character of the file. Omitting the first argument achieves the same result.

The second argument specifies how many characters are to be read. To read all the characters, `infile~chars` was specified as the second argument. The CHARS method returns the number of characters remaining to be read in the input stream receiving the message. CHARIN then returns all the characters in the stream.

Reading Text Files a Character at a Time

You can use the CHARIN and other character methods on text files. Because you read the file as characters CHARIN returns the line-end character to your program. Line methods, on the contrary, do not return the line-end characters to your program.

The line-end character on Linux is a line feed (ASCII value of 10). REXX adds this character to the end of every line written using the LINEOUT method.

Text-processing applications also add the character. When reading a text file with CHARIN, interpret an ASCII sequence of 10 as the end of a line.

As an example, run the following program. It writes lines to a file using LINEOUT and then reads those lines using CHARIN. You can mix line methods and character methods. REXX maintains separate read and write pointers, so there is no need to close the file or search for another position before reading the lines just written.

```
/* linechar.cmd - demonstrate line end characters                  */
file=.stream~new('test.dat')  /* Create a new stream object        */

file~open('both replace')  /* Open the file for reading and writing */
do i=1 to 3                /* Write three lines to the file        */
   file~lineout('Line' i)
end /* do */

do while file~chars<>0     /* Read the file a character at a time   */
   byte=file~charin        /* Read a character                      */
   ascii_value=byte~c2d    /* Convert character to a decimal value  */
   if ascii_value=10 then        /* Line feed?                      */
      say 'Line feed'
   else say byte ascii_value     /* Ordinary character              */
end /* do */
file~close                       /* Close the file                  */
```

REXX does not write end-of-file characters when it closes a file that has been opened for writing.

It is not recommended to use line methods to read binary files. Your binary file might not contain any new-line characters. And, if it did, the characters probably are not meant to be interpreted as new-line characters.

Writing Binary Files

To write a binary file, you use CHAROUT. CHAROUT writes only the characters that you specify in an argument of the method. CHAROUT does not add the line-feed character to the end of the string. Here is an example:

```
/* jack.cmd - demonstrate that CHAROUT does not add new-line characters  */
filebin=.stream~new('binary.dat')   /* Create a new stream object       */
filebin~open('replace')             /* Open the file for replacement    */
do i=1 to 50                        /* Write fifty strings              */
  filebin~charout('All work and no play makes Jack a dull boy. ')
end
filebin~close                  /* Close the file so we can display it   */
'cat binary.dat'               /* Use the cat command to display file   */
```

Because new-line characters are not added, the text displayed by the **cat** command is concatenated.

CHAROUT writes the string specified and advances the write pointer. If you want to position the write pointer before writing the string, specify the starting position as a second argument:

```
filebin~charout('Jack is loosing it.',30) /* start writing at character 30 */
```

In the example, the file is explicitly opened and closed. If you do not open the file, REXX attempts to open the file for both reading and writing. If you do not close the file, REXX closes it when the procedure ends. If you omit the CLOSE method in the example, the **cat** command does not type the file. To Linux, the file does not exist yet because it is not closed yet.

Closing Files

If you do not explicitly close a file, REXX closes the file for you at the end of the procedure (that is, the end of the REXX file in which the files were opened). If your procedure is called as an external procedure by another REXX program, REXX closes the files before returning to the caller. In any case, it is recommended to explicitly close files when you are finished with them.

Direct File Access

REXX provides several ways for you to read records of a file directly (that is, in random order). The following example, **direct.cmd**, shows several cases that illustrate some of your options.

direct opens a file for both reading and writing, which is indicated by the BOTH argument of the OPEN method. The REPLACE argument of the OPEN method causes any existing **direct.dat** file to be replaced.

The OPEN method also has the arguments BINARY and RECLENGTH, which are useful for direct file access.

The BINARY argument opens the stream in binary mode, which means that line-end characters are ignored. Binary mode is useful if you want to process binary data using line methods. It is easier to use line methods for direct access. With line methods, you can search a position in a file using line numbers. With character methods, you must calculate the character displacement of the file.

The RECLENGTH argument defines a record length of 50 for the file. It enables you to use line methods in a binary-mode stream. Because REXX now knows how long each record is, it can calculate the displacement of the file for a given record number and read the record directly.

```
/* direct.cmd - demonstration of direct file access       */
db=.stream~new('direct.dat')
db~open('both replace binary reclength 50')

/* Write three records of 50 bytes each using LINEOUT     */
db~lineout('Cole, Gary:  Blue')
db~lineout('McGuire, Rick: Red')
db~lineout('Pritko, Steve: Red. Oops.. I mean blue!')

/* Case 1: Read the records in order using LINEIN.        */
say 'Case 1: Sequential reads with LINEIN...'
do i=1 to 3
   say db~linein
end
say 'Press Enter to continue'; parse pull resp

/* Case 2: Read records in random order using LINEIN      */
say 'Case 2: Random reads with LINEIN...'
do i=1 to 5
   lineno=random(1,3)
   say 'Record' lineno '=' db~linein(lineno)
end
say 'Press Enter to continue'; parse pull resp

/* Case 3: Read entire file with CHARIN                   */
say 'Case 3: Read entire file with a single CHARIN...'
say db~charin(1,150)
say 'Press Enter to continue'; parse pull resp

/* Case 4: Read file sequentially with CHARIN             */
say 'Case 4: Sequential reads with CHARIN...'
db~seek(1 read)         /* Reposition read pointer        */
do i=1 to 3
   say db~charin(,50)
end
say 'Press Enter to continue'; parse pull resp

/* Case 5: Read records in random order with CHARIN       */
say 'Case 5: Random reads with CHARIN...'
do i=1 to 5
   lineno=random(1,3)
   charno=((lineno-1)*50)+1
   say 'Record' lineno 'Character' charno '=' db~charin(charno,50)
end
say 'Press Enter to continue'; parse pull resp

/* Case 6: Write records in random order with LINEOUT     */
say 'Case 6: Replace record 2 with LINEOUT'
db~lineout('This should replace line 2',2)
do i=1 to 3
   say db~linein(i)
end
say 'Press Enter to continue'; parse pull resp

/* Case 7: Write records in random order with CHAROUT     */
```

```
say 'Case 7: Replace record 2 with CHARIN...'
db~charout('New record 2 from CHAROUT'~left(50,'.'),51)
db~seek(1 read)         /* Reposition read pointer        */
do i=1 to 3
   say db~charin(,50)
end
say 'Press Enter to continue'; parse pull resp
db~close
```

After opening the file, **direct** writes three records using LINEOUT. The records are not padded to 50 characters. REXX handles that. Because the file is opened in binary mode, REXX does not write line-end characters at the end of each line. It only writes the strings one after another to the stream.

In Case 1, the LINEIN method is used to read the file. Because the file is open in binary mode, LINEIN does not look for line-end characters to mark the end of a line. Instead, it relies on the record length that you specify on open. In fact, if there were a carriage-return or line-feed sequence of the line, REXX would return those characters to your program.

Case 2 demonstrates how to read the file in random order. In this case, the RANDOM function is used to choose a record to be retrieved. Then the desired record number is specified as an argument on LINEIN. Note that records are numbered starting from 1, not from 0. Because the file is opened in binary mode, REXX does not look for line-end characters. It uses the RECLENGTH to determine where to read. The LINEIN method can, therefore, retrieve a line directly, without having to scan through the file counting line-end characters.

Case 3 proves that no line-end characters exist in the file. The CHARIN method reads the entire file. SAY displays the returned string as one long string. If REXX inserted line-end characters, each record would be displayed on a separate line.

Case 4 shows how to read the binary mode file sequentially using CHARIN. But before reading the file, the read pointer must be reset to the beginning of the file. (Case 3 leaves the read pointer at the end of the file.) The SEEK method resets the read pointer to character 1, which is the beginning of the file. As with lines, REXX numbers characters starting with 1, not 0. Position 1 is the first character of the file.

By default, the number specified with SEEK refers to a character position. You can also search by line number or by offsets. SEEK allows offsets from the current read or write position, or from the beginning or ending of the file. If you prefer typing longer method names, you can use POSITION as a synonym for SEEK.

In the loop of Case 4, the first argument on CHARIN is omitted. The first argument tells where to position the read pointer. If it is omitted, REXX automatically advances the read pointer based on the number of characters you are reading.

Case 5 demonstrates how to read records in random order with CHARIN. In the loop, a random record number is selected and assigned to variable `lineno.` This record number is then converted to a character number, which can be used to specify the read position on CHARIN. Compare Case 5 with Case 2. In Case 2, which uses line methods, it is not necessary to perform a calculation, you just request the record you want.

Cases 6 and 7 write records in random order. Case 6 uses LINEOUT, while Case 7 uses CHAROUT. Because the file is opened in binary mode, LINEOUT does not write line-end characters. You can write over a line by specifying a line number. With CHAROUT, you need to calculate the character position of the line to be replaced. Unlike LINEOUT, you need to ensure that the string being written with CHAROUT is padded to the appropriate record length. Otherwise, part of the record being replaced remains in the file.

Consequently, for random reading of files with fixed length records, line methods are often the better choice. However, one limitation of the line methods is that you cannot use them to write sparse records. That is, if a file already has 200 records, you can use LINEOUT to write record 201, but you cannot use LINEOUT to write record 300. With CHAROUT, however, you can open a new file and start writing at character position 5000 if you choose.

Checking for the Existence of a File

To check for the existence of a file, you use the QUERY method of the Stream class. The following **isthere.cmd** program accepts a file name as a command line argument and checks for the existence of that file.

```
/* isthere.cmd - test for the existence of a file        */
parse arg fid                    /* Get the file name    */
qfile=.stream~new(fid)           /* Create stream object */
if qfile~query('exists')='' then /* Check for existence  */
   say fid 'does not exist.'
else
   say fid 'exists.'
```

In the example, a stream object is created for the file even though it might not exist. This is acceptable because the file is not opened when the stream object is created.

The QUERY method accepts one argument. To check for the existence of a file, you specify the string 'exists' as previously shown. If the file exists, QUERY returns the full-path specification of the stream object. Otherwise, QUERY returns a null string.

Getting Other Information about a File

The QUERY method can also return date and time stamps, read position, write position, the size of the file, and so on. The following example shows most of the QUERY arguments.

```
/* infoon.cmd - display information about a file */
parse arg fid
qfile=.stream~new(fid)
fullpath=qfile~query('exists')
if fullpath='' then do
   say fid 'does not exist.'
   exit
end
qfile~open('both')
say ''
say 'Full path name:' fullpath
say 'Date and time stamps (U.S. format):' qfile~query('datetime')
say '            (International format):' qfile~query('timestamp')
say ''
say 'Handle associated with stream:' qfile~query('handle')
say '                  Stream type:' qfile~query('streamtype')
say ''
say '          Size of the file (characters):' qfile~query('size')
say ' Read position (in terms of characters):' qfile~query('seek
read')
say 'Write position (in terms of characters):' qfile~query('seek
write') qfile~close
```

Using Standard I/O

All of the preceding topics dealt with the reading and writing of files. You can use the same methods to read from standard input (usually the keyboard) and to write to standard output (usually the display). You can also use the methods to write to the standard error stream. In Object REXX, these default streams are represented by public objects of the Monitor class: .input, .output, and .error.

The streams STDIN, STDOUT, and STDERR are transient streams. For transient streams, you cannot use any method or method argument for positioning the read and write pointers. You cannot, for example, use the SEEK method on STDOUT.

Writing to STDOUT has the same effect as using the SAY instruction. However, the SAY instruction always writes line-end characters at the end of the string. By using the CHAROUT method to write to STDOUT, you can control when line-end characters are written.

The following example shows a modified **count** program previously shown in "Reading a Text File" on page 84. **count** has been modified to display a progress indicator. For every line processed, **count** now uses CHAROUT to display a single period. **count** does not write any line-end characters, so the periods wrap to the next line when they reach the end of the line in the Linux window.

```
                            /* count counts the words in a file */
parse arg path                          /* Get the file name    */
count=0                                 /* Initialize the count */
file=.stream~new(path)        /* Create a stream object for the input file */
do while file~lines<>0        /* Process each line of the file */
   text=file~linein           /* Read a line                   */
   count=count+(text~words)   /* Count blank-delimited tokens  */
   .output~charout('.')       /* Write period to STDOUT        */
end
say ''
say count
```

Reading from STDIN using LINEIN is similar to reading with the PARSE PULL instruction:

```
/* inexam.cmd - example of reading STDIN with LINEIN  */

/* Prompt for input with SAY and PARSE instructions   */
say 'What is your name?'
parse pull response
say 'Hi' response
say ''

/* Now prompt using LINEOUT and LINEIN                */
.output~lineout('What is your name?')
response=.input~linein
.output~lineout('Hi' response)
```

Using character methods with STDIN and STDOUT gives you more control over the reading and writing of line-end characters. In the following example, the prompting string is written to STDOUT using CHAROUT. Because CHAROUT does not add any line-end characters to the stream, the display cursor is positioned after the prompt string on the same line.

```
/* inchar.cmd - example of reading STDIN with CHARIN  */
.output~charout('What is your name? ')
response=.input~charin(,10)
.output~charout('Hi' response)
```

CHARIN is used to read the user's response. The user's keystrokes are not returned to your program until the user presses the Enter key. In the example, a length of 10 is specified. If fewer characters than the specified length are available, CHARIN waits until they become available. Otherwise, the characters are returned to your program. CHARIN does not strip any carriage-return or line-feed characters before returning the string to your program. You can observe this with INCHAR by typing several strings that have less than ten characters and pressing Enter after each string:

```
/home/user>inchar
What is your name? John
Q.
Public
Hi John
Q.
Pu
```

Appendix A. REXX Application Programming Interfaces

This appendix describes how to interface applications to REXX or extend the REXX language by using REXX application programming interfaces (APIs). As used here, the term *application* refers to programs written in languages other than REXX. This is usually the C language. Conventions in this appendix are based on the C language. Refer to a C programming reference manual if you need a better understanding of these conventions.

The features described here let an application extend many parts of the REXX language or extend an application with REXX. This includes creating handlers for subcommands, external functions, and system exits.

Subcommands
: are commands issued from a REXX program. A REXX expression is evaluated and the result is passed as a command to the currently addressed subcommand handler. Subcommands are used in REXX programs running as application macros.

Functions
: are direct extensions of the REXX language. An application can create functions that extend the native REXX function set. Functions can be general-purpose extensions or specific to an application.

System exits
: are programmer-defined variations of the operating system. The application programmer can tailor the REXX interpreter behavior by replacing REXX system requests.

Subcommand, function, and system exit handlers have similar coding, compilation, and packaging characteristics.

In addition, applications can manipulate the variables in REXX programs (see "Variable Pool Interface" on page 135), and execute REXX routines directly from memory (see "Macrospace Interface" on page 148).

Handler Characteristics

The basic requirements for subcommand, function, and system exit handlers are:

- REXX handlers must use the APIENTRY (_stdcall) linkage convention. Handler functions should be declared with the appropriate type definition from the **rexx.h** include file. Using C++, the functions must be declared as extern C:
  - RexxSubcomHandler
  - RexxFunctionHandler
  - RexxExitHandler
- A REXX handler must be packaged as either of the following:
  - An exported routine within a library
  - An entry point within an executable module
- A handler must be registered with REXX before it can be used. REXX uses the registration information to locate and call the handler. For example, external function registration of a library external function identifies both the library and routine that contains the external function. Also note:
  - Library handlers are global to the system; any REXX program can call them.
  - Executable file handlers are local to the registering process; only a REXX program running in the same process as an executable module can call a handler packaged within that executable module.

RXSTRINGs

Many of the REXX application programming interfaces pass REXX character strings to and from a REXX procedure. The RXSTRING data structure is used to describe REXX character strings. An RXSTRING is a content-insensitive, flat model character string with a theoretical maximum length of 4 gigabytes. The following structure defines an RXSTRING:

```
typedef struct {
   ULONG           strlength;     /* length of string            */
   PCH             strptr;        /* pointer to string           */
} RXSTRING;

typedef RXSTRING *PRXSTRING;      /* pointer to an RXSTRING      */
```

Notes:

1. The **rexx.h** include file contains a number of convenient macros for setting and testing RXSTRING values.
2. An RXSTRING can have a value (including the null string, "") or it can be empty.
   - If an RXSTRING has a value, the *strptr* field is not null. The RXSTRING macro RXVALIDSTRING(string) returns TRUE.

- If an RXSTRING is the REXX null string (""), the *strptr* field is not null and the *strlength* field is 0. The RXSTRING macro RXZEROLENSTRING(string) returns TRUE.
- If an RXSTRING is empty, the field *strptr* is null. The RXSTRING macro RXNULLSTRING(string) returns TRUE.

3. When the REXX interpreter passes an RXSTRING to a subcommand handler, external function, or exit handler, the interpreter adds a null character (hexadecimal zero) at the end of the RXSTRING data. You can use the C string library functions on these strings. However, the RXSTRING data can also contain null characters. There is no guarantee that the first null character encountered in an RXSTRING marks the end of the string. You use the C string functions only when you do not expect null characters in the RXSTRINGs, such as file names passed to external functions. The *strlength* field in the RXSTRING does not include the terminating null character.
4. On calls to subcommand and external functions handlers, as well as to some of the exit handlers, the REXX interpreter expects that an RXSTRING value is returned. The REXX interpreter provides a default RXSTRING with a *strlength* of 256 for the returned information. If the returned data is shorter than 256 characters, the handler can copy the data into the default RXSTRING and set the *strlength* field to the length returned.

   If the returned data is longer than 256 characters, a new RXSTRING can be allocated using `malloc(size)`. The *strptr* field must point to the new storage and the *strlength* must be set to the string length. The REXX interpreter returns the newly allocated storage to the system for the handler routine.

Calling the REXX Interpreter

A REXX program can be run directly from the command prompt of the operating system, or from within an application.

From the Operating System

You can run a console-based REXX program directly from the operating system command prompt using **rexx** followed by the program name.

If you specify **#!/usr/local/orexx/bin/rexx** at the beginning of a REXX program, you can omit the command **rexx** when calling this program and only specify the program name. In this case, the REXX program to be called must be flagged as executable. Remember, however, that if you use this method the program is no longer portable to other systems.

From within an Application

The REXX interpreter is a library routine. Any application can call the REXX interpreter to run a REXX program. The interpreter is fully reentrant and supports REXX procedures running on several threads within the same process.

A C-language prototype for calling REXX is in the **rexx.h** include file.

The RexxStart Function

RexxStart calls the REXX interpreter to run a REXX procedure.

RexxStart(ArgCount, ArgList, ProgramName, Instore, EnvName, CallType, Exits, ReturnCode, Result)

Parameters

ArgCount *(LONG) - input*
is the number of elements in the *ArgList* array. This is the value that the *ARG()* built-in function in the REXX program returns. *ArgCount* includes RXSTRINGs that represent omitted arguments. Omitted arguments are empty RXSTRINGs (*strptr* is null).

ArgList *(PRXSTRING) - input*
is an array of RXSTRING structures that are the REXX program arguments.

ProgramName *(PSZ) - input*
is the address of the ASCII name of the REXX procedure. If *Instore* is null, *ProgramName* must contain at least the file name of the REXX procedure. You can also provide an extension, and path. A REXX program can use any extension. If you do not provide the path, the REXX interpreter uses the usual file search (current directory, then environment path).

If *Instore* is not null, *ProgramName* is the name used in the PARSE SOURCE instruction. If *Instore* requests a REXX procedure from the macrospace, *ProgramName* is the macrospace function name (see "Macrospace Interface" on page 148).

Instore *(PRXSTRING) - input*
is an array of two RXSTRING descriptors for in-storage REXX procedures. If the *strptr* fields of both RXSTRINGs are null, the interpreter searches for REXX procedure *ProgramName* in the REXX macrospace (see "Macrospace Interface" on page 148). If the procedure is not in the macrospace, the call to RexxStart terminates with an error return code.

If either *Instore strptr* field is not null, *Instore* is used to run a REXX procedure directly from storage.

Instore[0]
is an RXSTRING describing a memory buffer that contains the REXX procedure source. The source must be an exact image of a REXX procedure disk file, complete with carriage returns, line feeds, and end-of-file characters.

Instore[1]
is an RXSTRING containing the translated image of the REXX procedure. If *Instore[1]* is empty, the REXX interpreter returns the translated image in *Instore[1]* when the REXX procedure finishes running. The translated image may be used in *Instore[1]* on subsequent RexxStart calls.

If *Instore[1]* is not empty, the interpreter runs the translated image directly. The program source provided in *Instore[0]* is used only if the REXX procedure uses the SOURCELINE built-in function. *Instore[0]* can be empty if SOURCELINE is not used. If *Instore[0]* is empty and the procedure uses the SOURCELINE built-in function, SOURCELINE() returns no lines and any attempt to access the source returns Error 40.

If *Instore[1]* is not empty, but does not contain a valid REXX translated image, unpredictable results can occur. The REXX interpreter might be able to determine that the translated image is incorrect and translate the source again.

Instore[1] is both an input and an output parameter.

If the procedure is executed from disk, the *Instore pointer* must be null. If the first argument string in *Arglist* contains the string //T and the *CallType* is RXCOMMAND, the interpreter translates the procedure source and writes the translated image to disk. Because the *Instore pointer* is null, the translated image is not returned in an *Instore* parameter.

The program calling RexxStart must release *Instore[1]* using `free(ptr)` when the translated image is no longer needed.

Only the interpreter version that created the image can run the translated image. Therefore, neither change the format of the translated image of a REXX program, nor move a translated image to other systems or save it for later use. You can, however, use the translated image several times during a single application execution.

EnvName *(PSZ) - input*
is the address of the initial ADDRESS environment name. The ADDRESS environment is a subcommand handler registered using RexxRegisterSubcomExe or RexxRegisterSubcomDll. *EnvName* is used as the initial setting for the REXX ADDRESS instruction.

If *EnvName* is null, the file extension is used as the initial ADDRESS environment. The environment name cannot be longer than 250 characters.

CallType *(LONG) - input*
is the type of the REXX procedure execution. Allowed execution types are:

RXCOMMAND
The REXX procedure is a system or application command. REXX commands usually have a single argument string. The REXX PARSE SOURCE instruction returns `COMMAND` as the second token.

RXSUBROUTINE
The REXX procedure is a subroutine of another program. The subroutine can have several arguments and does not need to return a result. The REXX PARSE SOURCE instruction returns SUBROUTINE as the second token.

RXFUNCTION
The REXX procedure is a function called from another program. The subroutine can have several arguments and must return a result. The REXX PARSE SOURCE instruction returns `FUNCTION` as the second token.

Exits *(PRXSYSEXIT) - input*
is an array of RXSYSEXIT structures defining exits for the REXX interpreter to be used. The RXSYSEXIT structures have the following form:

```
typedef struct {
   PSZ            sysexit_name;  /* name of exit handler        */
   LONG           sysexit_code;  /* system exit function code   */
} RXSYSEXIT;
```

The *sysexit_name* is the address of an ASCII exit handler name registered with RexxRegisterExitExe or RexxRegisterExitDll. *Sysexit_code* is a code identifying the handler exit type. See “System Exit Interface” on page 119 for exit code definitions. An RXENDLST entry identifies the system-exit list end. *Exits* must be null if exits are not used.

ReturnCode *(PLONG) - output*
is the integer form of the *Result* string. If the *Result* string is a whole number in the range –(2**15) to 2**15–1, it is converted to an integer and also returned in *ReturnCode*.

Result *(PRXSTRING) - output*
is the string returned from the REXX procedure with the REXX RETURN or EXIT instruction. A default RXSTRING can be provided for the returned result. If a default RXSTRING is not provided or the default is too small for the returned result, the REXX interpreter allocates an

RXSTRING using `malloc(size)`. The caller of RexxStart is responsible for releasing the RXSTRING storage with `free(ptr)`.

The REXX interpreter does not add a terminating null to *Result*.

Return Codes

The possible RexxStart return codes are:

negative
: Interpreter errors. See Appendix A in the *Object REXX for Linux: Reference* for the list of REXX errors.

0
: No errors occurred. The REXX procedure ran normally.

positive
: A system return code that indicates problems finding or loading the interpreter. See the return codes for the Linux functions dlopen and dlsym for details.

When a macrospace REXX procedure (see "Macrospace Interface" on page 148) is not loaded in the macrospace, the return code is -3 ("Program is unreadable").

Example

```
LONG      return_code;                /* interpreter return code    */
RXSTRING  argv[1];                    /* program argument string    */
RXSTRING  retstr;                     /* program return value       */
LONG      rc;                         /* converted return code      */
CHAR      return_buffer[250];         /* returned buffer            */

                                      /* build the argument string  */
   MAKERXSTRING(argv[0], macro_argument,
       strlen(macro_argument));
                                      /* set up default return      */
   MAKERXSTRING(retstr, return_buffer, sizeof(return_buffer));

   return_code = RexxStart(1,         /* one argument               */
                           argv,      /* argument array             */
                           "CHANGE.ED", /* REXX procedure name      */
                           NULL,      /* use disk version           */
                           "Editor",  /* default address name       */
                           RXCOMMAND, /* calling as a subcommand    */
                           NULL,      /* no exits used              */
                           &rc,       /* converted return code      */
                           &retstr);  /* returned result            */

                                      /* process return value       */
.
.
.
   RexxWaitForTermination();
```

```
                                    /* need to return storage?    */
  if (RXSTRPTR(retval) != return_buffer)
    free(RXSTRPTR(retval));         /* release the RXSTRING       */
```

When RexxStart is executed within an external program (usually a C program), it runs synchronously with the main REXX activity (thread) that it started. That is, when the main activity terminates, RexxStart returns, and the external program continues running.

To run RexxStart asynchronously, you can start several concurrent activities from the main activity using START or REPLY (see "Concurrency" on page 66). RexxStart is still synchronous to the main activity, that is, it returns when the main activity terminates, but it is asynchronous to the concurrent activities. If a concurrent activity runs longer than the main activity, RexxStart returns when the main activity ends, but the concurrent activity continues to run.

If, however, a concurrent activity is still running when the external program ends, the activity is terminated. To ensure that all activities finish executing, use RexxWaitForTermination or a looped RexxDidRexxTerminate in your external program after RexxStart. The use of RexxWaitForTermination is recommended also for programs that are not expected to use concurrency.

The following example demonstrates how to call MyCMD.CMD from a Cobol program (REXX is running in the same process as the Cobol program).

```
       PROCESS PGMNAME(MIXED)
      * You need to specify REXX.LIB when you link this program,
      * for example on the COB2 command.
      * Note that the name RexxStart, used later, is case-sensitive,
      * and requires the PGMNAME(MIXED) compiler option.
      ************************************************************
       IDENTIFICATION DIVISION.
      ************************************************************
       PROGRAM-ID.    'CALLREXX'
       AUTHOR.        IBM VISUALAGE FOR COBOL.

      ************************************************************
      *NAME:     CALLREXX                                       ***
      *                                                        ***
      *FUNCTION: CALL A REXX PROCEDURE NAME XXXXXXXX,           ***
      *          PASSING ARGUMENT AND GETTING RETURNED DATA.    ***
      *                                                         ***
      *EXTERNAL SUBROUTINES: NONE                              ***
      *COPY MEMBERS: NONE                                      ***
      *                                                        ***
      ************************************************************

      ************************************************************
       ENVIRONMENT DIVISION.
      ************************************************************
```

```
 CONFIGURATION SECTION.

************************************************************
 DATA DIVISION.
************************************************************

 WORKING-STORAGE SECTION.

************************************************************
*  INTERNAL VARIABLES                                      *
************************************************************

 01  WS-WORK-FIELDS.
     05  WS-RESULT-AREA          PIC X(255)     VALUE SPACES.
     05  WS-ARGUMENT-AREA        PIC X(255)     VALUE SPACES.
     05  WS-PARM1                PIC X(50)      VALUE '55'.
     05  WS-PARM2                PIC X(8)       VALUE '66'.
 01  WS-REXXSTART-PARAMETERS.
     05  WS-REXX-ARGUMENT-COUNT  PIC S9(9)      VALUE +1  COMP-5.
     05  WS-REXX-ARGUMENT-LIST.
         10  WS-ARG-LENGTH       PIC  9(9)      COMP-5.
         10  WS-ARG-POINTER                     POINTER.
     05  WS-REXX-PROGRAM-NAME    PIC X(255)     VALUE LOW-VALUES.
     05  WS-REXX-ENV-NAME        PIC X(20)      VALUE LOW-VALUES.
     05  WS-REXX-RETURN-CODE     PIC S9(9)      VALUE 0   COMP-5.
     05  WS-REXX-RESULT.
         10  WS-RESULT-LENGTH    PIC  9(9)      COMP-5.
         10  WS-RESULT-POINTER                  POINTER.
     05  WS-REXX-INTERPRETER-RC  PIC S9(9)      COMP-5.

 LINKAGE SECTION.
 PROCEDURE DIVISION.
      SET WS-ARG-POINTER          TO ADDRESS OF WS-ARGUMENT-AREA.
      MOVE LENGTH OF WS-ARGUMENT-AREA
                                  TO WS-ARG-LENGTH.
      STRING WS-PARM1                  DELIMITED BY SPACE
             ' '                       DELIMITED BY SIZE
             WS-PARM2                  DELIMITED BY SPACE
                                       INTO WS-ARGUMENT-AREA
      END-STRING.
      STRING 'MYCMD'                   DELIMITED BY SIZE
             X'00'                     DELIMITED BY SIZE
                                       INTO WS-REXX-PROGRAM-NAME
      END-STRING.
      STRING 'CGISRV'                  DELIMITED BY SIZE
             X'00'                     DELIMITED BY SIZE
                                       INTO WS-REXX-ENV-NAME
      END-STRING.
      SET WS-RESULT-POINTER            TO
                                       ADDRESS OF WS-RESULT-AREA.
      MOVE LENGTH OF WS-RESULT-AREA    TO WS-RESULT-LENGTH.

* Note that the name RexxStart is case sensitive
      CALL 'RexxStart'   USING
                   BY VALUE     WS-REXX-ARGUMENT-COUNT
```

```
                    BY REFERENCE WS-REXX-ARGUMENT-LIST
                    BY REFERENCE WS-REXX-PROGRAM-NAME
                    BY VALUE     0
                    BY REFERENCE WS-REXX-ENV-NAME
                    BY VALUE     0
                    BY VALUE     0
                    BY REFERENCE WS-REXX-RETURN-CODE
                    BY REFERENCE WS-REXX-RESULT
                    RETURNING    WS-REXX-INTERPRETER-RC.
DISPLAY WS-REXX-RETURN-CODE ' ' WS-REXX-INTERPRETER-RC.
DISPLAY WS-RESULT-LENGTH.
DISPLAY WS-RESULT-AREA.
GOBACK.
```

The RexxWaitForTermination Function

RexxWaitForTermination waits for the termination of all activities of the program that were started by RexxStart. This function puts your program into a wait state. It cannot continue until the activities have finished.

RexxWaitForTermination()

The RexxDidRexxTerminate Function

RexxDidRexxTerminate checks for the termination of all activities of the program that were started by RexxStart, and allows your program to continue.

RexxDidRexxTerminate()

Return Codes

0 Activities are still running.

1 All activities have been terminated.

Example

```
while (!RexxDidRexxTerminate())
{
   /* do your processing */
}
```

Subcommand Interface

An application can create handlers to process commands from a REXX program. Once created, the subcommand handler name can be used with the RexxStart function or the REXX ADDRESS instruction. Subcommand handlers must be registered with the RexxRegisterSubcomExe or RexxRegisterSubcomDll function before they are used.

Registering Subcommand Handlers

A subcommand handler can reside in the same module (executable or library) as an application, or it can reside in a separate library. It is recommended that an application that runs REXX procedures with RexxStart uses RexxRegisterSubcomExe to register subcommand handlers. The REXX interpreter passes commands to the application subcommand handler entry point. Subcommand handlers created with RexxRegisterSubcomExe are available only to REXX programs called from the registering application.

The RexxRegisterSubcomDll interface creates subcommand handlers that reside in a library. Any REXX program using the REXX ADDRESS instruction can access a library subcommand handler. A library subcommand handler can also be registered directly from a REXX program using the RXSUBCOM command.

Creating Subcommand Handlers

The following example is a sample subcommand handler definition.

```
ULONG APIENTRY command_handler(
  PRXSTRING Command,    /* Command string from REXX           */
  PUSHORT   Flags,      /* Returned Error/Failure flags       */
  PRXSTRING Retstr);    /* Returned RC string                 */
```

where:

Command
is the command string created by REXX.

command is a null-terminated RXSTRING containing the issued command.

Flags is the subcommand completion status. The subcommand handler can indicate success, error, or failure status. The subcommand handler can set *Flags* to one of the following values:

RXSUBCOM_OK
The subcommand completed normally. No errors occurred during subcommand processing and the REXX procedure continues when the subcommand handler returns.

RXSUBCOM_ERROR
A subcommand error occurred. RXSUBCOM_ERROR indicates a subcommand error occurred; for example, incorrect command options or syntax.

If the subcommand handler sets *Flags* to RXSUBCOM_ERROR, the REXX interpreter raises an ERROR condition if SIGNAL ON

ERROR or CALL ON ERROR traps have been created. If TRACE ERRORS has been issued, REXX traces the command when the subcommand handler returns.

RXSUBCOM_FAILURE
A subcommand failure occurred. RXSUBCOM_FAILURE indicates that general subcommand processing errors have occurred. For example, unknown commands usually return RXSUBCOM_FAILURE.

If the subcommand handler sets *Flags* to RXSUBCOM_FAILURE, the REXX interpreter raises a FAILURE condition if SIGNAL ON FAILURE or CALL ON FAILURE traps have been created. If TRACE FAILURES has been issued, REXX traces the command when the subcommand handler returns.

Retstr is the address of an RXSTRING for the return code. It is a character string return code that is assigned to the REXX special variable RC when the subcommand handler returns to REXX. The REXX interpreter provides a default 256-byte RXSTRING in *Retstr*. A longer RXSTRING can be allocated with `malloc(size)` if the return string is longer than the default RXSTRING. If the subcommand handler sets *Retstr* to an empty RXSTRING (a null *strptr*), REXX assigns the string 0 to RC.

Example:

```
ULONG APIENTRY Edit_Commands(
  PRXSTRING Command,    /* Command string passed from the caller    */
  PUSHORT   Flags,      /* pointer too short for return of flags    */
  PRXSTRING Retstr)     /* pointer to RXSTRING for RC return        */
{
  LONG      command_id;               /* command to process         */
  LONG      rc;                       /* return code                */
  PSZ       scan_pointer;             /* current command scan       */
  PSZ       target;                   /* general editor target      */

  scan_pointer = command->strptr;     /* point to the command       */
                                      /* resolve command            */
  command_id = resolve_command(&scan_pointer);

  switch (command_id) {               /* process based on command   */

    case   LOCATE:                    /* locate command             */

                                      /* validate rest of command   */
      if (rc = get_target(&scan_pointer, &target)) {
        *Flags = RXSUBCOM_ERROR;      /* raise an error condition   */
        break;                        /* return to REXX             */
      }
      rc = locate(target);            /* locate target in the file  */
      *Flags = RXSUBCOM_OK;           /* not found is not an error  */
      break;                          /* finish up                  */
```

```
:

      default:                          /* unknown command           */
        rc = 1;                         /* return code for unknown   */
        *Flags = RXSUBCOM_FAILURE;      /* this is a command failure */
        break;
  }

  sprintf(Retstr->strptr, "%d", rc);    /* format return code string */
                                        /* and set the correct length */
  Retstr->strlength = strlen(Retstr->strptr);
  return 0;                             /* processing completed      */
}
```

Subcommand Interface Functions

The following sections explain the functions for registering and using subcommand handlers.

RexxRegisterSubcomDll

RexxRegisterSubcomDll registers a subcommand handler that resides in a library routine.

RexxRegisterSubcomDll(EnvName, ModuleName, EntryPoint, UserArea, DropAuth)

Parameters:

EnvName *(PSZ) - input*
is the address of an ASCII subcommand handler name.

ModuleName *(PSZ) - input*
is the address of an ASCII library name. *ModuleName* is the library file containing the subcommand handler routine.

EntryPoint *(PSZ) - input*
is the address of an ASCII library procedure name. *EntryPoint* is the name of the exported routine within *ModuleName* that REXX calls as a subcommand handler.

UserArea *(PUCHAR) - input*
is the address of an 8-byte area of user-defined information. The 8 bytes *UserArea* addresses are saved with the subcommand handler registration. *UserArea* can be null if there is no user information to be saved. The RexxQuerySubcom function can retrieve the saved user information.

DropAuth *(ULONG) - input*
is the drop authority. *DropAuth* identifies the processes that can deregister the subcommand handler. The possible *DropAuth* values are:

RXSUBCOM_DROPPABLE
Any process can deregister the subcommand handler with RexxDeregisterSubcom.

RXSUBCOM_NONDROP
Only a thread within the same process as the thread that registered the handler can deregister the handler with RexxDeregisterSubcom.

Return Codes:

0 RXSUBCOM_OK

10 RXSUBCOM_DUP

1002 RXSUBCOM_NOEMEM

Remarks: *EntryPoint* can only be a 32-bit routine.

RexxRegisterSubcomExe

RexxRegisterSubcomExe registers a subcommand handler that resides within the application code.

RexxRegisterSubcomExe(EnvName, EntryPoint, UserArea)

Parameters:

EnvName *(PSZ) - input*
is the address of an ASCII subcommand handler name.

EntryPoint *(PFN) - input*
is the address of the subcommand handler entry point within the application executable code.

UserArea *(PUCHAR) - input*
is the address of an 8-byte area of user-defined information. The 8 bytes *UserArea* addresses are saved with the subcommand handler registration. *UserArea* can be null if there is no user information to be saved. The RexxQuerySubcom function can retrieve the user information.

Return Codes:

0 RXSUBCOM_OK

10 RXSUBCOM_DUP

30 RXSUBCOM_NOTREG

1002 RXSUBCOM_NOEMEM

Remarks: If *EnvName* is the same as a subcommand handler already registered with RexxRegisterSubcomDll, RexxRegisterSubcomExe returns

RXSUBCOM_DUP. This is not an error condition. It means that RexxRegisterSubcomExe has successfully registered the new subcommand handler.

A REXX procedure can register library subcommand handlers with the RXSUBCOM command. For example:

```
                            /* register Dialog Manager       */
                            /* subcommand handler            */
'RXSUBCOM REGISTER ISPCIR ISPCIR ISPCIR'
Address ispcir              /* send commands to dialog mgr   */
```

The RXSUBCOM command registers the Dialog Manager subcommand handler ISPCIR as routine ISPCIR in the ISPCIR library.

Example:

```
WORKAREARECORD  *user_info[2];       /* saved user information     */

user_info[0] = global_workarea;      /* save global work area for  */
user_info[1] = NULL;                 /* re-entrance                */

rc = RexxRegisterSubcomExe("Editor", /* register editor handler    */
    &Edit_Commands,                  /* located at this address    */
    user_info);                      /* save global pointer        */
```

RexxDeregisterSubcom

RexxDeregisterSubcom deregisters a subcommand handler.

RexxDeregisterSubcom(EnvName, ModuleName)

Parameters:

EnvName *(PSZ) - input*
is the address of an ASCII subcommand handler name.

ModuleName *(PSZ) - input*
is the address of an ASCII library name. *ModuleName* is the name of the library containing the registered subcommand handler. When *ModuleName* is null, RexxDeregisterSubcom searches the RexxRegisterSubcomExe subcommand handler list for a handler within the current process. If RexxDeregisterSubcom does not find a RexxRegisterSubcomExe handler, it searches the RexxRegisterSubcomDll subcommand handler list.

Return Codes:

0 RXSUBCOM_OK

30 RXSUBCOM_NOTREG

40 RXSUBCOM_NOCANDROP

Remarks: The handler is removed from the active subcommand handler list.

RexxQuerySubcom

RexxQuerySubcom queries a subcommand handler and retrieves saved user information.

RexxQuerySubcom(EnvName, ModuleName, Flag, UserWord)

Parameters:

EnvName *(PSZ) - input*
is the address of an ASCII subcommand handler name.

ModuleName *(PSZ) - input*
is the address of an ASCII library name. *ModuleName* restricts the query to a subcommand handler within the *ModuleName* library. When *ModuleName* is null, RexxQuerySubcom searches the RexxRegisterSubcomExe subcommand handler list for a handler within the current process. If RexxQuerySubcom does not find a RexxRegisterSubcomExe handler, it searches the RexxRegisterSubcomDll subcommand handler list.

Flag *(PUSHORT) - output*
is the subcommand handler registration flag. *Flag* is the *EnvName* subcommand handler registration status. When RexxQuerySubcom returns RXSUBCOM_OK, the *EnvName* subcommand handler is currently registered. When RexxQuerySubcom returns RXSUBCOM_NOTREG, the *EnvName* subcommand handler is not registered.

UserWord *(PUCHAR) - output*
is the address of an 8-byte area that receives the user information saved with RexxRegisterSubcomExe or RexxRegisterSubcomDll. *UserWord* can be null if the saved user information is not required.

Return Codes:

0 RXSUBCOM_OK

30 RXSUBCOM_NOTREG

Example:

```
ULONG APIENTRY Edit_Commands(
  PRXSTRING Command,    /* Command string passed from the caller    */
  PUSHORT   Flags,      /* pointer too short for return of flags    */
  PRXSTRING Retstr)     /* pointer to RXSTRING for RC return        */
{
  WORKAREARECORD  *user_info[2];       /* saved user information     */
  WORKAREARECORD   global_workarea;    /* application data anchor    */
  USHORT           query_flag;         /* flag for handler query     */
```

```
rc = RexxQuerySubcom("Editor",       /* retrieve application work  */
    NULL,                            /* area anchor from REXX      */
    &query_flag,
    user_info);

global_workarea = user_info[0];      /* set the global anchor      */
```

Return Codes

RXSUBCOM_ERROR	0x01	An error in subcommand execution has occurred; the interpreter raises an ERROR condition.
RXSUBCOM_FAILURE	0x02	A failure in subcommand execution has occurred; the interpreter raises a FAILURE condition.
RXSUBCOM_NOEMEM	1002	There is insufficient memory available to complete this request.
RXSUBCOM_OK	0	A subcommand has executed successfully.
RXSUBCOM_DUP	10	A duplicate handler name has been successfully registered. There is either an executable handler with the same name registered in another process, or a library handler with the same name registered in another library. (To address this subcommand, you must specify its library name.)
RXSUBCOM_NOTREG	30	Registration was unsuccessful due to duplicate handler and library names (RexxRegisterSubcomExe or RexxRegisterSubcomDll); the subroutine environment is not registered (other REXX subcommand functions).
RXSUBCOM_NOCANDROP	40	The subcommand handler has been registered as “not droppable.”
RXSUBCOM_LOADERR	50	An error has occurred during the loading of a library; most commonly this is caused by a missing library file.

External Function Interface

There are two types of REXX external functions:

- Routines written in REXX
- Routines written in other Linux-supported languages

External functions written in REXX do not need to be registered. These functions are found by a disk search for a REXX procedure file that matches the function name.

Registering External Functions

An external function can reside in the same module (executable or library) as an application, or in a separate library. RexxRegisterFunctionExe registers

external functions within an application module. External functions registered with RexxRegisterFunctionExe are available only to REXX programs called from the registering application.

The RexxRegisterFunctionDll interface registers external functions that reside in a library. Any REXX program can access such an external function after it is registered. It can also be registered directly from a REXX program using the REXX RXFUNCADD built-in function.

Creating External Functions

The following is a sample external function definition:

```
ULONG APIENTRY SysLoadFuncs(
     PSZ       Name,                  /* name of the function       */
     LONG      Argc,                  /* number of arguments        */
     RXSTRING  Argv[],                /* list of argument strings   */
     PSZ       Queuename,             /* current queue name         */
     PRXSTRING Retstr)                /* returned result string     */
```

where:

Name is the address of an ASCII function name used to call the external function.

Argc is the number of elements in the *Argv* array. *Argv* contains *Argc* RXSTRINGs.

Argv is an array of null-terminated RXSTRINGs for the function arguments.

Queuename
is the name of the currently defined external REXX data queue.

Retstr is the address of an RXSTRING for the returned value. *Retstr* is a character string function or subroutine return value. When a REXX program calls an external function with the REXX CALL instruction, *Retstr* is assigned to the special REXX variable RESULT. When the REXX program calls an external function with a function call, *Retstr* is used directly within the REXX expression.

The REXX interpreter provides a default 256-byte RXSTRING in *Retstr*. A longer RXSTRING can be allocated with `malloc(size)` if the returned string is longer than 256 bytes. The REXX interpreter releases *Retstr* with `free(ptr)` when the external function completes.

Returns
is an integer return code from the function. When the external function returns `0`, the function completed successfully. *Retstr* contains the return value. When the external function returns a nonzero return code, the REXX interpreter raises REXX error 40, “Incorrect call to routine”. The *Retstr* value is ignored.

If the external function does not have a return value, the function must set *Retstr* to an empty RXSTRING (null *strptr*). When an external function called as a function does not return a value, the interpreter raises error 44, “Function or message did not return data”. When an external function called with the REXX CALL instruction does not return a value, the REXX interpreter drops (unassigns) the special variable RESULT.

Calling External Functions

RexxRegisterFunctionExe external functions are local to the registering process. Another process can call the RexxRegisterFunctionExe to make these functions local to this process. RexxRegisterFunctionDll functions, however, are available to all processes. The function names cannot be duplicated.

Example

```
ULONG APIENTRY SysMkDir(
     PSZ       Name,                  /* name of the function       */
     LONG      Argc,                  /* number of arguments        */
     RXSTRING  Argv[],                /* list of argument strings   */
     PSZ       Queuename,             /* current queue name         */
     PRXSTRING Retstr)                /* returned result string     */
{
  ULONG  rc;                          /* Return code of function    */

  if (Argc != 1)                      /* must be 1 argument         */
    return 40;                        /* incorrect call if not      */

                                      /* make the directory         */
  rc = mkdir(Argv[0].strptr, S_IRWXU|S_IRGRP|S_IKGRP|S_IROTH|S_IXOTH);

  sprintf(Retstr->strptr, "%d", rc);  /* result: <> 0 failed        */
                                      /* set proper string length   */
  Retstr->strlength = strlen(Retstr->strptr);
  return 0;                           /* successful completion      */
}
```

External Function Interface Functions

The following sections explain the functions for registering and using external functions.

RexxRegisterFunctionDll

RexxRegisterFunctionDll registers an external function that resides in a library routine.

RexxRegisterFunctionDll(FuncName, ModuleName, EntryPoint)

Parameters:

FuncName *(PSZ) - input*
is the address of an external function name. The function name must not exceed 1024 characters.

ModuleName *(PSZ) - input*
is the address of an ASCII library name. *ModuleName* is the file containing the external function routine.

EntryPoint *(PSZ) - input*
is the address of an ASCII procedure name. *EntryPoint* is the name of the exported external function routine within *ModuleName*.

Return Codes:

0 RXFUNC_OK

10 RXFUNC_DEFINED

20 RXFUNC_NOMEM

Remarks: *EntryPoint* can only be a 32-bit routine.

External functions that reside in a library routine must be exported. You can do this by specifying a module-definition (**.exp**) file that lists the external functions in the EXPORT section. You must export these functions with both the mixed-case and the uppercase name. For example:

```
EXPORTS
   SYSMKDIR = SysMkDir
```

A REXX procedure can register library-external functions with the RXFUNCADD built-in function. For example:

```
                            /* register function SysLoadFuncs*/
                            /* in library librexxutil.so     */
Call RxFuncAdd 'SysLoadFuncs', 'rexxutil', 'SysLoadFuncs'
Call SysLoadFuncs           /* call to load other functions  */
```

RXFUNCADD registers the external function SysLoadFuncs as routine SysLoadFuncs in the **librexxutil.so** library. SysLoadFuncs registers additional functions in **librexxutil.so** with RexxRegisterFunctionDll. To call the library, you specify only the name, namely **rexxutil**. The system automatically adds the prefix **lib** and the suffix **.so**. The following function registration example includes a SysLoadFuncs routine.

Example:

```
static PSZ  RxFncTable[] =            /* function package list      */
{
      "SysCls",
      "SysFileTree",
      "SysGetMessage",
      }
```

```
ULONG SysLoadFuncs(
     PSZ       Name,                  /* name of the function       */
     LONG      Argc,                  /* number of arguments        */
     RXSTRING  Argv[],                /* list of argument strings   */
     PSZ       Queuename,             /* current queue name         */
     PRXSTRING Retstr)                /* returned result string     */
{
  INT    entries;                     /* Number of entries          */
  INT    j;                           /* Counter                    */

  Retstr->strlength = 0;              /* set null string return     */

  if (Argc > 0)                       /* check arguments            */
    return 40;                        /* too many, raise an error   */

                                      /* get count of arguments     */
  entries = sizeof(RxFncTable)/sizeof(PSZ);
                                      /* register each function in  */
  for (j = 0; j < entries; j++) {     /* the table                  */
    RexxRegisterFunctionDll(RxFncTable[j],
          "rexxutil", RxFncTable[j]);
  }
  return 0;                           /* successful completion      */
}
```

RexxRegisterFunctionExe

RexxRegisterFunctionExe registers an external function that resides within the application code.

RexxRegisterFunctionExe(FuncName, EntryPoint)

Parameters:

FuncName *(PSZ) - input*
is the address of an external function name. The function name must not exceed 1024 characters.

EntryPoint *(PFN) - input*
is the address of the external function entry point within the executable application file. Functions registered with RexxRegisterFunctionExe are *local* to the current process. REXX procedures in the same process as the RexxRegisterFunctionExe issuer can call local external functions.

Return Codes:

0 RXFUNC_OK

10 RXFUNC_DEFINED

20 RXFUNC_NOMEM

RexxDeregisterFunction

RexxDeregisterFunction deregisters an external function.

RexxDeregisterFunction(FuncName)

Parameters:

FuncName *(PSZ) - input*
is the address of an external function name to be deregistered.

Return Codes:

0 RXFUNC_OK

30 RXFUNC_NOTREG

RexxQueryFunction

RexxQueryFunction queries the existence of a registered external function.

RexxQueryFunction(FuncName)

Parameters:

FuncName *(PSZ) - input*
is the address of an external function name to be queried.

Return Codes:

0 RXFUNC_OK

30 RXFUNC_NOTREG

Remarks: RexxQueryFunction returns RXFUNC_OK only if the requested function is available to the current process. If not, the RexxQueryFunction searches the external RexxRegisterFunctionDll function list.

Return Codes

RXFUNC_OK	0	The call to the function completed successfully.
RXFUNC_DEFINED	10	The requested function is already registered.
RXFUNC_NOMEM	20	There is not enough memory to register a new function.
RXFUNC_NOTREG	30	The requested function is not registered.
RXFUNC_MODNOTFND	40	The library module could not be found.
RXFUNC_ENTNOTFND	50	The library entry point could not be found.

System Exit Interface

The REXX system exits let the programmer create a customized REXX operating environment. You can set up user-defined exit handlers to process specific REXX activities.

Applications can create exits for:

- The administration of resources at the beginning and the end of interpretation
- Linkages to external functions and subcommand handlers
- Special language features; for example, input and output to standard resources
- Polling for halt and external trace events

Exit handlers are similar to subcommand handlers and external functions. Applications must register named exit handlers with the REXX interpreter. Exit handlers can reside in libraries or within an executable application module.

Writing System Exit Handlers

The following is a sample exit handler definition:

```
LONG APIENTRY Rexx_IO_exit(
     LONG  ExitNumber,    /* code defining the exit function    */
     LONG  Subfunction,   /* code defining the exit subfunction */
     PEXIT ParmBlock);    /* function-dependent control block   */
```

where:

ExitNumber
is the major function code defining the type of exit call.

Subfunction
is the subfunction code defining the exit event for the call.

ParmBlock
is a pointer to the exit parameter list.

The exit parameter list contains exit-specific information. See the exit descriptions following the parameter list formats.

Note: Some exit subfunctions do not have parameters. *ParmBlock* is set to null for exit subfunctions without parameters.

Exit Return Codes

Exit handlers return an integer value that signals one of the following actions:

RXEXIT_HANDLED
The exit handler processed the exit subfunction and updated the subfunction parameter list as required. The REXX interpreter continues with processing as usual.

RXEXIT_NOT_HANDLED
The exit handler did not process the exit subfunction. The REXX interpreter processes the subfunction as if the exit handler were not called.

RXEXIT_RAISE_ERROR
A fatal error occurred in the exit handler. The REXX interpreter raises REXX error 48 (“Failure in system service”).

For example, if an application creates an input/output exit handler, one of the following happens:

- When the exit handler returns RXEXIT_NOT_HANDLED for an RXSIOSAY subfunction, the REXX interpreter writes the output line to STDOUT.
- When the exit handler returns RXEXIT_HANDLED for an RXSIOSAY subfunction, the REXX interpreter assumes the exit handler has handled all required output. The interpreter does not write the output line to STDOUT.
- When the exit handler returns RXEXIT_RAISE_ERROR for an RXSIOSAY subfunction, the interpreter raises REXX error 48, “Failure in system service”.

Exit Parameters

Each exit subfunction has a different parameter list. All RXSTRING exit subfunction parameters are passed as null-terminated RXSTRINGs. The RXSTRING value can also contain null characters.

For some exit subfunctions, the exit handler can return an RXSTRING character result in the parameter list. The interpreter provides a default 256-byte RXSTRING for the result string. If the result is longer than 256 bytes, a new RXSTRING can be allocated using `malloc(size)`. The REXX interpreter returns the RXSTRING storage for the exit handler.

Identifying Exit Handlers to REXX

System exit handlers must be registered with RexxRegisterExitDll or RexxRegisterExitExe. The system exit handler registration is similar to the subcommand handler registration.

The REXX system exits are enabled with the RexxStart function parameter *Exits*. *Exits* is a pointer to an array of RXSYSEXIT structures. Each RXSYSEXIT structure in the array contains a REXX exit code and the address of an ASCII exit handler name. The RXENDLST exit code marks the exit list end.

```
typedef struct {
   PSZ             sysexit_name;       /* name of exit handler        */
   LONG            sysexit_code;       /* system exit function code   */
} RXSYSEXIT;
```

The REXX interpreter calls the registered exit handler named in *sysexit_name* for all of the *sysexit_code* subfunctions.

Example:

```
.
.
.
{
WORKAREARECORD  *user_info[2];         /* saved user information     */
RXSYSEXIT exit_list[2];                /* system exit list           */

  user_info[0] = global_workarea;      /* save global work area for  */
  user_info[1] = NULL;                 /* re-entrance                */

  rc = RexxRegisterExitExe("EditInit", /* register exit handler      */
      &Init_exit,                      /* located at this address    */
      user_info);                      /* save global pointer        */

                                       /* set up for RXINI exit      */
  exit_list[0].sysexit_name = "EditInit";
  exit_list[0].sysexit_code = RXINI;
  exit_list[1].sysexit_code = RXENDLST;

  return_code = RexxStart(1,           /* one argument               */
                          argv,        /* argument array             */
                          "CHANGE.ED", /* REXX procedure name        */
                          NULL,        /* use disk version           */
                          "Editor",    /* default address name       */
                          RXCOMMAND,   /* calling as a subcommand    */
                          exit_list,   /* exit list                  */
                          &rc,         /* converted return code      */
                          &retstr);    /* returned result            */

                                       /* process return value       */
.
.
.
}

LONG APIENTRY Init_exit(
     LONG  ExitNumber,    /* code defining the exit function    */
     LONG  Subfunction,   /* code defining the exit subfunction */
     PEXIT ParmBlock)     /* function dependent control block   */
{
  WORKAREARECORD  *user_info[2];       /* saved user information     */
  WORKAREARECORD   global_workarea;    /* application data anchor    */
  USHORT           query_flag;         /* flag for handler query     */
  rc = RexxQueryExit("EditInit",       /* retrieve application work  */
      NULL,                            /* area anchor from REXX      */
      &query_flag,
      user_info);
```

```
  global_workarea = user_info[0];      /* set the global anchor      */

  if (global_workarea->rexx_trace)     /* trace at start?            */
                                       /* turn on macro tracing      */
    RexxSetTrace(global_workarea->rexx_pid, global_workarea->rexx_tid);
  return RXEXIT_HANDLED;               /* successfully handled       */
}
```

System Exit Definitions

The REXX interpreter supports the following system exits:

RXFNC
External function call exit.

RXFNCCAL
Call an external function.

RXCMD
Subcommand call exit.

RXCMDHST
Call a subcommand handler.

RXMSQ
External data queue exit.

RXMSQPLL
Pull a line from the external data queue.

RXMSQPSH
Place a line in the external data queue.

RXMSQSIZ
Return the number of lines in the external data queue.

RXMSQNAM
Set the active external data queue name.

RXSIO
Standard input and output exit.

RXSIOSAY
Write a line to the standard output stream for the SAY instruction.

RXSIOTRC
Write a line to the standard error stream for the REXX trace or REXX error messages.

RXSIOTRD
Read a line from the standard input stream for PULL or PARSE PULL.

RXSIODTR
Read a line from the standard input stream for interactive debugging.

RXHLT
Halt processing exit.

RXHLTTST
Test for a HALT condition.

RXHLTCLR
Clear a HALT condition.

RXTRC
External trace exit.

RXTRCTST
Test for an external trace event.

RXINI Initialization exit.

RXINIEXT
Allow additional REXX procedure initialization.

RXTER
Termination exit.

RXTEREXT
Process REXX procedure termination.

The following sections describe each exit subfunction, including:

- The service the subfunction provides
- When REXX calls the exit handler
- The default action when the exit is not provided or the exit handler does not process the subfunction
- The exit action
- The subfunction parameter list
- The state of the variable pool interface during the exit handler call (the variable pool interface is fully enabled for the RXCMD, RXFNC, RXINI, and RXTER exit handler calls; the variable pool interface is enabled for RXSHV_EXIT requests for RXHLT, RXCMD, RXFNC, RXSIO, and RXMSQ exit handler calls)

RXFNC

Processes calls to external functions.

Note: The variable pool interface is fully enabled during calls to the RXFNC exit handler.

RXFNCCAL

Processes calls to external functions.

When called: When REXX calls an external subroutine or function.

Default action: Call the external routine using the usual external function search order.

Exit action: Call the external routine, if possible.

Continuation: If necessary, raise REXX error 40 ("Incorrect call to routine"), 43 ("Routine not found"), or 44 ("Function or message did not return data").

Parameter list:

```
typedef struct {
   struct {
      unsigned rxfferr  : 1;          /* Invalid call to routine.    */
      unsigned rxffnfnd : 1;          /* Function not found.         */
      unsigned rxffsub  : 1;          /* Called as a subroutine if   */
                                      /* TRUE.  Return values are    */
                                      /* optional for subroutines,   */
                                      /* required for functions.     */
   } rxfnc_flags ;

   PUCHAR            rxfnc_name;     /* Pointer to function name.   */
   USHORT            rxfnc_namel;    /* Length of function name.    */
   PUCHAR            rxfnc_que;      /* Current queue name.         */
   USHORT            rxfnc_quel;     /* Length of queue name.       */
   USHORT            rxfnc_argc;     /* Number of args in list.     */
   PRXSTRING         rxfnc_argv;     /* Pointer to argument list.   */
                                     /* List mimics argv list for   */
                                     /* function calls, an array of */
                                     /* RXSTRINGs.                  */
   RXSTRING          rxfnc_retc;     /* Return value.               */
} RXFNCCAL_PARM;
```

The name of the external function is defined by *rxfnc_name* and *rxfnc_namel*. The arguments to the function are in *rxfnc_argc* and *rxfnc_argv*. If you call the named external function with the REXX CALL instruction (rather than using a function call), the flag *rxffsub* is TRUE.

The exit handler can set *rxfnc_flags* to indicate whether the external function call was successful. If neither *rxfferr* nor *rxffnfnd* is TRUE, the exit handler successfully called the external function. The error flags are checked only when the exit handler handles the request.

The exit handler sets *rxffnfnd* to TRUE when the exit handler cannot locate the external function. The interpreter raises REXX error 43, "Routine not found". The exit handler sets *rxfferr* to TRUE when the

exit handler locates the external function, but the external function returned an error return code. The REXX interpreter raises error 40, "Incorrect call to routine."

The exit handler returns the external function result in the *rxfnc_retc* RXSTRING. The REXX interpreter raises error 44, "Function or method did not return data," when the external routine is called as a function and the exit handler does not return a result. When the external routine is called with the REXX CALL instruction, a result is not required.

RXCMD

Processes calls to subcommand handlers.

Note: The variable pool interface function is fully enabled during calls to the RXCMD exit handlers.

RXCMDHST

Calls a named subcommand handler.

When called: When REXX procedure issues a command.

Default action: Call the named subcommand handler specified by the current REXX ADDRESS setting.

Exit action: Process the call to a named subcommand handler.

Continuation: Raise the ERROR or FAILURE condition when indicated by the parameter list flags.

Parameter list:

```
typedef struct {
   struct {                             /* Condition flags             */
      unsigned rxfcfail : 1;            /* Command failed.  Trap with  */
                                        /* CALL or SIGNAL on FAILURE.  */
      unsigned rxfcerr  : 1;            /* Command ERROR occurred.     */
                                        /* Trap with CALL or SIGNAL on */
                                        /* ERROR.                      */
   } rxcmd_flags;
   PUCHAR            rxcmd_address;     /* Pointer to address name.    */
   USHORT            rxcmd_addressl;    /* Length of address name.     */
   PUCHAR            rxcmd_dll;         /* Library name for command.   */
   USHORT            rxcmd_dll_len;     /* Length of library name.     */
                                        /* 0 ==> executable file.             */
   RXSTRING          rxcmd_command;     /* The command string.         */
   RXSTRING          rxcmd_retc;        /* Pointer to return code      */
                                        /* buffer.  User allocated.    */
} RXCMDHST_PARM;
```

The *rxcmd_command* field contains the issued command. *Rxcmd_address*, *rxcmd_addressl*, *rxcmd_dll*, and *rxcmd_dll_len* fully define the current

ADDRESS setting. *Rxcmd_retc* is an RXSTRING for the return code value assigned to REXX special variable RC.

The exit handler can set *rxfcfail* or *rxfcerr* to TRUE to raise an ERROR or FAILURE condition.

RXMSQ

External data queue exit.

RXMSQPLL

Pulls a line from the external data queue.

When called: When a REXX PULL instruction, PARSE PULL instruction, or LINEIN built-in function reads a line from the external data queue.

Default action: Remove a line from the current REXX data queue.

Exit action: Return a line from the data queue that the exit handler provided.

Parameter list:

```
typedef struct {
   RXSTRING          rxmsq_retc;       /* Pointer to dequeued entry   */
                                 /* buffer.  User allocated.    */
} RXMSQPLL_PARM;
```

The exit handler returns the queue line in the *rxmsq_retc* RXSTRING.

RXMSQPSH

Places a line in the external data queue.

When called: When a REXX PUSH instruction, QUEUE instruction, or LINEOUT built-in function adds a line to the data queue.

Default action: Add the line to the current REXX data queue.

Exit action: Add the line to the data queue that the exit handler provided.

Parameter list:

```
typedef struct {
   struct {                            /* Operation flag              */
      unsigned rxfmlifo : 1;           /* Stack entry LIFO when TRUE, */
                                       /* FIFO when FALSE.            */
   } rxmsq_flags;
   RXSTRING          rxmsq_value;      /* The entry to be pushed.     */
} RXMSQPSH_PARM;
```

The *rxmsq_value* RXSTRING contains the line added to the queue. It is the responsibility of the exit handler to truncate the string if the exit handler data queue has a maximum length restriction. *Rxfmlifo* is the stacking order (LIFO or FIFO).

RXMSQSIZ
Returns the number of lines in the external data queue.

When called: When the REXX QUEUED built-in function requests the size of the external data queue.

Default action: Request the size of the current REXX data queue.

Exit action: Return the size of the data queue that the exit handler provided.

Parameter list:

```
typedef struct {
   ULONG            rxmsq_size;       /* Number of Lines in Queue    */
} RXMSQSIZ_PARM;
```

The exit handler returns the number of queue lines in *rxmsq_size*.

RXMSQNAM
Sets the name of the active external data queue.

When called: Called by the RXQUEUE("SET", *newname*) built-in function.

Default action: Change the current default queue to *newname*.

Exit action: Change the default queue name for the data queue that the exit handler provided.

Parameter list:

```
typedef struct {
   RXSTRING         rxmsq_name;       /* RXSTRING containing         */
                                      /* queue name.                 */
} RXMSQNAM_PARM;
```

rxmsq_name contains the new queue name.

RXSIO

Standard input and output.

Note: The PARSE LINEIN instruction and the LINEIN, LINEOUT, LINES, CHARIN, CHAROUT, and CHARS built-in functions do not call the RXSIO exit handler.

RXSIOSAY
Writes a line to the standard output stream.

When called: When the SAY instruction writes a line to the standard output stream.

Default action: Write a line to the standard output stream (STDOUT).

Exit action: Write a line to the output stream that the exit handler provided.

Parameter list:

```
typedef struct {
   RXSTRING          rxsio_string;     /* String to display.          */
} RXSIOSAY_PARM;
```

The output line is contained in *rxsio_string*. The output line can be of any length. It is the responsibility of the exit handler to truncate or split the line if necessary.

RXSIOTRC

Writes trace and error message output to the standard error stream.

When called: To output lines of trace output and REXX error messages.

Default action: Write a line to the standard error stream (.ERROR).

Exit action: Write a line to the error output stream that the exit handler provided.

Parameter list:

```
typedef struct {
   RXSTRING          rxsio_string;     /* Trace line to display.      */
} RXSIOTRC_PARM;
```

The output line is contained in *rxsio_string*. The output line can be of any length. It is the responsibility of the exit handler to truncate or split the line if necessary.

RXSIOTRD

Reads from standard input stream.

When called: To read from the standard input stream for the REXX PULL and PARSE PULL instructions.

Default action: Read a line from the standard input stream (STDIN).

Exit action: Return a line from the standard input stream that the exit handler provided.

Parameter list:

```
typedef struct {
   RXSTRING          rxsiotrd_retc;    /* RXSTRING for output.        */
} RXSIOTRD_PARM;
```

The input stream line is returned in the *rxsiotrd_retc* RXSTRING.

RXSIODTR

Interactive debug input.

When called: To read from the debug input stream for interactive debug prompts.

Default action: Read a line from the standard input stream (STDIN).

Exit action: Return a line from the standard debug stream that the exit handler provided.

Parameter list:

```
typedef struct {
   RXSTRING          rxsiodtr_retc;    /* RXSTRING for output.       */
} RXSIODTR_PARM;
```

The input stream line is returned in the *rxsiodtr_retc* RXSTRING.

RXHLT

HALT condition processing.

Note: Because the RXHLT exit handler is called after every REXX instruction, enabling this exit slows REXX program execution. The RexxSetHalt function can halt a REXX program without between-instruction polling.

RXHLTTST

Tests the HALT indicator.

When called: When the interpreter polls externally raises HALT conditions. The exit will be called after completion of every REXX instruction.

Default action: The interpreter uses the system facilities for trapping Cntrl-Break signals.

Exit action: Return the current state of the HALT condition (either TRUE or FALSE).

Continuation: Raise the REXX HALT condition if the exit handler returns TRUE.

Parameter list:

```
typedef struct {
   struct {                          /* Halt flag                  */
      unsigned rxfhhalt : 1;         /* Set if HALT occurred.      */
   } rxhlt_flags;
} RXHLTTST_PARM;
```

If the exit handler sets *rxfhhalt* to TRUE, the HALT condition is raised in the REXX program.

When the exit handler has set *rxfhhalt* to TRUE, it can also use the RXSHV_EXIT operation of RexxVariablePool to return a string describing the HALT condition reason. The REXX program can retrieve the reason string using the CONDITION("D") built-in function.

RXHLTCLR

Clears the HALT condition.

When called: When the interpreter has recognized and raised a HALT condition, to acknowledge processing of the HALT condition.

Default action: The interpreter resets the Cntrl-Break signal handlers.

Exit action: Reset exit handler HALT state to FALSE.

Parameters: None.

RXTRC

Tests the external trace indicator.

Note: Because the RXTRC exit is called after every REXX instruction, enabling this exit slows REXX procedure execution. The RexxSetTrace function can turn on REXX tracing without the between-instruction polling.

RXTRCTST

Tests the external trace indicator.

When called: When the interpreter polls for an external trace event. The exit is called after completion of every REXX instruction.

Default action: None.

Exit action: Return the current state of external tracing (either TRUE or FALSE).

Continuation: When the exit handler switches from FALSE to TRUE, the REXX interpreter enters the interactive REXX debug mode using TRACE ?R level of tracing. When the exit handler switches from TRUE to FALSE, the REXX interpreter exits the interactive debug mode.

Parameter list:

```
typedef struct {
   struct {
      unsigned rxftrace : 1;        /* External trace setting        */
   } rxtrc_flags;
} RXTRCTST_PARM;
```

If the exit handler switches *rxftrace* to TRUE, REXX switches on the interactive debug mode. If the exit handler switches *rxftrace* to FALSE, REXX switches off the interactive debug mode.

RXINI

Initialization processing. This exit is called as the last step of REXX program initialization.

Note: The variable pool interface is fully enabled for this exit.

RXINIEXT

Initialization exit.

When called: Before the first instruction of the REXX procedure is interpreted.

Default action: None.

Exit action: The exit handler can perform additional initialization. For example:

– Use RexxVariablePool to initialize application-specific variables.

– Use RexxSetTrace to switch on the interactive REXX debug mode.

Parameters: None.

RXTER

Termination processing.

The RXTER exit is called as the first step of REXX program termination.

Note: The variable pool interface is fully enabled for this exit.

RXTEREXT
Termination exit.

When called: After the last instruction of the REXX procedure has been interpreted.

Default action: None.

Exit action: The exit handler can perform additional termination activities. For example, the exit handler can use RexxVariablePool to retrieve the REXX variable values.

Parameters: None.

System Exit Interface Functions

The system exit functions are similar to the subcommand handler functions. The system exit functions are:

RexxRegisterExitDll

RexxRegisterExitDll registers an exit handler that resides in a library routine.

RexxRegisterExitDll(ExitName, ModuleName, EntryPoint, UserArea, DropAuth)

Parameters:

ExitName *(PSZ) - input*
is the address of an ASCII exit handler name.

ModuleName *(PSZ) - input*
is the address of an ASCII library name. *ModuleName* is the library file containing the exit handler routine.

EntryPoint *(PSZ) - input*
is the address of an ASCII procedure name. *EntryPoint* is the routine within *ModuleName* that REXX calls as an exit handler.

UserArea *(PUCHAR) - input*
is the address of an 8-byte area of user-defined information. The 8 bytes *UserArea* addresses are saved with the exit handler registration. *UserArea* can be null if there is no user information to be saved. The RexxQueryExit function can retrieve the saved user information.

DropAuth *(ULONG) - input*
is the drop authority. *DropAuth* identifies the processes that can deregister the exit handler. Possible *DropAuth* values are:

RXEXIT_DROPPABLE
Any process can deregister the exit handler with RexxDeregisterExit.

RXEXIT_NONDROP
Only a thread within the same process as the thread that registered the handler can deregister the handler with RexxDeregisterExit.

Return Codes:

0 RXEXIT_OK

10 RXEXIT_DUP

1002 RXEXIT_NOEMEM

Remarks: *EntryPoint* can be only a 32-bit routine.

RexxRegisterExitExe

RexxRegisterExitExe registers an exit handler that resides within the application code.

RexxRegisterExitExe(ExitName, EntryPoint, UserArea)

Parameters:

ExitName *(PSZ) - input*
is the address of an ASCII exit handler name.

EntryPoint *(PFN) - input*
is the address of the exit handler entry point within the application executable file.

UserArea *(PUCHAR) - input*
is the address of an 8-byte area of user-defined information. The 8 bytes *UserArea* addresses are saved with the exit handler registration. *UserArea*

can be null if there is no user information to be saved. The RexxQueryExit function can retrieve the user information.

Return Codes:

0 RXEXIT_OK

10 RXEXIT_DUP

30 RXEXIT_NOTREG

1002 RXEXIT_NOEMEM

Remarks: If *ExitName* has the same name as a handler registered with RexxRegisterExitDll, RexxRegisterExitExe returns RXEXIT_DUP, which means that the new exit handler has been properly registered.

Example:

```
WORKAREARECORD  *user_info[2];       /* saved user information     */

user_info[0] = global_workarea;      /* save global work area for  */
user_info[1] = NULL;                 /* re-entrance                */

rc = RexxRegisterExitExe("IO_Exit",  /* register editor handler    */
    &Edit_IO_Exit,                   /* located at this address    */
    user_info);                      /* save global pointer        */
```

RexxDeregisterExit

RexxDeregisterExit deregisters an exit handler.

RexxDeregisterExit(ExitName, ModuleName)

Parameters:

ExitName *(PSZ) - input*
is the address of an ASCII exit handler name.

ModuleName *(PSZ) - input*
is the address of an ASCII library name. *ModuleName* restricts the query to an exit handler within the *ModuleName* library. When *ModuleName* is null, RexxDeregisterExit searches the RexxRegisterExitExe exit handler list for a handler within the current process. If RexxDeregisterExit does not find a RexxRegisterExitExe handler, it searches the RexxRegisterExitDll exit handler list.

Return Codes:

0 RXEXIT_OK

30 RXEXIT_NOTREG

40 RXEXIT_NOCANDROP

Remarks: The handler is removed from the exit handler list.

RexxQueryExit

RexxQueryExit queries an exit handler and retrieves saved user information.

RexxQueryExit(ExitName, ModuleName, Flag, UserWord)

Parameters:

ExitName *(PSZ) - input*
is the address of an ASCII exit handler name.

ModuleName *(PSZ) - input*
restricts the query to an exit handler within the *ModuleName* library. When *ModuleName* is null, RexxQueryExit searches the RexxRegisterExitExe exit handler list for a handler within the current process. If RexxQueryExit does not find a RexxRegisterExitExe handler, it searches the RexxRegisterExitDll exit handler list.

Flag *(PUSHORT) - output*
is the *ExitName* exit handler registration status. When RexxQueryExit returns RXEXIT_OK, the *ExitName* exit handler is currently registered. When RexxQueryExit returns RXEXIT_NOTREG, the *ExitName* exit handler is not registered.

UserWord *(PUCHAR) - output*
is the address of an 8-byte area to receive the user information saved with RexxRegisterExitExe or RexxRegisterExitDll. *UserWord* can be null if the saved user information is not required.

Return Codes:

0 RXEXIT_OK

30 RXEXIT_NOTREG

Example:

```
ULONG APIENTRY Edit_IO_Exit(
  PRXSTRING Command,    /* Command string passed from the caller    */
  PUSHORT   Flags,      /* pointer too short for return of flags    */
  PRXSTRING Retstr)     /* pointer to RXSTRING for RC return        */
{
  WORKAREARECORD  *user_info[2];       /* saved user information     */
  WORKAREARECORD   global_workarea;    /* application data anchor    */
  USHORT           query_flag;         /* flag for handler query     */


  rc = RexxQueryExit("IO_Exit",        /* retrieve application work  */
```

```
        NULL,                           /* area anchor from REXX.     */
        &query_flag,
        user_info);

  global_workarea = user_info[0];       /* set the global anchor      */
.
.
.
}
```

Return Codes

RXEXIT_OK	0	The system exit function executed successfully.
RXEXIT_DUP	10	A duplicate handler name has been successfully registered. There is either an executable handler with the same name registered in another process, or a library handler with the same name registered in another library. (To address this exit handler, you must specify its library name.)
RXEXIT_NOTREG	30	Registration was unsuccessful due to duplicate handler and library names (RexxRegisterExitExe or RexxRegisterExitDll); the exit handler is not registered (other REXX exit handler functions).
RXEXIT_NOCANDROP	40	The exit handler has been registered as “not droppable.”
RXEXIT_NOEMEM	1002	There is insufficient memory available to complete this request.

Variable Pool Interface

Application programs can use the REXX Variable Pool Interface to manipulate the variables of a currently active REXX procedure.

Interface Types

Three of the Variable Pool Interface functions (set, fetch, and drop) have dual interfaces.

Symbolic Interface

The symbolic interface uses normal REXX variable rules when interpreting variables. Variable names are valid REXX symbols (in mixed case if desired) including compound symbols. Compound symbols are referenced with tail substitution. The functions that use the symbolic interface are RXSHV_SYSET, RXSHV_SYFET, and RXSHV_SYDRO.

Direct Interface

The direct interface uses no substitution or case translation. Simple symbols must be valid REXX variable names. A valid REXX variable name:

- Does not begin with a digit or period.
- Contains only uppercase A to Z, the digits 0 - 9, or the characters _, ! or ? before the first period of the name.
- Can contain any characters after the first period of the name.

Compound variables are specified using the derived name of the variable. Any characters (including blanks) can appear after the first period of the name. No additional variable substitution is used. RXSHV_SET, RXSHV_FETCH, and RXSHV_DROP use the direct interface.

RexxVariablePool Restrictions

Only the main thread of an application can access the REXX variable pool. Applications can create and use new threads, but only the original thread that called RexxStart can use RexxVariablePool.

Executable Linux modules called from a REXX procedure execute in a new process. Because the modules do not use the same process and thread as the REXX procedure, the modules cannot use RexxVariablePool to access REXX variables. You can use RexxVariablePool from subcommand handlers, external functions, and exit handlers.

RexxVariablePool Interface Function

REXX procedure variables are accessed using the RexxVariablePool function.

RexxVariablePool

RexxVariablePool accesses variables of a currently active REXX procedure.

RexxVariablePool(RequestBlockList)

Parameters:

RequestBlockList *(PSHVBLOCK) - input*
is a linked list of shared variable request blocks (SHVBLOCK). Each block is a separate variable access request.

The SHVBLOCK has the following form:

```
typedef struct shvnode {
    struct shvnode    *shvnext;
    RXSTRING           shvname;
    RXSTRING           shvvalue;
    ULONG              shvnamelen;
    ULONG              shvvaluelen;
    UCHAR              shvcode;
    UCHAR              shvret;
}   SHVBLOCK;
```

where:

shvnext
is the address of the next SHVBLOCK in the request list. *shvnext* is null for the last request block.

shvname
is an RXSTRING containing a REXX variable name. *shvname* usage varies with the SHVBLOCK request code:

RXSHV_SET

RXSHV_SYSET

RXSHV_FETCH

RXSHV_SYFET

RXSHV_DROPV

RXSHV_SYDRO

RXSHV_PRIV
shvname is an RXSTRING pointing to the name of the REXX variable that the shared variable request block accesses.

RXSHV_NEXTV
shvname is an RXSTRING defining an area of storage to receive the name of the next variable. *shvnamelen* is the length of the RXSTRING area. If the variable name is longer than the *shvnamelen* characters, the name is truncated and the RXSHV_TRUNC bit of *shvret* is set. On return, *shvname.strlength* contains the length of the variable name; *shvnamelen* remains unchanged.

If *shvname* is an empty RXSTRING (*strptr* is null), the REXX interpreter allocates and returns an RXSTRING to hold the variable name. If the REXX interpreter allocates the RXSTRING, an RXSHV_TRUNC condition cannot occur. However, RXSHV_MEMFL errors are possible for these operations. If an RXSHV_MEMFL condition occurs, memory is not allocated for that request block. The RexxVariablePool caller must release the storage with `free(ptr)`.

Note: The RexxVariablePool does not add a terminating null character to the variable name.

RXSHV_EXIT
shvname is unused for the RXSHV_EXIT function.

shvvalue

An RXSTRING containing a REXX variable value. The meaning of *shvvalue* varies with the SHVBLOCK request code:

RXSHV_SET

RXSHV_SYSET
shvvalue is the value assigned to the REXX variable in *shvname*. *shvvaluelen* contains the length of the variable value.

RXSHV_EXIT
shvvalue is the value assigned to the exit handler return value. *shvvaluelen* contains the length of the variable value.

RXSHV_FETCH

RXSHV_SYFET

RXSHV_PRIV

RXSHV_NEXT
shvvalue is a buffer the REXX interpreter uses to return a copy of the REXX variable *shvname*. *shvvaluelen* contains the length of the value buffer. On return, *shvvalue.strlength* is set to the length of the returned value but *shvvaluelen* remains unchanged. If the variable value is longer than the *shvvaluelen* characters, the value is truncated and the RXSHV_TRUNC bit of *shvret* is set. On return, *shvvalue.strlength* is set to the length of the returned value; *shvvaluelen* remains unchanged.

If *shvvalue* is an empty RXSTRING (*strptr* is null), the REXX interpreter allocates and returns an RXSTRING to hold the variable value. If the REXX interpreter allocates the RXSTRING, an RXSHV_TRUNC condition cannot occur. However, RXSHV_MEMFL errors are possible for these operations. If an RXSHV_MEMFL condition occurs, memory is not allocated for that request block. The RexxVariablePool caller must release the storage with `free(ptr)`.

Note: The RexxVariablePool does not add a terminating null character to the variable value.

RXSHV_DROPV

RXSHV_SYDRO
shvvalue is not used.

shvcode

The shared variable block request code. Valid request codes are:

RXSHV_SET

RXSHV_SYSET
Assign a new value to a REXX procedure variable.

RXSHV_FETCH

RXSHV_SYFET
Retrieve the value of a REXX procedure variable.

RXSHV_DROPV

RXSHV_SYDRO
Drop (unassign) a REXX procedure variable.

RXSHV_PRIV
Fetch the private information of the REXX procedure. The following information items can be retrieved by name:

EXITNAME
The name of the current system exit handler for this thread. If not called from within an exit handler, a null string is returned.

PARM
The number of arguments supplied to the REXX procedure. The number is formatted as a character string.

PARM.n
The nth argument string to the REXX procedure. If the nth argument was not supplied to the procedure (either omitted or fewer than n parameters were specified), a null string is returned.

QUENAME
The current REXX data queue name.

SOURCE
The REXX procedure source string used for the PARSE SOURCE instruction.

VERSION
The REXX interpreter version string used for the PARSE SOURCE instruction.

RXSHV_NEXTV
Fetch the next variable, excluding variables hidden by PROCEDURE instructions. The variables are not returned in any specified order.

The REXX interpreter maintains an internal pointer to its list of variables. The variable pointer is reset to the first REXX variable whenever:

- An external program returns control to the interpreter

- A set, fetch, or drop RexxVariablePool function is used

RXSHV_NEXTV returns both the name and the value of REXX variables until the end of the variable list is reached. If no REXX variables are left to return, RexxVariablePool sets the RXSHV_LVAR bit in *shvret*.

RXSHV_EXIT
Set a return value for an external function or system exit call. RXSHV_EXIT is valid only from external functions or system exit events that return a string value. An external function or exit handler can use RXSHV_EXIT only once.

shvret The individual shared variable request return code. *shvret* is a 1-byte field of status flags for the individual shared variable request. The *shvret* fields for all request blocks in the list are ORed together to form the RexxVariablePool return code. The individual status conditions are:

RXSHV_OK
The request was processed without error (all flag bits are FALSE).

RXSHV_NEWV
The named variable was uninitialized at the time of the call.

RXSHV_LVAR
No more variables are available for an RXSHV_NEXTV operation.

RXSHV_TRUNC
A variable value or variable name was truncated because the supplied RXSTRING was too small for the copied value.

RXSHV_BADN
The variable name specified in *shvname* was invalid for the requested operation.

RXSHV_MEMFL
The REXX interpreter was unable to obtain the storage required to complete the request.

RXSHV_BADF
The shared variable request block contains an invalid function code.

The REXX interpreter processes each request block in the order provided. RexxVariablePool returns to the caller after the last block is processed or a severe error occurred (such as an out-of-memory condition).

The RexxVariablePool function return code is a composite return code for the entire set of shared variable requests. The return codes for all of the

individual requests are ORed together to form the composite return code. Individual shared variable request return codes are returned in the shared variable request blocks.

RexxVariablePool Return Codes:

0 to 127
RexxVariablePool has processed the entire shared variable request block list.

The RexxVariablePool function return code is a composite return code for the entire set of shared variable requests. The low-order 6 bits of the *shvret* fields for all request blocks are ORed together to form the composite return code. Individual shared variable request status flags are returned in the shared variable request block *shvret* field.

RXSHV_NOAVL
The variable pool interface was not enabled when the call was issued.

Example:

```
/*********************************************************************/
/*                                                                   */
/* SetRexxVariable - Set the value of a REXX variable                */
/*                                                                   */
/*********************************************************************/

INT SetRexxVariable(
  PSZ        name,                   /* REXX variable to set       */
  PSZ        value)                  /* value to assign            */
{
  SHVBLOCK   block;                  /* variable pool control block*/

  block.shvcode = RXSHV_SYSET;       /* do a symbolic set operation*/
  block.shvret=(UCHAR)0;             /* clear return code field    */
  block.shvnext=(PSHVBLOCK)0;        /* no next block              */
                                     /* set variable name string   */
  MAKERXSTRING(block.shvname, name, strlen(name));
                                     /* set value string           */
  MAKERXSTRING(block.shvvalue, value, strlen(value));
  block.shvvaluelen=strlen(value);   /* set value length           */
  return RexxVariablePool(&block);   /* set the variable           */
}
```

Queue Interface

Application programs can use the REXX Queue Interface to establish and manipulate named queues. Named queues prevent different REXX programs that are running in a single session from interfering with each other. Named queues also allow REXX programs running in different sessions to

synchronize execution and pass data. These queuing services are entirely separate from the Linux InterProcess Communications queues.

Queue Interface Functions

The following sections explain the functions for creating and using named queues.

RexxCreateQueue

RexxCreateQueue creates a new (empty) queue.

RexxCreateQueue(Buffer, BuffLen, RequestedName, DupFlag)

Parameters:

Buffer *(PSZ) - input*
is the address of the buffer where the ASCII name of the created queue is returned.

BuffLen *(ULONG) - input*
is the size of the buffer.

RequestedName *(PSZ) - input*
is the address of an ASCII queue name. If no queue of that name exists, a queue is created with the requested name. If the name already exists, a queue is created, but REXX assigns an arbitrary name to it. In addition, the *DupFlag* is set. The maximum length for a queue name is 1024 characters.

When *RequestedName* is null, REXX provides a name for the created queue.

In all cases, the actual queue name is passed back to the caller.

DupFlag *(PULONG) - output*
is the duplicate name indicator. This flag is set when the requested name already exists.

Return Codes:

0 RXQUEUE_OK

1 RXQUEUE_STORAGE

5 RXQUEUE_BADQNAME

Remarks: Queue names must conform to the same syntax rules as REXX variable names. Lowercase characters in queue names are translated to uppercase.

The queue name must be a valid REXX symbol. However, there is no connection between queue names and variable names. A program can have a variable and a queue with the same name.

RexxDeleteQueue

RexxDeleteQueue deletes a queue.

RexxDeleteQueue(QueueName)

Parameters:

QueueName *(PSZ) - input*
is the address of the ASCII name of the queue to be deleted.

Return Codes:

0 RXQUEUE_OK

5 RXQUEUE_BADQNAME

9 RXQUEUE_NOTREG

10 RXQUEUE_ACCESS

Remarks: If a queue is busy (for example, wait is active), it is not deleted.

RexxQueryQueue

RexxQueryQueue returns the number of entries remaining in the named queue.

RexxQueryQueue(QueueName, Count)

Parameters:

QueueName *(PSZ) - input*
is the address of the ASCII name of the queue to be queried.

Count *(PULONG) - output*
is the number of entries in the queue.

Return Codes:

0 RXQUEUE_OK

5 RXQUEUE_BADQNAME

9 RXQUEUE_NOTREG

RexxAddQueue

RexxAddQueue adds an entry to a queue.

RexxAddQueue(QueueName, EntryData, AddFlag)

Parameters:

QueueName *(PSZ) - input*
is the address of the ASCII name of the queue to which data is to be added.

EntryData *(PRXSTRING) - input*
is the address of an RXSTRING containing the data to be added to the queue.

AddFlag *(ULONG) - input*
is the LIFO/FIFO flag. When *AddFlag* is `RXQUEUE_LIFO`, data is added LIFO (Last In, First Out) to the queue. When *AddFlag* is `RXQUEUE_FIFO`, data is added FIFO (First In, First Out).

Return Codes:

0 RXQUEUE_OK

5 RXQUEUE_BADQNAME

6 RXQUEUE_PRIORITY

9 RXQUEUE_NOTREG

12 RXQUEUE_MEMFAIL

RexxPullQueue

RexxPullQueue removes the top entry from the queue and returns it to the caller.

RexxPullQueue(QueueName, DataBuf, DateTime, WaitFlag)

Parameters:

QueueName *(PSZ) - input*
is the address of the ASCII name of the queue from which data is to be pulled.

DataBuf *(PRXSTRING) - output*
is the address of an RXSTRING for the returned value.

DateTime *(PDATETIME) - output*
is the address of the entry's date and time stamp.

WaitFlag *(ULONG) - input*
is the wait flag. When *WaitFlag* is `RXQUEUE_NOWAIT` and the queue is empty, RXQUEUE_EMPTY is returned. Otherwise, when *WaitFlag* is `RXQUEUE_WAIT`, REXX waits until a queue entry is available and returns that entry to the caller.

Return Codes:

0 RXQUEUE_OK

5 RXQUEUE_BADQNAME

7 RXQUEUE_BADWAITFLAG

8 RXQUEUE_EMPTY

9 RXQUEUE_NOTREG

12 RXQUEUE_MEMFAIL

Remarks: The caller is responsible for freeing the returned memory that *DataBuf* points to.

Return Codes

RXQUEUE_OK	0	The system queue function completed successfully.
RXQUEUE_STORAGE	1	The name buffer is not large enough for the queue name.
RXQUEUE_BADQNAME	5	The queue name is not valid, or you tried to create or delete a queue named SESSION.
RXQUEUE_PRIORITY	6	The order flag is not equal to RXQUEUE_LIFO or RXQUEUE_FIFO.
RXQUEUE_BADWAITFLAG	7	The wait flag is not equal to RXQUEUE_WAIT or RXQUEUE_NOWAIT.
RXQUEUE_EMPTY	8	Attempted to pull the item off the queue but it was empty.
RXQUEUE_NOTREG	9	The queue does not exist.
RXQUEUE_ACCESS	10	The queue cannot be deleted because it is busy.
RXQUEUE_MEMFAIL	12	There is insufficient memory available to complete the request.

Halt and Trace Interface

The halt and trace functions raise a REXX HALT condition or change the REXX interactive debug mode while a REXX procedure is running. You might prefer these interfaces to the RXHLT and RXTRC system exits. The system exits require an additional call to an exit routine after each REXX instruction completes, possibly causing a noticeable performance degradation. The Halt

and Trace functions, on the contrary, are a single request to change the halt or trace state and do not degrade the REXX procedure performance.

Halt and Trace Interface Functions

The Halt and Trace functions are:

RexxSetHalt

RexxSetHalt raises a HALT condition in a running REXX program.

RexxSetHalt(ProcessId, ThreadId)

Parameters:

ProcessId *(PID) - input*
is the process ID of the target REXX procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId *(TID) - input*
is the thread ID of the target REXX procedure. *ThreadId* is the application thread that called the RexxStart function. If *ThreadId*=0, all the threads of the process are canceled.

Return Codes:

0 RXARI_OK

1 RXARI_NOT_FOUND

2 RXARI_PROCESSING_ERROR

Remarks: This call is not processed if the target REXX program is running with the RXHLT exit enabled.

RexxSetTrace

RexxSetTrace turns on the interactive debug mode for a REXX procedure.

RexxSetTrace(ProcessId, ThreadId)

Parameters:

ProcessId *(PID) - input*
is the process ID of the target REXX procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId *(TID) - input*
is the thread ID of the target REXX procedure. *ThreadId* is the application thread that called the RexxStart function. If *ThreadId*=0, all the threads of the process are traced.

Return Codes:

0 RXARI_OK

1 RXARI_NOT_FOUND

2 RXARI_PROCESSING_ERROR

Remarks: A RexxSetTrace call is not processed if the REXX procedure is using the RXTRC exit.

RexxResetTrace

RexxResetTrace turns off the interactive debug mode for a REXX procedure.

RexxResetTrace(ProcessId,ThreadId)

Parameters:

ProcessId *(PID) - input*
is the process ID of the target REXX procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId *(TID) - input*
is the thread ID of the target REXX procedure. *ThreadId* is the application thread that called the RexxStart function. If *ThreadId*=0, the trace of all threads of the process is reset.

Return Codes:

0 RXARI_OK

1 RXARI_NOT_FOUND

2 RXARI_PROCESSING_ERROR

Remarks:

- A RexxResetTrace call is not processed if the REXX procedure uses the RXTRC exit.
- Interactive debugging is not turned off unless the interactive debug mode was originally started with RexxSetTrace.

Return Codes

RXARI_OK	0	The function completed successfully.
RXARI_NOT_FOUND	1	The target REXX procedure was not found.
RXARI_PROCESSING_ERROR	2	A failure in REXX processing occurred.

Macrospace Interface

The macrospace can improve the performance of REXX procedures by maintaining REXX procedure images in memory for immediate load and execution. This is useful for frequently-used procedures and functions such as editor macros.

Programs registered in the REXX macrospace are available to all processes. You can run them by using the RexxStart function or calling them as functions or subroutines from other REXX procedures.

Procedures in the macrospace are called in the same way as other REXX external functions. However, the macrospace REXX procedures can be placed at the front or at the very end of the external function search order.

Procedures in the macrospace are stored without source code information and therefore cannot be traced.

REXX procedures in the macrospace can be saved to a disk file. A saved macrospace file can be reloaded with a single call to RexxLoadMacroSpace. An application, such as an editor, can create its own library of frequently-used functions and load the entire library into memory for fast access. Several macrospace libraries can be created and loaded.

Search Order

When RexxAddMacro loads a REXX procedure into the macrospace, the position in the external function search order is specified. Possible values are:

RXMACRO_SEARCH_BEFORE
The REXX interpreter locates a function registered with RXMACRO_SEARCH_BEFORE before any registered functions or external REXX files.

RXMACRO_SEARCH_AFTER
The REXX interpreter locates a function registered with RXMACRO_SEARCH_AFTER after any registered functions or external REXX files.

Storage of Macrospace Libraries

The REXX macrospace is placed in shared memory. Only the amount of memory and swap space available to the system limits the size of the macrospace. However, as the macrospace grows, it limits the memory available to other processes in the system. Allowing the macrospace to grow

too large can degrade overall system performance due to increased system swap file access. Try to place only the most frequently-used functions in the macrospace.

Macrospace Interface Functions

The functions to manipulate macrospaces are:

RexxAddMacro

RexxAddMacro loads a REXX procedure into the macrospace.

RexxAddMacro(FuncName, SourceFile, Position)

Parameters:

FuncName *(PSZ) - input*
is the address of the ASCII function name. REXX procedures in the macrospace are called using the assigned function name.

SourceFile *(PSZ) - input*
is the address of the ASCII file specification for the REXX procedure source file. When a file extension is not supplied, **.cmd** is used. When the full path is not specified, the current directory and path are searched.

Position *(ULONG) - input*
is the position in the REXX external function search order. Possible values are:

RXMACRO_SEARCH_BEFORE
The REXX interpreter locates the function before any registered functions or external REXX files.

RXMACRO_SEARCH_AFTER
The REXX interpreter locates the function after any registered functions or external REXX files.

Return Codes:

0 RXMACRO_OK

1 RXMACRO_NO_STORAGE

7 RXMACRO_SOURCE_NOT_FOUND

8 RXMACRO_INVALID_POSITION

RexxDropMacro

RexxDropMacro removes a REXX procedure from the macrospace.

RexxDropMacro(FuncName)

Parameter:

FuncName *(PSZ) - input*
is the address of the ASCII function name.

Return Codes:

0 RXMACRO_OK

2 RXMACRO_NOT_FOUND

RexxClearMacroSpace

RexxClearMacroSpace removes all loaded REXX procedures from the macrospace.

RexxClearMacroSpace()

Return Codes:

0 RXMACRO_OK

2 RXMACRO_NOT_FOUND

Remarks: RexxClearMacroSpace must be used with care. This function removes all functions from the macrospace, including functions loaded by other processes.

RexxSaveMacroSpace

RexxSaveMacroSpace saves all or part of the macrospace REXX procedures to a disk file.

RexxSaveMacroSpace(FuncCount, FuncNames, MacroLibFile)

Parameters:

FuncCount *(ULONG) - input*
Number of REXX procedures to be saved.

FuncNames *(PSZ *) - input*
is the address of a list of ASCII function names. *FuncCount* gives the size of the function list.

MacroLibFile *(PSZ) - input*
is the address of the ASCII macrospace file name. If *MacroLibFile* already exists, it is replaced with the new file.

Return Codes:

0 RXMACRO_OK

2 RXMACRO_NOT_FOUND

3 RXMACRO_EXTENSION_REQUIRED

5 RXMACRO_FILE_ERROR

Remarks: When *FuncCount* is 0 or *FuncNames* is null, RexxSaveMacroSpace saves all functions in the macrospace.

Saved macrospace files can be used only with the same interpreter version that created the images. If RexxLoadMacroSpace is called to load a saved macrospace and the release level or service level is incorrect, RexxLoadMacroSpace fails. The REXX procedures must then be reloaded individually from the original source programs.

RexxLoadMacroSpace

RexxLoadMacroSpace loads all or part of the REXX procedures from a saved macrospace file.

RexxLoadMacroSpace(FuncCount, FuncNames, MacroLibFile)

Parameters:

FuncCount *(ULONG) - input*
is the number of REXX procedures to load from the saved macrospace.

FuncNames *(PSZ *) - input*
is the address of a list of REXX function names. *FuncCount* gives the size of the function list.

MacroLibFile *(PSZ) - input*
is the address of the saved macrospace file name.

Return Codes:

0 RXMACRO_OK

1 RXMACRO_NO_STORAGE

2 RXMACRO_NOT_FOUND

4 RXMACRO_ALREADY_EXISTS

5 RXMACRO_FILE_ERROR

6 RXMACRO_SIGNATURE_ERROR

Remarks: When *FuncCount* is 0 or *FuncNames* is null, RexxLoadMacroSpace loads all REXX procedures from the saved file.

If a RexxLoadMacroSpace call replaces an existing macrospace REXX procedure, the entire load request is discarded and the macrospace remains unchanged.

RexxQueryMacro

RexxQueryMacro searches the macrospace for a specified function.

RexxQueryMacro(FuncName, Position)

Parameters:

FuncName *(PSZ) - input*
is the address of an ASCII function name.

Position *(PUSHORT) - output*
is the address of an unsigned short integer flag. If the function is loaded in the macrospace, *Position* is set to the search-order position of the current function.

Return Codes:

0 RXMACRO_OK

2 RXMACRO_NOT_FOUND

RexxReorderMacro

RexxReorderMacro changes the search order position of a loaded macrospace function.

RexxReorderMacro(FuncName, Position)

Parameters:

FuncName *(PSZ) - input*
is the address of an ASCII macrospace function name.

Position *(ULONG) - input*
is the new search-order position of the macrospace function. Possible values are:

RXMACRO_SEARCH_BEFORE
The REXX interpreter locates the function before any registered functions or external REXX files.

RXMACRO_SEARCH_AFTER
The REXX interpreter locates the function after any registered functions or external REXX files.

Return Codes:

0 RXMACRO_OK

2 RXMACRO_NOT_FOUND

8 RXMACRO_INVALID_POSITION

Return Codes

Macrospace functions can return the following return codes. Each code indicates the cause of a failure in its respective function:

RXMACRO_OK	0	The call to the function completed successfully.
RXMACRO_NO_STORAGE	1	There was not enough memory to complete the requested function.
RXMACRO_NOT_FOUND	2	The requested function was not found in the macrospace.
RXMACRO_EXTENSION_REQUIRED	3	An extension is required for the macrospace file name.
RXMACRO_ALREADY_EXISTS	4	Duplicate functions cannot be loaded from a macrospace file.
RXMACRO_FILE_ERROR	5	An error occurred accessing a macrospace file.
RXMACRO_SIGNATURE_ERROR	6	A macrospace save file does not contain valid function images.
RXMACRO_SOURCE_NOT_FOUND	7	The requested file was not found.
RXMACRO_INVALID_POSITION	8	An invalid search-order position request flag was used.

Example

```
                                    /* first load entire package  */
RexxLoadMacroSpace(0, NULL, "EDITOR.MAC");

for (i = 0; i < MACRO_COUNT; i++) {    /* verify each macro          */
                                    /* if not there               */
  if (RexxQueryMacro(macro[i], &position))
                                    /* add to list                */
    RexxAddMacro(macro[i], macro_files[i],
                 RXMACRO_SEARCH_BEFORE);
}
                                    /* rebuild the macrospace     */
RexxSaveMacroSpace(0, NULL, "EDITOR.MAC");
```

```
:

                                    /* build the argument string  */
  MAKERXSTRING(argv[0], macro_argument,
      strlen(macro_argument));
                                    /* set up default return      */
  MAKERXSTRING(retstr, return_buffer, sizeof(return_buffer));
                                    /* set up for macrospace call */
  MAKERXSTRING(macrospace[0],NULL, 0);
  MAKERXSTRING(macrospace[1],NULL, 0);

  return_code = RexxStart(1,             /* one argument               */
                          argv,          /* argument array             */
                          macro[pos],    /* REXX procedure name        */
                          macrospace,    /* use macrospace version     */
                          "Editor",      /* default address name       */
                          RXCOMMAND,     /* calling as a subcommand    */
                          NULL,          /* no exits used              */
                            &rc,         /* converted return code      */
                          &retstr);      /* returned result            */
```

Appendix B. Sample REXX Programs

REXX supplies the following sample programs as **.cmd** files.

ccreply.cmd
A concurrent programming example.

This program demonstrates how to use reply to run two methods at the same time.

complex.cmd
A complex number class.

This program demonstrates how to create a complex number class using the ::CLASS and ::METHOD directives. An example of subclassing the complex number class (the Vector subclass) is also shown. Finally, the Stringlike class demonstrates the use of a mixin to provide to the complex number class with some string behavior.

factor.cmd
A factorial program.

This program demonstrates a way to define a factorial class using the subclass method and the *.methods* environment symbol.

greply.cmd
An example contrasting the GUARDED and UNGUARDED methods.

This program demonstrates the difference between GUARDED and UNGUARDED methods with respect to their use of the object variable pool.

guess.cmd
An animal guessing game.

This sample creates a simple node class and uses it to create a logic tree. The logic tree is filled in by playing a simple guessing game.

ktguard.cmd
A GUARD instruction example.

This program demonstrates the use of the START method and the GUARD instruction to control the running of several programs. In this sample, the programs are controlled by one "guarded" variable.

month.cmd
An example that displays the days of the month January 1994.

This version demonstrates the use of arrays to replace stems.

pipe.cmd
A pipeline implementation.

This program demonstrates the use of the ::CLASS and ::METHOD directives to create a simple implementation of a CMS-like pipeline function.

qdate.cmd
An example that types or pushes today's date and moon phase, in English date format.

qtime.cmd
An example that lays or stacks time in English time format, and also chimes.

semcls.cmd
An Object REXX semaphore class.

This file implements a semaphore class in Object REXX.

stack.cmd
A stack class.

This program demonstrates how to implement a stack class using the ::CLASS and ::METHOD directives. Also included is a short example of the use of a stack.

usecomp.cmd
A simple demonstration of the complex number class.

This program demonstrates the use of the ::REQUIRES directive, using the complex number class included in the samples.

usepipe.cmd
Sample uses of the pipe implementation in **pipe.rex**.

This program demonstrates how you could use the pipes implemented in the pipe sample.

Appendix C. 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 be incorporated in new editions of the publication. IBM may make

improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Deutschland
Informationssysteme GmbH
Department 3982
Pascalstrasse 100
70569 Stuttgart
Germany

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement or any equivalent agreement between us.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks and Service Marks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

IBM
OS/2

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries..

Other company, product, and service names may be trademarks or service marks of others.

Index

Special Characters

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Readers' Comments — We'd Like to Hear from You

Object REXX for Linux
Programming Guide
Version 1.2

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

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Deutschland Entwicklung GmbH
Information Development, Dept. 0446
Postfach 1380
71003 Boeblingen
Germany

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.