ooRexx Documentation 4.2

Open Object Rexx™

Programmer Guide

W. David Ashley

Rony G. Flatscher

Mark Hessling

Rick McGuire

Lee Peedin

Oliver Sims

Jon Wolfers

ooRexx Documentation 4.2 Open Object Rexx™ Programmer Guide
Edition 1

Author	W. David Ashley
Author	Rony G. Flatscher
Author	Mark Hessling
Author	Rick McGuire
Author	Lee Peedin
Author	Oliver Sims
Author	Jon Wolfers

Preface

This book describes the Open Object Rexx, or Object Rexx programming language. In the following, it is called Rexx unless compared to its traditional predecessor.

This book is aimed at developers 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.

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file `my_next_bestselling_novel` in your current working directory, enter the `cat my_next_bestselling_novel` command at the shell prompt and press `Enter` to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press `Enter` to execute the command.

> Press `Ctrl`+`Alt`+`F2` to switch to the first virtual terminal. Press `Ctrl`+`Alt`+`F1` to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in `mono-spaced bold`. For example:

---

[1] https://fedorahosted.org/liberation-fonts/

File-related classes include `filesystem` for file systems, `file` for files, and `dir` for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

`Mono-spaced Bold Italic` or **`Proportional Bold Italic`**

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type `ssh` *`username@domain.name`* at a shell prompt. If the remote machine is `example.com` and your username on that machine is john, type `ssh john@example.com`.

The `mount -o remount` *`file-system`* command remounts the named file system. For example, to remount the `/home` file system, the command is `mount -o remount /home`.

To see the version of a currently installed package, use the `rpm -q` *`package`* command. It will return a result as follows: *`package-version-release`*.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **`mono-spaced roman`** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **`mono-spaced roman`** but add syntax highlighting as follows:

```java
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. How to Read the Syntax Diagrams

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

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

    The **>>---** symbol indicates the beginning of a statement.

    The **--->** symbol indicates that the statement syntax is continued on the next line.

    The **>---** symbol indicates that a statement is continued from the previous line.

    The **--->** symbol indicates the end of a statement.

    Diagrams of syntactical units other than complete statements start with the **>---** symbol and end with the **--->** symbol.

- Required items appear on the horizontal line (the main path).

```
>>-STATEMENT--required_item------------------------------------><
```

- Optional items appear below the main path.

```
>>-STATEMENT--+---------------+--------------------------------><
              +-optional_item-+
```

- If you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path.

```
>>-STATEMENT--+-required_choice1-+-----------------------------><
              +-required_choice2-+
```

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

```
>>-STATEMENT--+-----------------+------------------------------><
              +-optional_choice1-+
              +-optional_choice2-+
```

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

```
              +-default_choice--+
>>-STATEMENT--+-----------------+------------------------------><
              +-optional_choice-+
```

```
                   +-optional_choice-+
```

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

```
              +----------------+
              V                |
  >>-STATEMENT----repeatable_item-+----------------------------><
```

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

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

```
  >>-STATEMENT--| fragment |-------------------------------------><
```

fragment:

```
  |--expansion_provides_greater_detail----------------------------|
```

- Keywords appear in uppercase (for example, **PARM1**). They must be spelled exactly as shown but you can type them in upper, lower, or mixed case. Variables appear in all lowercase letters (for example, **parmx**). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

The following example shows how the syntax is described:

```
         +-,------+
         V        |
  >>-MAX(----number-+--)----------------------------------------><
```

3. Getting Help and Submitting Feedback

The Open Object Rexx Project has a number of methods to obtain help and submit feedback for ooRexx and the extension packages that are part of ooRexx. These methods, in no particular order of preference, are listed below.

3.1. The Open Object Rexx SourceForge Site

The *Open Object Rexx Project*[2] utilizes *SourceForge*[3] to house the *ooRexx Project*[4] source repositories, mailing lists and other project features. Over time it has become apparent that the Developer and User mailing lists are better tools for carrying on discussions concerning ooRexx and that the Forums provided by SourceForge are cumbersome to use. The ooRexx user is most likely to get timely replies from one of the mailing lists.

Here is a list of some of the most useful facilities provided by SourceForge.

The Developer Mailing List

>   You can subscribe to the oorexx-devel mailing list at *ooRexx Mailing List Subscriptions*[5] page. This list is for discussing ooRexx project development activities and future interpreter enhancements. It also supports a historical archive of past messages.

The Users Mailing List

>   You can subscribe to the oorexx-users mailing list at *ooRexx Mailing List Subscriptions*[6] page. This list is for discussing using ooRexx. It also supports a historical archive of past messages.

The Announcements Mailing List

>   You can subscribe to the oorexx-announce mailing list at *ooRexx Mailing List Subscriptions*[7] page. This list is only used to announce significant ooRexx project events.

The Bug Mailing List

>   You can subscribe to the oorexx-bugs mailing list at *ooRexx Mailing List Subscriptions*[8] page. This list is only used for monitoring changes to the ooRexx bug tracking system.

Bug Reports

>   You can create a bug report at *ooRexx Bug Report*[9] page. Please try to provide as much information in the bug report as possible so that the developers can determine the problem as quickly as possible. Sample programs that can reproduce your problem will make it easier to debug reported problems.

Documentation Feedback

>   You can submit feedback for, or report errors in, the documentation at *ooRexx Documentation Report*[10] page. Please try to provide as much information in a documentation report as possible. In addition to listing the document and section the report concerns, direct quotes of the text will help the developers locate the text in the source code for the document. (Section numbers are generated when the document is produced and are not available in the source code itself.) Suggestions as to how to reword or fix the existing text should also be included.

Request For Enhancement

>   You can suggest ooRexx features at the *ooRexx Feature Requests*[11] page.

---

[2] http://www.oorexx.org/

[3] http://sourceforge.net/

[4] http://sourceforge.net/projects/oorexx

[5] http://sourceforge.net/mail/?group_id=119701

[6] http://sourceforge.net/mail/?group_id=119701

[7] http://sourceforge.net/mail/?group_id=119701

[8] http://sourceforge.net/mail/?group_id=119701

[9] http://sourceforge.net/tracker/?group_id=119701&atid=684730

[10] http://sourceforge.net/tracker/?group_id=119701&atid=1001880

[11] http://sourceforge.net/tracker/?group_id=119701&atid=684733

Patch Reports

> If you create an enhancement patch for ooRexx please post the patch using the *ooRexx Patch Report*[12] page. Please provide as much information in the patch report as possible so that the developers can evaluate the enhancement as quickly as possible.
>
> Please do not post bug fix patches here, instead you should open a bug report and attach the patch to it.

The ooRexx Forums

> The ooRexx project maintains a set of forums that anyone may contribute to or monitor. They are located on the *ooRexx Forums*[13] page. There are currently three forums available: Help, Developers and Open Discussion. In addition, you can monitor the forums via email.

3.2. The Rexx Language Association Mailing List

The *Rexx Language Association*[14] maintains a mailing list for its members. This mailing list is only available to RexxLA members thus you will need to join RexxLA in order to get on the list. The dues for RexxLA membership are small and are charged on a yearly basis. For details on joining RexxLA please refer to the *RexxLA Home Page*[15] or the *RexxLA Membership Application*[16] page.

3.3. comp.lang.rexx Newsgroup

The *comp.lang.rexx*[17] newsgroup is a good place to obtain help from many individuals within the Rexx community. You can obtain help on Open Object Rexx or on any number of other Rexx interpreters and tools.

4. Related Information

See also: *Open Object Rexx: Reference*

---

[12] http://sourceforge.net/tracker/?group_id=119701&atid=684732

[13] http://sourceforge.net/forum/?group_id=119701

[14] http://www.rexxla.org/

[15] http://rexxla.org/

[16] http://www.rexxla.org/rexxla/join.html

[17] http://groups.google.com/group/comp.lang.rexx/topics?hl=en

Meet Open Object Rexx (ooRexx)

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.

1.1. The Main Attractions

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

1.1.1. Object-Oriented Programming

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

• Classes, objects, and methods

• Messaging and polymorphism

• Inheritance and multiple inheritance

Object Rexx supplies the user with a base set of built-in classes providing many useful functions. Open Object Rexx is fully compatible with earlier versions of Rexx that were not object-oriented.

1.1.2. 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.

1.1.3. Cross-Platform Versatility

Versions of ooRexx are now available for a wide variety of platforms, and the programs you create with Object Rexx will run on any of these, including Linux™, AIX®, as well as Windows XP®, and Windows Vista®. It is also available in 64-bit versions that can exploit larger address spaces.

1.1.4. 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.

1.1.5. 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.

1.1.6. Built-In Classes and Functions

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

1.1.7. 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.

1.1.8. 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.

1.1.9. 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.

1.1.10. Impressive Development Tools

The ooRexx places many powerful tools at your disposal. These include a Rexx API to other languages like C/C++ or Cobol, OLE/ActiveX support, a mathematical functions package.

1.2. Rexx and the Operating System

The most important role Rexx plays is as a programming language for Windows and Unix-based systems. A Rexx program can serve as a script for the operating system. Using Rexx, you can reduce long, complex, or repetitious tasks to a single command or program.

1.3. 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.

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.
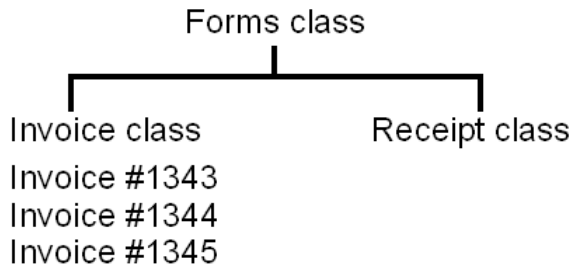
Figure 1.1. Objects in a Billing Application

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.

1.3.1. From Traditional Rexx to Object Rexx

In traditional (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 just one kind of object: the string. In object-oriented terminology, each string variable is a *object* that is an reference to an instance of the String class.

With ooRexx, variables can now reference objects other than strings. In addition to the String class, Rexx includes classes for creating arrays, queues, streams, and many other useful objects. Additionally, you can create your own classes that can interoperate seamlessly with the language built-in classes. Objects in these Rexx classes are manipulated by methods instead of traditional functions. To activate a method, you just send a message to the object.

For example, instead of using the SUBSTR function on a string variable Name, you send a SUBSTR message to the string object. In classic Rexx, you would do the following:

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

In Object Rexx, the equivalent would be:

```
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. Conceptually, you are "asking" the String object refered to by variable NAME to give you its substring starting at character 2 for 3 characters. Many String operations are available as both class methods and built-in functions, but the String class also provides many method enhancements for which there are no corresponding built-in functions.

For classes other than String (such as Array or Queue), methods are provided, but not equivalent built-in functions. For example, suppose you want to use a 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 of the Array class is created and reference to it is stored in the variable MyArray. A period and the class name identify the class, .Array. The period tells the interpreter to look in the Rexx environment for a class named "ARRAY". The Myarray array object has five elements, but the array itself is currently empty. Items can be added using methods defined by the array class.

```
myarray[1] = "Rick"
myarray[2] = "David"
myarray[3] = "Mark"

say myarray[1] myarray[2] myarray[3]
```

The Array class implements many more methods. See the ooRexx reference for details on what additional methods are provided.

By adding object technology to its repertoire of traditional programming techniques, Rexx has evolved into an object-oriented language, like Smalltalk. Rexx accommodates the programming techniques of traditional Rexx while adding new ones. With ooRexx, 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.

1.4. 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 good 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 ooRexx, you get the user-friendliness of Rexx in an object-oriented environment.

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

Object Rexx also provides access to FTP commands by means of its RxFtp package, and the use of mathematical functions by means of its RxMath utility package. The Sockets, FTP, and mathematical functions packages are each supplied with separate, full documentation.

1.5. 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*.

If you are unfamiliar with traditional Rexx, continue to read *Chapter 2, A Quick Tour of Traditional Rexx*.

A Quick Tour of Traditional Rexx

Because this book is for Windows and Unix 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.

2.1. 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 Windows or Unix/Linux, the ooRexx 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 Windows, Unix/Linux commands and other applications from a Rexx program. This is similar to what you can do with a Windows Batch facility program or a Unix shell script. 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 a DIR command can be intercepted by a Rexx program and used in subsequent processing.

Rexx can also direct commands to environments other than Windows. 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 scripting language for the application.

2.2. Running a Rexx Program

Rexx programs should have a file extension of .rex (the default searched for by the ooRexx interpreter). Here is a typical Rexx program named greeting.rex. It prompts the user to type in a name and then displays a personalized greeting:

Example 2.1. GREETING.REX

```
/* greeting.rex - 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 */
```

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. In this case, the message is the literal string "Please enter

your name.". The data between the quotes is a constant and will appear exactly as typed. You can use either single (') or double quote (") delimiters for literal strings.

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 the operation system command prompt. EXIT can also return a value. In our example, 0 is returned. The EXIT instruction is optional. Running off the end of the program is equivalent to coding "EXIT 0".

You can terminate a running Rexx program by pressing the Ctrl+Break key combination. Rexx stops running the program and control returns to the command prompt.

Rexx programs are often run from the command line, although, on the Windows operating systems there are several other options. These options are discussed several paragraphs later. The ooRexx interpreter is invoked by the command, **rexx**. With no arguments, the command produces a simple syntax reminder:

```
C:\Rexx>rexx

Syntax: REXX [-v] ProgramName [parameter_1....parameter_n]
or    : REXX [-e] ProgramString [parameter_1....parameter_n]

C:\Rexx>
```

To run the program **greeting.rex**, for example, use the command

```
rexx greeting.rex
```

or

```
rexx greeting
```

If not provided, an extension of ".rex" is assumed.

The **-v** option produces the version and copyright information. For example:

```
C:\Rexx>rexx -v
Open Object Rexx Version 4.0.0
Build date: Jul 16 2009
Addressing Mode: 64

Copyright (c) IBM Corporation 1995, 2004.
```

```
Copyright (c) RexxLA 2005-2009.
All Rights Reserved.
This program and the accompanying materials
are made available under the terms of the Common Public License v1.0
which accompanies this distribution.
http://www.oorexx.org/license.html
C:\Rexx>
```

The **-e** accepts a complete Rexx program in the form of a single string and executes it immediately. Enclose the string in double quotes and separate lines of the program with semi-colons. Arguments can follow the string:

```
C:\Rexx>rexx -e "use arg name; say 'Hello to you' name" Mark
Hello to you Mark

C:\Rexx>

C:\Rexx>rexx -e "parse arg a b; say a '*' b 'is' a*b" 12 3
12 * 3 is 36

C:\Rexx>rexx -e "parse arg a b; say a '*' b 'is' a*b" 22 -1
22 * -1 is -22

C:\Rexx>rexx -e "parse arg a b; say a '*' b 'is' a*b" 126 456
126 * 456 is 57456

C:\Rexx>
```

On *Windows only* there are these additional ways to run your Rexx programs:

- The installation program on Windows sets up a file association for the **.rex** file extension. This association allows the ooRexx programs to be run from Windows Explorer by double-clicking on the icon of the program file. In addition, the program can be run from a command prompt in a console window by simply typing the file name. The **.rex** extension is not needed. For example, simply type **greeting** to execute the *greeting.rex* program:

```
C:\>greeting
Please enter your name.
Mark
Hello MARK

C:\>
```

- A Rexx program can be run in *silent mode* by using **rexxhide**. This executes the program without creating a console window. This is most useful when creating a program shortcut. For the shortcut target, enter *rexxhide.exe* followed by the program name and the program arguments. Double-clicking on the shortcut then runs the program without creating a console window. **Note** that *silent* means there is no output from the Rexx program. When your program is run by **rexxhide**, either by double clicking on its icon, or from within a console window, there is no output displayed. Therefore **rexxhide** would not normally be used for programs like *greeting.rex*. This is what the *greeting.rex* program would look like when executed through **rexxhide**:

```
C:\>rexxhide greeting.rex

C:\>
```

- As a compliment to **rexxhide** is the **rexxpaws** program. When a Rexx program is executed through **rexxpaws**, at completion of the Rexx program, there will be a pause waiting for the user to hit the enter key. For example, using the *greeting.rex* program, **rexxpaws** would produce the following:

```
C:\>rexxpaws greeting.rex
Please enter your name.
Mark
Hello MARK

Press ENTER key to exit...

C:\>
```

rexxpaws is useful for running a Rexx program from a shortcut, where the program *does* produce output. On Windows, when double-clicking on the program file icon, a console window opens, the program is run, and then the console window immediately closes. **rexxpaws** prevents the console window from closing until the user hits the enter key. This allows the user to see what output the program produced.

2.3. 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 *Open Object Rexx: Reference* 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 Windows itself. These functions, known as the Rexx Utilities, let you work with resources managed by Windows or Linux, such as the display, the desktop, 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.

2.4. Writing Your Program

You can create Rexx programs using any editor that can create simple ASCII files without hidden format controls. Windows Notepad or Linux gedit are a couple widely available editors.

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 (,) or hypen (-) at the end of the line:

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

or

```
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!"
```

2.5. Testing Your Program

When writing your program, you can test statements as you go along using the REXXTRY command from the Windows 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 command prompt type:

```
rexx rexxtry  /* on windows the case of the REXX is insignificant */
```

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, just type the next line in your program and REXXTRY will check it for you.

Enter an equal sign (=) to repeat your previous statement.

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.

2.6. Variables, Constants, and Literal Strings

Comprehensive rules for variables, constants, and literal strings are contained in the *Open Object Rexx: Reference*.

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 _ .

- 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. Rexx translates lowercase letters in variables to uppercase before using them. Thus the variables names "abc", "Abc", and "ABC" all refer to the single variable "ABC". If you reference a variable name that has not yet been set, the name, in uppercase, is returned.

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 of the same type that starts the string is interpreted as a single character of that type. For example:

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

2.7. 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:

Example 2.2. Arithmetic

```
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:

Example 2.3. PARSE PULL

```
/* 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 break up data using a template. 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.)

Example 2.4. PARSE PULL

```
/* 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: Reference* for more information on Rexx parsing.

2.8. 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 2.5. Rexx function use

```
/* 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 2.6. Rexx function use

```
/* 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 2.7. Rexx expressions

```
/* 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 2.8. Rexx function use

```
/* 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)
```

2.9. Program Control

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

Example 2.9. 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 *Open Object Rexx: 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:

Example 2.10. IF and ELSE instructions

```
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:

Example 2.11. IF and ELSE instructions

```
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:

Example 2.12. ELSE IF instruction

```
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:

Example 2.13. SELECT instruction

```
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:

Example 2.14. SELECT and OTHERWISE instructions

```
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 has knows the DO/END and LOOP/END pair. All of the traditional looping variations are incorporated into the DO and LOOP instructions, which can be used interchangeably for looping:

Example 2.15. DO and LOOP instructions

```
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
```

or, using LOOP

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

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

b=3; a=0            /* LOOP WHILE - the conditional expression*/
loop 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                    /* LOOP UNTIL - like many other languages,*/
b=4                    /* a Rexx LOOP 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:

Example 2.16. DO FOREVER instruction

```
                  /* 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:

Example 2.17. ITERATE instruction

```
loop 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:

Example 2.18. ITERATE instruction

```
/* Say hello ten times if I is equal to 1 */
if i=1 then
   loop 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:

Example 2.19. SIGNAL instruction

```
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 use SIGNAL to jump to the middle of a DO/END block or into a SELECT structure.

2.10. 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:

Example 2.20. ROUTINE instruction

```
/* 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:

Example 2.21. RETURN instruction

```
/* 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:

Example 2.22. RESULT special variable

```
/* 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. This eliminates 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:

Example 2.23. PROCEDURE instruction

```
/* 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 the Ctrl+C key combination or close the command window 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:

Example 2.24. 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:

Example 2.25. Passing arguments

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

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

Into the Object World

Open Object Rexx includes features typical of an object-oriented language—features like subclassing, polymorphism, and data encapsulation. Object Rexx is new version 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.

3.1. 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.

3.2. 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.

```
PROGRAM ...
-----------------------------
-----------------------------
  PRINT ----------------
    ------------------------
    ------------------------
    ------------------------
  -----------------------------
  -----------------------------
  -----------------------------
  SEND ---------------
    -----------------------
    -----------------------
    -----------------------
  -----------------------------
  -----------------------------
  -----------------------------
  ERASE -------------
    -----------------------
    -----------------------
    -----------------------
```

```
            data
        data     data
          data
   data data     data
      data data
             data
      data     data
          data
```
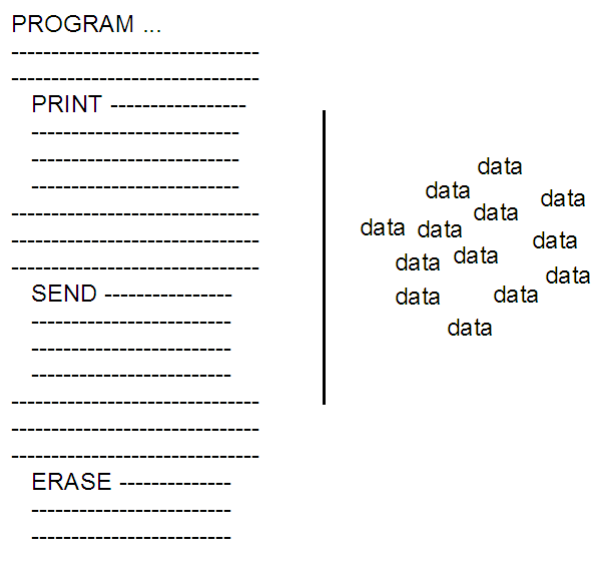
But with object-oriented programming, it is the data that is modularized. And each data module includes its own operations for performing actions directly related to its data.
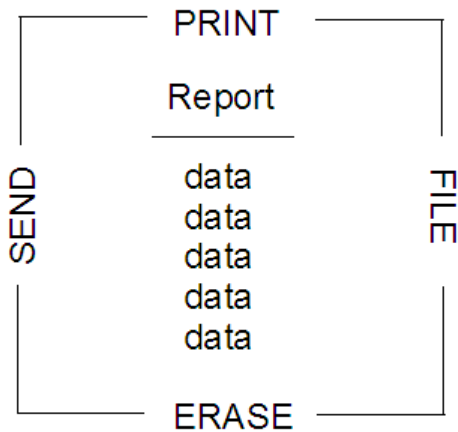
Figure 3.1. Modular Data—a Report Object

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

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

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

3.3. 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:
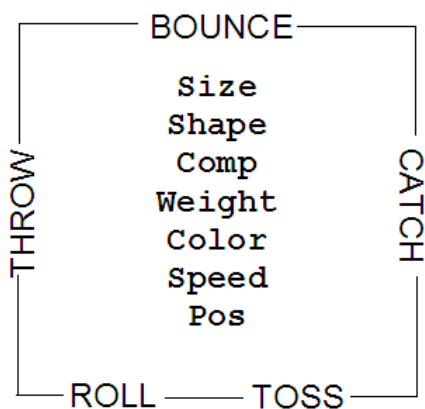
Figure 3.2. A Ball Object

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

There are many kinds of objects; for example, character strings, collections, and input and output streams. An object—such as a character string—always consists of two parts: the possible actions

or operations related to it, and its characteristics or variables. A variable has a variable name, and an associated data value that can change over time. These actions and characteristics are so closely associated that they cannot be separated:
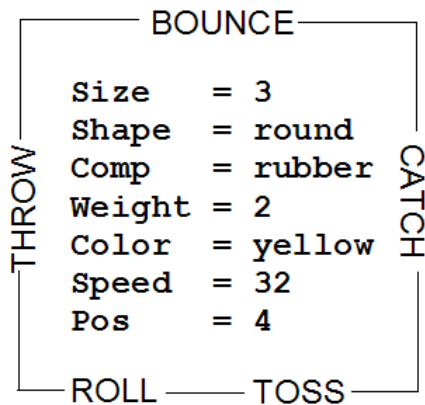
Figure 3.3. Ball Object with Variable Names and Values

To access an object's data, you must always specify an action. For example, suppose the object is the number **5**. Its actions might include addition, subtraction, multiplication, and division. Each of these actions is an interface to the object's data. The data is said to be encapsulated because the only way to access it is through one of these surrounding actions. The encapsulated internal characteristics of an object are its variables. Variables are associated with an object and exist for the lifetime of that object:
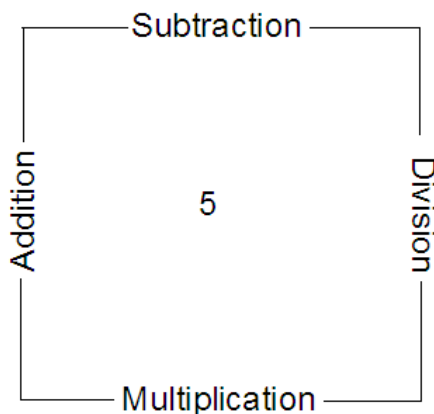
Figure 3.4. Encapsulated 5 Object

3.3.1. How Objects Interact

The actions defined by 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. Actions are invoked by sending messages. messages. Objects respond to these messages by invoking methods (*Section 3.3.2, "Methods"*) that perform an action, return data, or both. A message to an object must specify:

• A receiving object

• The "message send" symbol ~, which is called the twiddle

- The name of 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!.**

3.3.2. 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.

3.3.3. 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.

Polymorpism involves a form of contract between two objects. One object will send a message to another object expecting a particular result. Different types of objects can implement different versions of this message as long as it fulfills the expectations of the the invoking object. For example, the LOOP instruction has a form called OVER. Loop OVER will iterate over a number of elements provided by an object. For example,

Example 3.1. LOOP OVER

```
myarray = .array~of("Rick", "David", "Mark")
loop name over myarray
    say name
end
```

Will display the strings "Rick", "David", and "Mark", in that sequence. The LOOP OVER instruction will work with Arrays, Lists, stem variables, streams, etc. The LOOP instruction itself does not know anything about Arrays or Lists or stems or streams. The LOOP instruction specifies a contract. LOOP will send the target object the message MAKEARRAY and expects the target object to return an Array object that is used for the LOOP iteration. Any object can participate in LOOP iteration by implementing this contract. Objects that do implement the MAKEARRAY contract are polymorphic with the LOOP instruction.

3.3.4. 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. All Icon objects will share the actions and characteristics defined by the class:
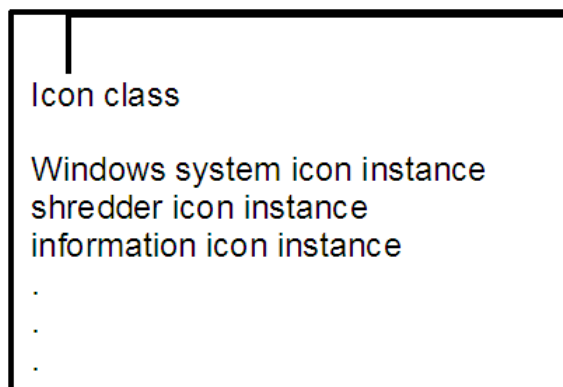
Figure 3.5. A Simple Class

All the icon objects will use common methods like DRAW or ERASE. They might will common variables like position, color, or size. What makes each icon object different from one another is the data assigned to its variables. For the Windows System icon, it might be **position="20,20"** while for the Shredder it is **"20,30"** and for Information it is **"20,40"**

Icon class

Windows system icon instance
(position='20,20')

shredder icon instance
(position='20,30')

information icon instance
(position='20,40')

Figure 3.6. Icon Class

Objects that belong to a class are called instances of that class. As instances of the Icon class, the Windows System icon, Shredder icon, and Information icon acquire the methods and variables of the 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.

Icon class
(position=)

Windows system icon instance
('20,20')

shredder icon instance
('20,30')

information icon instance
('20,40')
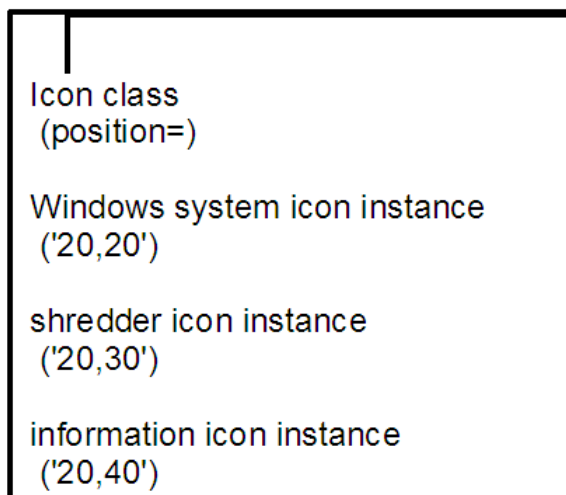
Figure 3.7. Instances of the Icon Class

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

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

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

3.3.5. 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.

3.3.6. 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 of your own, according to your needs.

Rexx classes are hierarchical. The original class is called a base class or a superclass. The derived class is called a subclass. Subclasses inherit methods and data from one or more superclasses. A subclass can introduce new methods and data, and can override methods from the superclass.
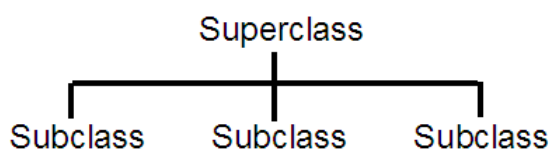
Figure 3.8. Superclass and Subclasses

You can add a class to an existing superclass. For example, you might add the Icon class to the Screen-Object superclass:
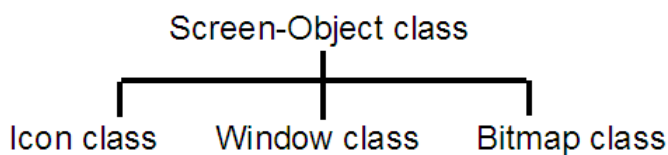
Figure 3.9. The Screen-Object Superclass

In this way, the subclass inherits additional methods from the superclass. A class can have more than one superclass, for example, subclass Bitmap might have the superclasses Screen-Object and Art-Object. Acquiring methods and variables from more than one superclass is known as multiple inheritance:
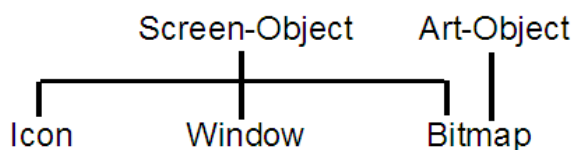
Figure 3.10. Multiple Inheritance

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 *Open Object Rexx: Reference*. The following list shows the classes Rexx provides (there may be others in the system). The classes indented are subclasses:

```
Object
    Alarm
    Buffer
    Class
    Collection classes
        MapCollection
            Directory
                Properties
            Relation
            Stem
            Table
            IdentityTable
        OrderedCollection
            Array
            List
            Queue
            CircularQueue
        SetCollection
            Bag
            Set
    Comparable
    Comparator
        CaselessColumnComparator
        CaselessComparator
        CaselessDescendingComparator
        ColumnComparator
        DescendingComparator
        InvertingComparator
        NumericComparator
    DateTime
    File
    InputOutputStream
        Stream
    InputStream
    Message
    Method
    Monitor
    MutableBuffer
    OutputStream
    Orderable
    Package
    Pointer
    RegularExpression
    RexxContext
    RexxQueue
    Routine
    Stream
    String
    Supplier
        StreamSupplier
    TimeSpan
    WeakReference
```

4.1. 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.

4.1.1. 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.

4.1.2. The Buffer Class

A Buffer instance is a Rexx interpreter managed block of storage. This class is designed primarily for writing methods and functions in native code and can only be created using the native code application programming interfaces.

4.1.3. The CaselessColumnComparator Class

The CaselessColumnComparator class performs caseless orderings of specific substrings of String objects.

4.1.4. The CaselessComparator Class

The CaselessComparator class performs caseless orderings of String objects.

4.1.5. The CaselessDescendingComparator Class

The CaselessDescendingComparator class performs caseless string sort orderings in descending order. This is the inverse of a CaselessComparator sort order.

4.1.6. 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.

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.

CircularQueue

The CircularQueue class allows for storing objects in a circular queue of a predefined size. Once the end of the queue has been reached, new item objects are inserted from the beginning, replacing earlier entries. Any object can be placed in the queue and the same object can occupy more than one position in the queue.

Directory

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

IdentityTable

An identity table is a collection with indexes that can be any object. In an identity table, each item is associated with a single index, and there can be only one item for each index. Index and item matches in an identity table are made using an object identity comparison. That is, an index will only match if the same instance is used in the collection.

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.

Properties

A properties object is a collection with unique indexes that are character strings representing names and items that are also restricted to character string values. Properties objects are useful for processing bundles of application option values.

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.

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.

Stem

A stem object is a collection with unique indexes that are character strings.

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.

4.1.7. The ColumnComparator Class

The ColumnComparator class performs orderings based on specific substrings of String objects.

4.1.8. The Comparable Class

Any object that inherits the Comparable mixin class and implements a compareTo() method can be sorted. The DateTime Class and TimeSpan Class are examples of built-in Rexx classes that can be sorted. Any user created class may also implement a compareTo() method to enable sorting.

4.1.9. The Comparator Class

The Comparator class is the base class for implementing Comparator objects that can be used with the Array sortWith() or stableSortWith() method. The compare() method implements some form of comparison that determines the relative ordering of two objects. Many Comparator implementations are specific to particular object types.

4.1.10. The DateTime Class

A DateTime object represents a point in between 1 January 0001 at 00:00.000000 and 31 December 9999 at 23:59:59.999999. A DateTime object has methods to allow formatting a date or time in various formats, as well as allowing arithmetic operations between dates.

4.1.11. The DescendingComparator Class

The DescendingComparator class performs sort orderings in descending order. This is the inverse of a Comparator sort order.

4.1.12. The File Class

The File class provides services which are common to all the filesystems supported by ooRexx. A File object represents a path to a file or directory. The path can be relative or absolute.

4.1.13. The InputOutputStream Class

This class is defined as an abstract mixin class. It must be implemented by subclassing it or inheriting from it as a mixin. Many of the methods in this class are abstract and must be overridden or they will throw a syntax error when invoked.

4.1.14. The InputStream Class

This class is defined as an abstract mixin class. It must be implemented by subclassing it or inheriting from it as a mixin. Many of the methods in this class are abstract and must be overridden or they will throw a syntax error when invoked.

4.1.15. The InvertingComparator Class

The InvertingComparator class inverts the comparison results of another Comparator object to reverse the resulting sort order.

4.1.16. The Message Class

Message objects allow you to run concurrently methods on other threads or to invoke dynamically calculated messages. Methods of for this class are used, for example, to start a message on another thread, to notify the sender object when an error occurs or when message processing is complete, or to return the results of that processing to the sender or to some other object.

4.1.17. The Method Class

The Method class creates method objects from Rexx source code.

4.1.18. The Monitor Class

The Monitor class provides a way to forward messages to a specified destination. The Monitor creates a proxy that can route dynamically route messages to different destinations. Monitor methods change or restore a destination object.

4.1.19. The MutableBuffer Class

The MutableBuffer class is a buffer on which certain string operations such as concatenation can be performed very efficiently. Unlike String objects, MutableBuffers can be altered without requiring a new object allocation. A MutableBuffer object can provide better performance for algorithms that involve frequent concatenations to build up longer string objects because it creates fewer intermediate objects.

4.1.20. The OutputStream Class

This class is defined as an abstract mixin class. It must be implemented by subclassing it or inheriting from it as a mixin. Many of the methods in this class are abstract and must be overridden or they will throw a syntax error when invoked.

4.1.21. The NumericComparator Class

The NumericComparator class compares string using numeric comparison rules to determine sort order.

4.1.22. The Orderable Class

The Orderable class can be inherited by classes which wish to provide each of the comparison operator methods without needing to implement each of the individual methods. The inheriting class need only implement the Comparable compareTo() method. This class is defined as a mixin class.

4.1.23. The Package Class

The Package class contains the source code for a package of Rexx code. A package instance holds all of the routines, classes, and methods created from a source code unit and also manages external dependencies referenced by ::REQUIRES directives. The files loaded by ::REQUIRES are also contained in Package class instances.

4.1.24. The Pointer Class

A Pointer instance is a wrapper around a native pointer value. This class is designed primarily for writing methods and functions in native code and can only be created using the native code application programming interfaces. The Pointer class new method will raise an error if invoked.

4.1.25. The RegularExpression Class

This class provides support for regular expressions. A regular expression is a pattern you can use to match strings.

4.1.26. The RexxContext Class

The RexxContext class gives access to context information about the currently executing Rexx code. Instances of the RexxContext class can only be obtained via the .CONTEXT environment symbol. They cannot be directly created by the user.

4.1.27. The RexxQueue Class

The RexxQueue class provides object-style access to Rexx external data queues.

4.1.28. The Routine Class

The Routine class creates routine objects from Rexx source code.

4.1.29. The StackFrame Class

The StackFrame class provides debugging information about a Rexx activity's execution chain.

4.1.30. The Stem Class

A stem variable is a symbol that must start with a letter and end with a period, like "FRED." or "A.". The value of a stem variable is a Stem *object*. A stem object is a collection of unique character string indexes. 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.

4.1.31. 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* for examples).

4.1.32. The StreamSupplier Class

A subclass of the Supplier class that will provided stream lines using supplier semantics. This allows the programmer to iterate over the remaining lines in a stream. A StreamSupplier object provides a snapshot of the stream at the point in time it is created, including the current line read position. In general, the iteration is not effected by later changes to the read and write positioning of the stream. However, forces external to the iteration may change the content of the remaining lines as the itertion progresses.

4.1.33. 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.

4.1.34. The Supplier Class

All collections have suppliers The Supplier class is used to enumerate items that a collection contained when the supplier was created. The supplier gives access to each index/value pair stored in the collection as a sequence.

4.1.35. The TimeSpan Class

A TimeSpan object represents a point in between 1 January 0001 at 00:00.000000 and 31 December 9999 at 23:59:59.999999. A TimeSpan object has methods to allow formatting a date or time in various formats, as well as allowing arithmetic operations between dates.

4.1.36. The WeakReference Class

A WeakReference instance maintains a non-pinning reference to another object. A non-pinning reference does not prevent an object from getting garbage collected or having its uninit method run when there are no longer normal references maintained to the object. Once the referenced object is eligible for garbage collection, the reference inside the WeakReference instance will be cleared and the VALUE method will return .nil on all subsequent calls. WeakReferences are useful for maintaining caches of objects without preventing the objects from being reclaimed by the garbage collector when needed.

4.2. Rexx Classes for Organizing Objects

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

4.2.1. The Object Class

Because the root class in the hierarchy, is the Object class, everything below it is an object. To interact with each other, objects require their own actions, called methods. These methods, which encode actions that are 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 own classes. Any method created for the Object class is automatically made available to every other class in the hierarchy.

4.2.2. 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 (*Section 4.6.4, "Metaclasses"*). The Class class is unique among Rexx classes in that it is the only metaclass that Rexx provides. 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:
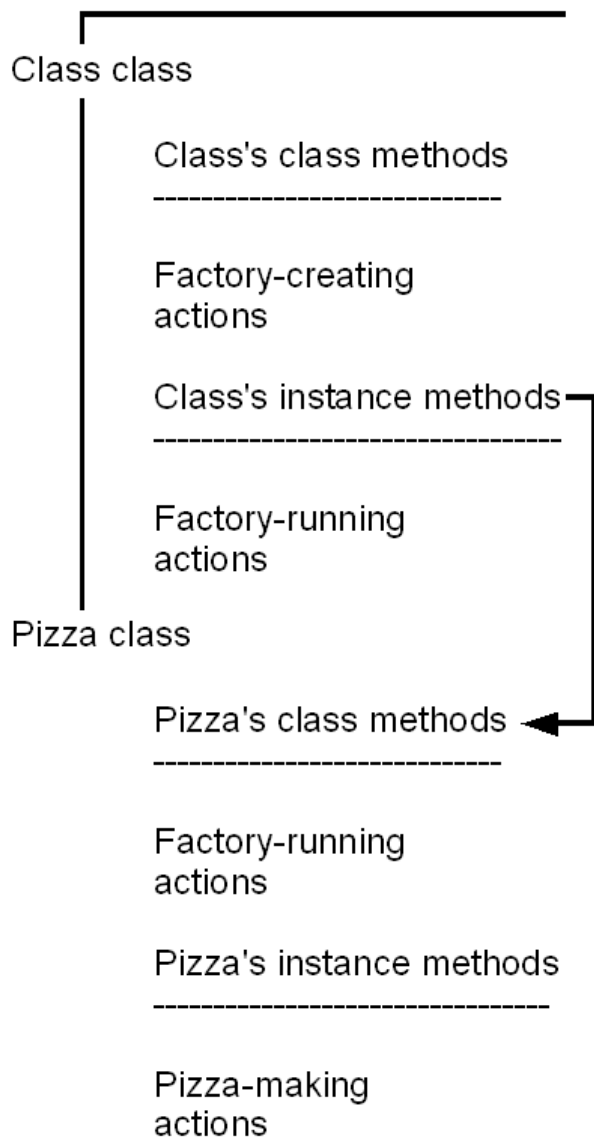
Figure 4.1. How Subclasses Inherit Instance Methods from the Class Class

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 declares class definitions in a simple, static form in your programs.

4.3. Rexx Classes: The Big Picture

The following are the supplied Rexx classes.

4.4. 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 your class definitions with others using the PUBLIC option.

4.4.1. 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).

4.4.2. The Directives Rexx Provides

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

4.4.2.1. 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 *Section 4.4.4, "A Sample Program Using Directives"*, 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 *Section 4.4.4, "A Sample Program Using Directives"*, 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 *Section 4.6.4, "Metaclasses"*.

- 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.

4.4.2.2. 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.

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 objects of the same type 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 *Section 5.7.2.1, "Activating Methods"* (the UNGUARDED option).

4.4.2.3. The ::ATTRIBUTE Directive

A ::CLASS directive can also be followed by ::ATTRIBUTE directives, which are used to create methods that directly access internal attributes of an object. For example, the Account class could define

```
::attribute balance
```

Which would allow the account balance to be set or retrieved.

```
anAccount~balance = 10000    -- set a new account balance
say anAccount~balance        -- display the current account balance
```

The access methods can created as read-only, or given private scope.

4.4.2.4. 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 organizing functions that are not specific to a particular class type.

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.

4.4.2.5. 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
```

::REQUIRES directives are processed before 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 imported through ::REQUIRES directives.

4.4.3. 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.

4.4.4. A Sample Program Using Directives

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

Example 4.1. Using directives

```
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"

   ::attribute 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 a TYPE method and ::ATTRIBUTE to create NAME and NAME= methods for Account. In the **::method type** instruction, 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 **::attribute name** directive, we're creating a pair of methods. The NAME method returns the current value of the NAME object variable. The NAME= method can assign a new value to the NAME 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 recovers 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.

4.4.5. 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:

Example 4.2. Defining methods

```
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
```

4.5. 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 newaccount~type
newaccount~name = "John Smith"
```

4.6. Types of Classes

There are four kinds of classes:

- Object classes

- Mixin classes

- Abstract classes

- Metaclasses

The following sections explain these.

4.6.1. Object Classes

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

The instance variables within an object are created on demand whenever a method EXPOSEs an object variable. The class creates the object instance, defines the methods the object has, and the object instance completes the job of constructing the object.

The String class and the Array Class are examples of object classes.

4.6.2. Mixin Classes

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

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

Mixins are always associated with a *base class*, which is the mixin's first non-mixin superclass. Any subclass of the mixin's base class can (directly or indirectly) inherit a mixin; other classes cannot. For example, a mixin class created as a subclass of the Array class can only be inherited by other Array subclasses. Mixins that use the Object class as a base class can be inherited by any class.

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

4.6.3. Abstract Classes

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

You create an abstract class like object or mixin classes. No extra messages or keywords on the ::CLASS directive are necessary. Rexx does not prevent users from creating instances of abstract classes. It is possible to create abstract methods on a class. An abstract method is a placeholder that subclasses are expected to override. Failing to provide a real method implementation will result in an error when the abstract version is called.

4.6.4. Metaclasses

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

To change the behavior of an object that is an instance, you generally use subclassing. For example, you can create Statarray, a subclass of the Array class. The statArray class can include a method for computing a total of all the numeric elements of an array.

Example 4.3. Creating an Array subclass

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

::class statArray subclass array public

::method init    /*  Initialize running total and forward to superclass */
  expose total
  total = 0
  forward class (super)

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

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

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

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

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

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

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

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

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

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

Version 1

Example 4.4. Adding a class method

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

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



::class point public             /* create Point class           */

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

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

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


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

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

Version 2

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

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

::class InstanceCounter subclass class /* Create a new metaclass that */
                                     /* will count its instances     */
::method init
  expose instanceCount
  instanceCount = 0                  /* Initialize instanceCount     */
  forward class (super)              /* Forward INIT to superclass   */
```

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

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


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

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

A Closer Look at Objects

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

A Rexx object consists of:

• Actions coded as methods

• Attributes, 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 with the matching name 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 are "of" the class, but are not classes themselves.

Both the classes and their instances contain variables and methods. The methods a class provides for use by its instances are called instance methods. The instance methods define which messages an object can respond to.

The methods available to the class itself are called class methods. Many of the methods are actually the instance methods of the Class class, but a class many have its own unique class methods. They define messages that only the class—and not its instances—can respond to.
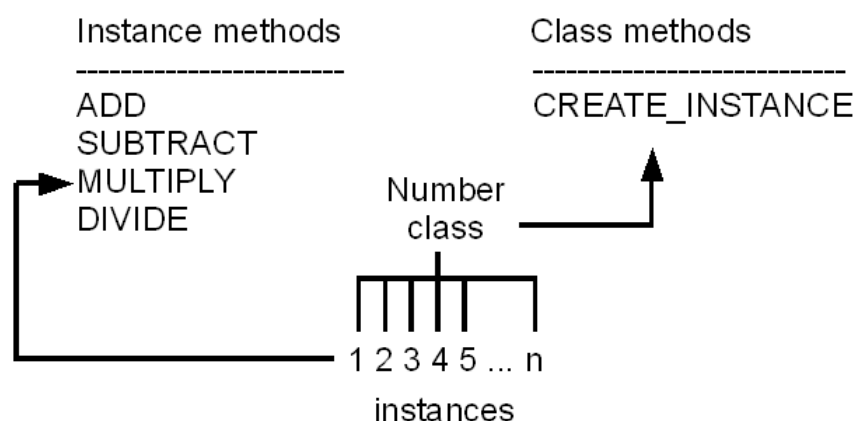
Figure 5.1. Instance Methods and Class Methods

5.1. Using Objects in Rexx

The following examples with Myarray illustrate how to use new objects in Rexx programs.

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

creates a new instance of the Array class, and assigns to the variable MYARRAY. The period precedes a class name in an expression, to distinguish the class environment symbol from other variables. The MYARRAY array object has five elements.

After the array 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 value added to the array, the second is the number of the element in which to place the value. Here, the string object **Hello** is associated with the third element of Myarray:

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

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 a string object as input, which is what AT returns. If you try to display a non-string object in the SAY instruction, SAY sends a STRING message to the object. The STRING method returns a human-readable string representation for the object. In this example, the STRING method for an Array object 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 produces 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 Array object referenced by MYARRAY. 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, "+" must be a literal string because the message name contains characters not allowed in a Rexx symbol.

5.2. Common Methods

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

5.2.1. 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 instance variables are initialized 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 message are passed to the INIT method, which can use them to set the initial state of object variables.

If a class overrides the INIT method it inherits from a superclass, the new INIT method must forward the INIT message up the hierarchy, to properly initialize the instance. An example in the next section demonstrates the use of INIT.

5.2.2. 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:

Example 5.1. STRING and INIT methods

```
/* 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 the new instance.. 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 single 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=12345)
```

Figure 5.2. Instances in the Part Class

The INIT method 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 identify which object variables it requires. The RETURN instruction returns the result string.

The following example shows how to use the Part class:

Example 5.2. 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.

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.

5.2.3. Uninitializing and Deleting Instances Using UNINIT

Normally, object classes can create instances but have no direct control over their deletion. Once an object is no longer referenced by any variables, Rexx automatically reclaims the storage for the old value in a process called garbage collection.

If the instance has allocated other system resources, Rexx cannot automatically release these resources because it is unaware that the instance has allocated them. An UNINIT method give an object the opportunity to perform resource cleanup before the object is reclaimed by the garbage collector.

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:

Example 5.3. UNINIT method

```
/* 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, If the object only contains references to normal Rexx objects, an UNINIT method is generally not needed. If the object contains references to external system resources such as open network connections or database connections, an UNINIT method might be required to release those resources. If an object requires uninitialization, define an UNINIT method to perform the cleanup processing you require.

If an object has UNINIT an method, Rexx runs it before reclaiming the object's storage. If an instance overrides an UNINIT method of a superclass, each UNINIT method is responsible for sending the

UNINIT message up the hierarchy, using the SUPERCLASS overrides, so that each inherited UNINIT method has the opportunity to run.

5.3. Special Method Variables

When writing methods, there are several special variables that are set automatically when a method runs. Rexx supports the following variables:

SELF

    is set when a method is activated. Its value is the object that forms the execution context for the method (that is, the object that received the activating message).

    You can use SELF to:

- Send messages to the currently active object. 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.

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.

> Example 5.4. SELF variable
>
> ```
> ::class Account
>
>   ::method INIT
>     expose balance
>     use arg balance
>     self~init:super    /* Forwards to the Object INIT method */
>
>   ::method TYPE
>     return "an account"
> ```

```
    ::method name attribute

::class Savings subclass Account

  ::method INIT
    expose interest_rate
    use arg balance, interest_rate
    self~init:super(balance)  /* Forwards to the Account INIT method */

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

When the INIT method of the Savings class is called, the variable SUPER is set to the Account class object. 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. So in the Account class INIT:

```
self~init:super
```

calls the INIT method of the Object class.

5.4. 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:

5.4.1. The Public Environment Object (.environment)

The Environment object is a directory of public objects that are always accessible throughout the whole process. The Rexx built-in classes are stored in the Environment directory. 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, and the symbol is not a valid numeric value. 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.

5.4.1.1. 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.

5.4.2. 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.

5.4.3. 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 (.).

.line

> The .line environment symbol returns the line number of the current instruction being executed. If the current instruction is defined within an INTERPRET instruction, the value returned is the line number of INTERPRET instruction.

.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.

5.4.4. 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.

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, but .local should be preferred over .environment to avoid accidentally overwriting system-defined values. 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 5.5. .Local object

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

5.5. 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.

5.5.1. 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:
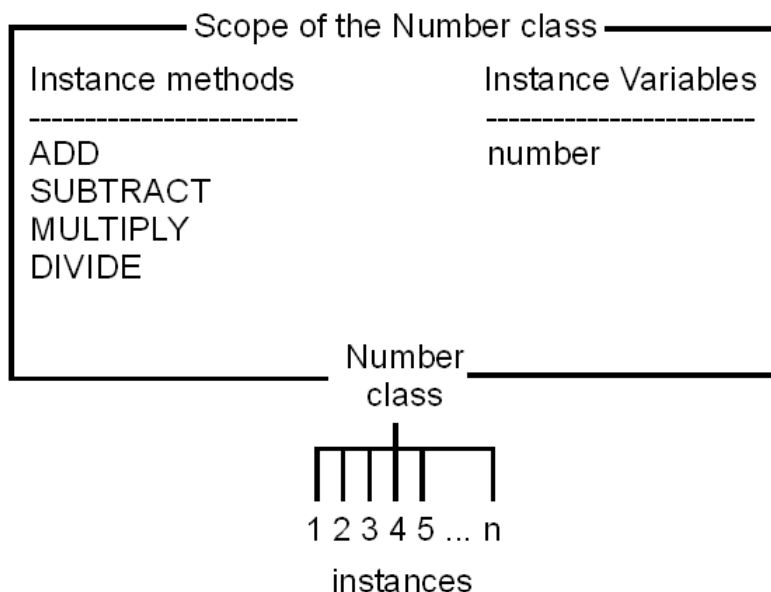
Figure 5.3. Scope of the Number Class

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 for completely unrelated purposes.

5.5.2. 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.

5.6. 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
```

5.6.1. 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 *Section 5.6.4, "Defining an UNKNOWN Method"*. 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.
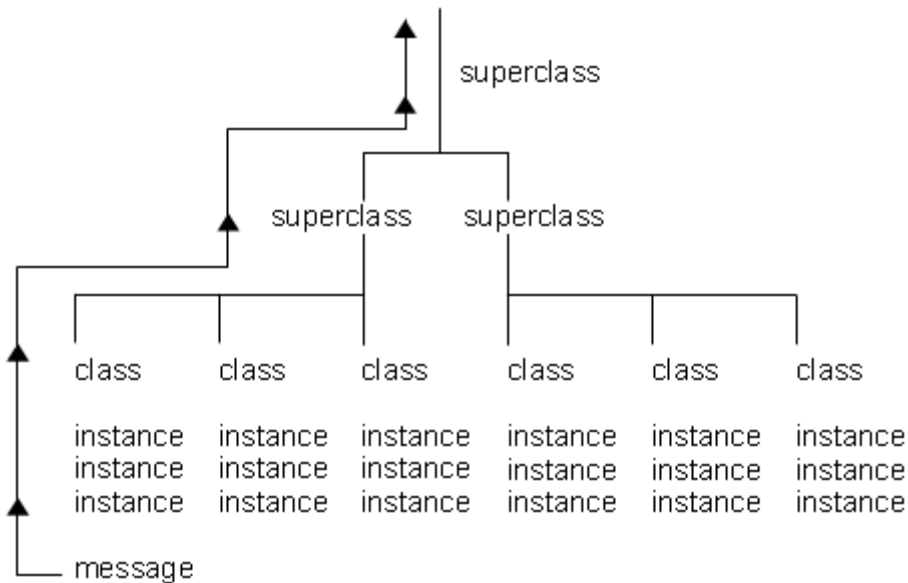
Figure 5.4. Searching the Hierarchy for a Method

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.

5.6.2. 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.

2. Specifying a colon and a starting scope after the message name. The starting scope is a variable or environment symbol that identifies the scope object to use as the method search starting point. This scope object can be:
    *   A direct superclass of the class that defines the active method

    *   The object itself (for example, the value of the variable SELF), if you used SETMETHOD to add methods to the object.

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

In *Section 4.4.4, "A Sample Program Using Directives"*, 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:

Example 5.7. SUBCLASS option

```
::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:

Example 5.8. SUBCLASS method search order

```
  ::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.

5.6.3. Public versus Private Methods

A method can be public or private. Any object can send a message that runs a public method. A private method can only be invoked from specific calling contexts. These contexts are:

1. From within a method owned by the same class as the target. This is frequently the same object, accessed via the special variable SELF. Private methods of an object can also be accessed from other instances of the same class (or subclass instances).

2. From within a method defined at the same class scope as the method. For example:

Example 5.9. PUBLIC and PRIVATE options

```
::class Savings
::method newCheckingAccount CLASS
  instance = self~new
  instance~makeChecking
  return instance

::method makeChecking private
  expose checking
  checking = .true
```

The newCheckingAccount CLASS method is able to invoke the makeChecking method because the scope of the makeChecking method is .Savings.

3. From within an instance (or subclass instance) of a class to a private class method of its class. For example:

Example 5.10. PUBLIC and PRIVATE options

```
::class Savings
::method init class
  expose counter
  counter = 0

::method allocateAccountNumber private class
  expose counter
  counter = counter + 1
  return counter

::method init
  expose accountNumber
  accountNumber = self~class~allocateAccountNumber
```

The instance INIT method of the Savings class is able to invoke the allocateAccountNumber private method of the .Savings class object because it is owned by an instance of the .Savings class.

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 to outsiders, but allowing objects to share information with each other and their defining classes.

5.6.4. 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.

5.7. 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.

5.7.1. 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, using 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.

5.7.1.1. Object Instance Variables

Every object has its own set of instance variables. 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 instance variables.

Therefore, an object's instance variables includes variables:

- Exposed by methods defined by the object's class. This set of variables is called a variable pool.

- Exposed by methods defined by other classes in the inheritance hierarchy. The methods of each class share variables in a pool scoped to just that class.

A class's variable pool, together with the methods that expose them, are called a class scope. Rexx exploits this class scope to achieve concurrency. To explain, the object's instance variables are contained in a collection of variable pools. Each pool is at a different scope in the object's inheritance chain. Methods defined at different class scopes do not directly share data and can run simultaneously.

Scopes, like objects, hide and protect data from outside manipulation. Methods of the same scope share the variable pool of that scope. The scope shields the variable pool 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 instance variables into pools that can operate independently of one another. Several methods can use the same object instance variables concurrently, as long as they confine themselves to variables in their own scope.

5.7.1.2. 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 scope, that method has exclusive use of the scope variable pool until processing is complete. Until then, Rexx delays the execution of any other method that needs the same scope variables.

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.

5.7.1.3. 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.

5.7.2. 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 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.

5.7.2.1. Activating Methods

By default, Rexx assumes that an active method requires exclusive use of its scope variable pool. If another method attempts access at that time, it is locked out until the first method finishes. 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, 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 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 no longer requires exclusive use of the variable

pool it can use the GUARD OFF instruction to allow methods running on different activities to become active on the same scope. 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.

Commands

From a Rexx program you can pass commands to Windows and Unix/Linux shells or to applications designed to work with Rexx. When used to run operating system commands, Rexx becomes a powerful substitute for the Windows Batch Facility or Unix shell scripts. You can use variables, control structures, mathematics, and parsing, and the full object oriented features of Rexx.

Applications that are designed to work with Rexx are often referred to as scriptable applications. To work with Rexx, a scriptable application registers a command environment with Rexx. An environment serves as a kind of workspace shared between Rexx and the application that accepts application commands issued from your Rexx programs.

For example, many editors provide a command prompt or dialog box from which you can issue commands to set margins or add lines. If the editor is scriptable from Rexx, you can issue the same editor commands 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. The application processes the command, and returns a status indicator as a return code.

The Rexx ADDRESS instruction allows you select which named command environment commands get directed to. There is always at least one active command environment, and all Rexx programs start with a default environment selected. For programs launched from a command shell, an operating-specific command handler is the normal default. Applications such as an editor can choose to make their own command environment the default.

6.1. How to Issue Commands

Rexx makes it easy to issue commands. The basic rule is that whatever Rexx cannot process directly gets passed to the current command 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 a command environment 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 the current command environment.

If the string is a valid valid command for that environment, the command handler will process it as if you had entered it at the command prompt.

The following example shows a Rexx clause that uses the Windows DIR command to display a list of files in the current directory.

```
/* display current directory */
say "DIR command using Rexx"
dir
```

The clause **dir** is not a Rexx instruction or a label, so Rexx evaluates it and passes the resulting string to Windows. Windows recognizes the string DIR as one of its commands and processes it.

Letting Rexx evaluate the command as an expression might cause problems, however. Try adding a path to the DIR command in the above program (such as, **dir c:\config.sys**). The Windows command in this case is an incorrect Rexx expression. The program ends with an error.

A safer way to issue commands is by enclosing the command in quotes, which makes the command a literal string. Rexx does not evaluate the contents of strings, so the string is passed to Windows as-is. Here is an example using the PATH command:

```
/* display current path      */
say "PATH command using Rexx"
"path"
```

The following example, DP.CMD, shows a program using the DIR and PATH commands. The PAUSE command is added to wait for the user to press a key before issuing the next instruction or command. Borders are added too.

Example 6.1. DIR and PATH commands

```
/* DP.CMD -- Issue DIR and PATH commands to Windows */

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

"dir"                 /* display listing of      */
                      /* the current directory   */

"pause"               /* pauses processing and   */
                      /* tells user to "Press    */
                      /* any key to continue."   */

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

When you specify the following:

```
[C:\]rexx dp
```

a possible output would be:

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

The volume label in drive C is WIN.
Directory of C:\EXAMPLES

.            <DIR>      10-16-94  12:43p
..           <DIR>      10-16-94  12:43p
EX4_1    CMD     nnnn  10-16-94   1:08p
DEMO     TXT      117  10-16-94   1:10p
4 File(s)   12163072 bytes free
Press any key when ready . . .

=======================================
PATH=C:\WINDOWS
[C:\]
```

> ## Note
>
> Usually, when executing a host command addressed to the Windows or Unix/Linux command shell, a new process is created in the system command handler to execute the command. Changes in a child process environment do not change the parent process environment. Therefore, any change in the environment, such as a directory change, made by a host command executed in a child process would not be reflected in the process running the Rexx program.
>
> The interpreter attempts to mitigate this to some extent by executing some host commands in the process running the Rexx program, rather than in a child process. This is done so that changes to the environment made by executing the host command are visible in the process running the Rexx program.
>
> This is only done when the host command line is simple. That is, the command line must contain a single command, without redirection and without pipe. On Windows this applies to the **CD** and **SET** commands. On Unix-like systems, including Linux, this applies to **cd**, **set**, **unset** and **export**. Rather than remembering the rules, it may be easier to avoid a potential problem by using the built in **directory()** or **value()** functions rather than issuing a host command for **cd**, **set**, etc.
>
> Example 6.2. Environment commands (Windows)
>
> ```
> 'cd c:\tmp'              /* executed in Rexx program process */
> 'cd "c:\R&D (secret)"'   /* executed in Rexx program process */
> 'cd c:\windows && dir c:' /* executed in child process (2 commands) */
> 'd:'                     /* executed in Rexx program process */
> 'set myvar=my value'     /* executed in Rexx program process */
> ```
>
> Example 6.3. Environment commands (Unix)
>
> ```
> 'cd'                     /* executed in Rexx program process: go to $HOME directory
>  */
> 'cd ~/"R&D (secret)"'    /* executed in Rexx program process: go to $HOME/R&D
>  (secret) */
> 'cd ~/"R&D \"secret\""'  /* executed in Rexx program process: go to $HOME/R&D
>  "secret" */
> 'cd ~john'               /* executed in Rexx program process: go to John's home
>  directory */
> 'cd /tmp && pwd'         /* executed in child process (2 commands) */
> 'set myvar=my value'     /* executed in Rexx program process */
> 'export myvar=my value'  /* executed in Rexx program process */
> 'unset myvar'            /* executed in Rexx program process */
> ```

6.2. Rexx and Batch Files

You can use a Rexx program whenever you now use Windows batch files or Unix/Linux shell scripts. The following example shows a Windows batch file that processes user input to display a help message:

Example 6.4. Windows batch file

```
@echo off
if %1.==. goto msg
if %1 == on goto yes
if %1 == off goto no
if %1 == ON goto yes
if %1 == OFF goto no
if %1 == On goto yes
if %1 == oN goto yes
if %1 == OFf goto no
if %1 == OfF goto no
if %1 == Off goto no
if %1 == oFF goto no
if %1 == oFf goto no
if %1 == ofF goto no
helpmsg %1
goto exit
:msg
helpmsg
goto exit
:yes
prompt $i[$p]
goto exit
:no
cls
prompt
:exit
```

Here is the equivalent program in Rexx:

Example 6.5. Rexx program

```
/* HELP.CMD -- Get help for a system message */
arg action .
select
  when action=""    then     "helpmsg"
  when action="ON"  then     "prompt $i[$p]"
  when action="OFF" then do
    "cls"
    "prompt"
  end
  otherwise "helpmsg" action
end
exit
```

6.3. 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 TYPE 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 Windows:

Example 6.6. SHOFIL.CMD (Windows)

```
/* SHOFIL.CMD - build command with variables  */

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

/* assign the response to variable FILENAME    */
pull filename

/* build a command string by concatenation     */
commandstr = "TYPE" filename

/* Assuming the user typed "demo.txt,"         */
/* the variable COMMANDSTR contains            */
/* the string "TYPE DEMO.TXT" and so...        */

commandstr              /* ...Rexx passes the    */
                        /* string on to Windows  */
```

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

```
[C:\]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.

[C:\]
```

6.4. 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 Windows programs. The DIRREX.CMD program below shows how Rexx evaluates a command when the command name and a variable name are the same:

Example 6.7. DIRREX.CMD (Windows)

```
/* DIRREX.CMD - assign a value to the symbol DIR  */
say "DIR command using Rexx"
dir = "echo This is not a directory."

/* pass the evaluated variable to Windows        */
dir
```

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

```
[C:\]rexx dirrex
DIR command using Rexx:
This is not a directory.
[C:\]
```

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:

Example 6.8. Passing values to command environments

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

/* pass the literal string "dir" to Windows */
"dir"
```

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"
"delete *."||extension

option = "/w"
"dir"||option
```

6.5. 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 Windows environment, use the symbol CMD. For *expression*, specify an expression that results in a string that Rexx passes to the environment. Here are some examples:

Example 6.9. ADDREXX instruction

```
address CMD "dir"     /* pass the literal string    */
                      /* "dir" to Windows           */

address "bash" "ls"   /* pass the literal string    */
                      /* "ls" to the Linux bash shell */

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

address CMD cmdstr    /* Rexx passes the string     */
                      /* "dir *.txt" to Windows     */
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.

6.6. Using Return Codes from Commands

With each command it processes, Windows and Unix/Linux command shells produce 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:

Example 6.10. GETRC.CMD (Windows)

```
/* GETRC.CMD report */
"TYPE 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 TYPE command returns a nonzero value in RC:

Example 6.11. RC special variable

```
/* Simple if/then error handler */
say "Type a file name:"
pull filename
"TYPE" 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 Ctrl+Break key combination 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

6.7. Subcommand Processing

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

To make your own applications scriptable from Rexx, see *Chapter 9, Classic Rexx Application Programming Interfaces*.

6.8. 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 *Open Object Rexx: Reference.*

6.8.1. 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 *Open Object Rexx: Reference.*

6.8.2. 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:

Example 6.12. SIGNAL

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

6.8.3. 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.          */
       .
       .
       .
"COPY"  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.   */
```
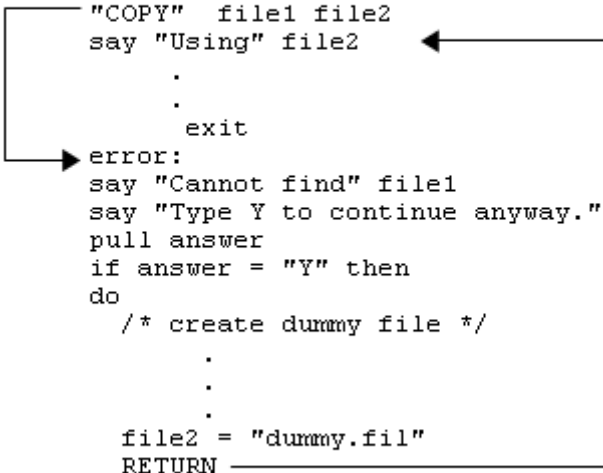
6.8.4. Using CALL ON ERROR

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 on error
       .
       .
       .
"COPY"  file1 file2
say "Using" file2        ◄
       .
       .
       exit
error:
say "Cannot find" file1
say "Type Y to continue anyway."
pull answer
if answer = "Y" then
do
   /* create dummy file */
       .
       .
       .
   file2 = "dummy.fil"
   RETURN
end
else exit
```

6.8.5. A Common Error-Handling Routine

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

Example 6.13. Common error handling routine

```
/* Here is a sample "main program" with an error          */
```

```
signal on error        /* enable error handling          */
"ersae myfiles.*"      /* mistyped "erase" 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
```

Input and Output

Object Rexx supports a stream I/O model. Using streams, 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. A Rexx stream object allows you read from a data stream using either the text-file line methods or using a continuous data stream method.

The Rexx Stream class is the mechanism for accessing I/O streams. 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 C:\out.dat:

```
/* Create a stream object for out.dat */
file=.stream~new("c:\out.dat")
```

Then you use the appropriate stream methods access the data. OUT.DAT is a text file, so you would normally use the LINES(), LINEIN, and LINEOUT() methods that that read or write data as delimited lines. If the stream represents a binary file (such as a WAV, GIF, TIF, AVI, or EXE file), you would use the CHAR, CHARIN, and CHAROUT methods that read and write data as characters.

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

7.1. More about Stream Objects

To use streams in 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.

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, read from a random position in the data stream—it can only be read as a sequential stream of characters. Once you write to the stream, 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 reread data that you have previously read. Rexx maintains separate read and write pointers to a stream that you can move the pointers independently using arguments on methods such as LINEIN, LINEOUT, CHARIN, and CHAROUT. The Stream class also provides SEEK and POSITION methods for setting the read and write positions.

7.2. 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 r:\rexx\articles\devcon7.scr
```

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

Example 7.1. COUNT.CMD

```
/* 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 characters.

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:

Example 7.2. COUNT.CMD

```
/* 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.

7.3. Reading a Text File into an Array

Rexx provides a Stream method, named ARRAYIN, that reads the contents of a stream into an array object. ARRAYIN 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 ARRAYIN method to read the entire CONFIG.SYS file into an array object. CVIEW displays selected lines from CONFIG.SYS. 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.

Example 7.3. CVIEW.CMD

```
/* CVIEW - display lines from CONFIG.SYS                         */
parse upper arg search_string     /* Get any command line argument  */
file=.stream~new("c:\config.sys")  /* Create stream object */
lines=file~arrayin                /* Read file into an array object */
                                  /* LINES points to the array obj. */
loop 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
   loop 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
   end
   search_string=""               /* Reset for next search          */
end
```

7.4. 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 CONFIG.SYS:

Example 7.4. LINEIN

```
/* Read and display line 3 of CONFIG.SYS */
infile=.stream~new("c:\config.sys")
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.

7.5. 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:

Example 7.5. TODO.CMD

```
/* 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.

Example 7.6. REPFILE.CMD

```
/* 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.

7.6. 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 CHORD.WAV file (supplied with Windows multimedia support in c:\winnt) into a variable:

Example 7.7. GETCHORD.CMD

```
/* GETCHORD - reads CHORD.WAV into a variable          */
chordf=.stream~new("c:\winnt\chord.wav")
say "Number of characters in the file=" chordf~chars

/* Read the whole WAV file into a single Rexx variable. */
/* Rexx variables are limited by available memory.      */
mychord=chordf~charin(1,chordf~chars)
say "Number of characters read into variable" mychord~length
```

The CHARIN method returns a string of characters from the stream, which in this case is CHORD.WAV. 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. CHORD.WAV has about 25000 characters.

7.7. 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 characters to your program. Line methods, on the contrary, do not return the line-end characters to your program.

The line-end characters on Windows consist of a carriage return (ASCII value of 13) and a line feed (ASCII value of 10). The line-end characters on Unix/Linux consist of a line feed (ASCII value of 10). Rexx adds these characters to the end of every line written using the LINEOUT method. Text-processing applications, such as the Windows Notepad, also add the characters. When reading a text file with CHARIN, interpret an ASCII sequence of 13 followed by 10 as the end of a line. As a convenience, Rexx has an environment symbol name .ENDOFLINE that contains the linend sequence used by the Stream class on the current system.

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.

Example 7.8. LINECHAR.CMD

```
/* 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=13 then       /* Carriage return?                */
      say "Carriage return"
   else if ascii_value=10 then  /* Line feed?                      */
      say "Line feed"
   else say byte ascii_value    /* Ordinary character              */
end /* do */
file~close                     /* Close the file                   */
```

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.

7.8. 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 carriage-return and line-feed characters to the end of the string. Here is an example:

Example 7.9. JACK.CMD

```
/* 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  */
"type binary.dat"              /* Use the TYPE command to display file */
```

Because new-line characters are not added, the text displayed by the TYPE 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.

7.9. Closing Files

If you do not explicitly close a file, Rexx closes the file when the Stream object is reclaimed by the garbage collector. This frequently does not occur until your program exits, so it is good practice to explicitly close files when you are finished with them.

7.10. 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.

Example 7.10. DIRECT.CMD

```
/* 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.

7.11. 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.

Example 7.11. ISTHERE.CMD

```
/* 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.

7.12. 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.

Example 7.12. INFOON.CMD

```
/* 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
```

7.13. 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 *Section 7.2, "Reading a Text File"*. 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 Windows window.

Example 7.13. Modified COUNT.CMD

```
                            /* 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:

Example 7.14. INEXAM.CMD

```
/* 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.

Example 7.15. INCHAR.CMD

```
/* 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:

```
[C:\]inchar
What is your name? John
Q.
Public
Hi John
Q.
Pu
```

7.14. Using Windows Devices

You can use Windows devices by substituting a device name (such as PRN, LPT1, LPT2, COM1, and so on) for the file name when you create a stream object. Then use line or character methods to read or write the device.

The following example sends data to a printer (device name PRN in the example). In addition to sending text data, the example also sends a control character for starting a new page. You can send

other control characters or escape sequences in a similar manner. (Generally, these are listed in the manual for the device.)

Usually the control characters are characters that you cannot type at the keyboard. To use them in your program, send a D2C message to the character's ASCII value as shown in the example.

Example 7.16. PRINTIT.CMD

```
/* PRINTIT.CMD - Prints a text file                           */
say "Type file name: "            /* prompt for a file name */
pull filename                     /* ...and get it from the user      */
infile=.stream~new(filename)
printer=.stream~new("prn:")

newpage = 12~d2c                  /* save page-eject character         */

/* Repeat this loop until no lines remain in the file */
/* and keep track of the line count with COUNT        */

do count = 1 until filename~lines = 0
   if printer~lineout(infile~linein) <>0 then do
      say "Error: unable to write to printer"
      leave
   end
   if count // 50 = 0 then                /* if the line count is a */
      printer~charout(newpage)            /* multiple of 50, then   */
                                          /* start a new page by    */
                                          /* sending the form feed  */

end                               /* go back to the start of loop   */
                                  /* until no lines remain          */

infile~close                      /* close the file                 */
exit                              /* end the program normally       */
```

Rexx C++ Application Programming Interfaces

This chapter describes how to interface applications to Rexx or extend the Rexx language by using Rexx C++ application programming interfaces (APIs). As used here, the term application refers to programs written in C++.

The features described here let a C++ application extend many parts of the Rexx language or extend an application with Rexx. This includes creating handlers for Rexx methods, external functions, and system exits.

Rexx methods
> are methods for Rexx classes written in C++. The methods reside in dynamically loaded external shared libraries.

Functions
> are function extensions of the Rexx language written in C++. Like the native methods, functions are packaged in external libraries. Functions can be general-purpose extensions or specific to an application.

Command Handlers
> are programmer-defined handlers for named command environments. The application programmer can tailor the Rexx interpreter behavior by creating named command environments to interfacing with application environments.

System exits
> are programmer-defined variations of the interpreter. The application programmer can tailor the Rexx interpreter behavior by using the defined exit points to control Rexx resources.

Methods, functions, system exit handlers, and command handlers have similar coding, compilation, and packaging characteristics.

In addition, applications can call methods defined of Rexx objects and execute them from externally defined methods and functions.

8.1. Rexx Interpreter API

Rexx programs run in an environment controlled by an interpreter instance. An interpreter instance environment is created with an enable set of exit handlers and a customized environment. An instance may have multiple active threads and each interpreter instance has a unique version of the .local environment directory, allowing programs to run with some degree of isolation.

If you use the older RexxStart (*Section 9.3.3, "The RexxStart Function"*) API to run a Rexx program, the Rexx environment initializes, runs a single program, and the environment is terminated. With the RexxCreateInterpreter() API, you have fine grain control over how the environment is used. You are able to create a tailored environment, perform multiple operations (potentially, on multiple threads), create objects that persist for longer than the life of a single program, etc. An application can create an interpreter instance once, and reuse it to run multiple programs.

Interpreter environments are created using the RexxCreateInterpreter() (*Section 8.1.1, "RexxCreateInterpreter"*) API:

Example 8.1. API - Rexx CreateInterpreter

```
RexxInstance *instance;
RexxThreadContext *threadContext;
RexxOption options[25];

if (RexxCreateInterpreter(&instance, &threadContext, options)) {
…
}
```

Once you've created an interpreter instance, you can use the APIs provided by the RexxInstance or RexxThreadContext interface to perform operations like running programs, loading class packages, etc. For example, the following code will run a program using a created instance, checking for syntax errors upon completion:

Example 8.2. API - RexxInstance and RexxThreadContext

```
    // create an Array object to hold the program arguments
    RexxArrayObject args = threadContext->NewArray(instanceInfo->argCount);
    // we're passing a variable number of arguments, so we need to create
    // String objects and insert them into the array
    for (size_t i = 0; i < argCount; i++)
    {
        if (arguments[i] != NULL)
        {
            // add the argument to the array, if specified.  Note that ArrayPut() requires
  an
            // index that is origin-1, unlike C arrays which are origin-0.
            threadContext->ArrayPut(args, threadContext->String(arguments[i]), i + 1);
        }
    }

    // call our program, using the provided arguments.
    RexxObjectPtr result = threadContext->CallProgram("myprogram.rex", args);
    // if an error occurred, get the decoded exception information
    if (threadContext->CheckCondition())
    {
        RexxCondition condition;

        // retrieve the error information and get it into a decoded form
        RexxDirectoryObject cond = threadContext->GetConditionInfo();
        threadContext->DecodeConditionInfo(cond, &condition);
        // display the errors
        printf("error %d: %s\n%s\n", condition.code, threadContext-
>CString(condition.errortext),
            threadContext->CString(condition.message));
    }
    else
    {
        // Copy any return value as a string
        if (result != NULLOBJECT)
        {
            CSTRING resultString = threadContext->CString(result);
            strncpy(returnResult, resultString, sizeof(returnResult));
        }
    }
    // make sure we terminate this first
    instance->Terminate();
```

The example above creates a Rexx String object for each program argument stores them in a Rexx array. It then uses CallProgram() (*Section 8.16.18, "CallProgram"*) to call "myprogram.rex", passing the array object as the program arguments. On return, if the program terminated with a Rexx SYNTAX error, it displays the error message to the console. Finally, if the program exited normally and returned a value, the ASCII-Z value of that result is copied to a buffer. As a final step, the interpreter instance is destroyed once we're finished using it.

8.1.1. RexxCreateInterpreter

RexxCreateInterpreter creates an interpreter instance and an associated thread context interface for the current thread.

Example 8.3. API - RexxInstance and RexxThreadContext

```
RexxInstance *instance;
RexxThreadContext *threadContext;
RexxOption options[25];

if (RexxCreateInterpreter(&instance, &threadContext, options)) {
…
}
```

Arguments

instance	The returned RexxInstance interface vector. The interface vector provides access to APIs that apply to the global interpreter environment.
threadContext	The returned RexxThreadContext interface vector for the thread that creates the interpreter instance. The thread context vector provides access to thread-specific services.
options	An array of RexxOption structures that control the interpreter instance initialization. See *Section 8.1.2, "Interpreter Instance Options"* for details on the available options.

Returns

1 (TRUE) if the interpreter instance was successfully created, 0 (FALSE) for any failure to create the interpreter.

8.1.2. Interpreter Instance Options

The third argument to RexxCreateInterpreter is an options array that sets characteristics of the interpreter instance. The **options** argument points to an array of RexxOption structures, and can be NULL if no options are required. Each RexxOption instance contains information for named options that can be specified in any order and even multiple times. The oorexxapi.h include file contains a #define for each option name. The information required by an option varies with each option type, and is specified using a ValueDescriptor struct to handle a variety of data types. An entry with a NULL option name terminates the option list. The available interpreter options are:

INITIAL_ADDRESS_ENVIRONMENT
Contains the ASCII-Z name of the initial address environment that will be used for all Rexx programs run under this instance.

Example 8.4. API - RexxOption

```
RexxOption options[2];

options[0].optionName = INITIAL_ADDRESS_ENVIRONMENT;
options[0].option = "EDITOR";
options[1].optionName = NULL;
```

APPLICATION_DATA
Contains a void * value that will be stored with the interpreter instance. The application data can be retrieved using the GetApplicationData() (*Section 8.16.44, "GetApplicationData"*) API. The application data pointer allows methods, functions, exits, and command handlers to recover access to globally defined application data.

Example 8.5. API - RexxOption

```
RexxOption options[2];

options[0].optionName = APPLICATION_DATA;
options[0].option = (void *)editorInfo;
options[1].optionName = NULL;
```

EXTERNAL_CALL_PATH
Contains an ASCII-Z string defining an additional search path that is used when searching for Rexx program files. The call path string uses the format appropriate for the host platform environment. On Windows, the path elements are separated by semicolons (;). On Unix-based systems, a colon (:) is used.

Example 8.6. API - RexxOption

```
RexxOption options[2];

options[0].optionName = EXTERNAL_CALL_PATH;
options[0].option = myCallPath;
options[1].optionName = NULL;
```

EXTERNAL_CALL_EXTENSIONS
Contains an ASCII-Z string defining a list of extensions that will be used when searching for Rexx program files. The specified extensions must include the extension ".". Multiple extensions are separated by a comma (,).

Example 8.7. API - RexxOption

```
RexxOption options[2];

options[0].optionName = EXTERNAL_CALL_EXTENSIONS;
```

```
options[0].option = ".ed,.mac";  // add ".ed" and ".mac" to search path.
options[1].optionName = NULL;
```

LOAD_REQUIRED_LIBRARY

Specifies the name of an external native library that will be loaded once the interpreter instance is created. The library name is an ASCII-Z string with the library name in the same format used for ::REQUIRED LIBRARY. Multiple libraries can be loaded by specifying this option multiple times.

Example 8.8. API - RexxOption

```
RexxOption options[2];

options[0].optionName = LOAD_REQUIRED_LIBRARY;
options[0].option = "rxmath";
options[1].optionName = NULL;
```

REGISTER_LIBRARY

Specifies a package that will be registerd with the Rexx environment without loading an external library. The library is specified with a RexxLibraryPackage structure that gives the library name and a pointer to the associated RexxPackageEntry (*Section 8.11, "Building an External Native Library"*) table that describes the package contents. The library name is an ASCII-Z string with the library name in the same format used for ::REQUIRED LIBRARY. Multiple libraries can be registered by specifying this option multiple times.

Example 8.9. API - RexxOption

```
RexxOption options[2];
RexxLibraryPackage package;

package.registeredName = "mypackage";
package.table = packageTable;

options[0].optionName = REGISTER_LIBRARY;
options[0].option = (void *)&package;
options[1].optionName = NULL;
```

DIRECT_EXITS

Specifies a list of system exits that will be used with this interpreter instance. The exits are a list of RexxContextExit structs. Each enabled exit is specified in a single RexxContextExit struct that identifies exit type and handler entry point. The list is terminated by an instance using an exit type of 0. The direct exits are called using the RexxExitContext calling convention. See *Section 8.14, "Rexx Exits Interface"* for details.

Example 8.10. API - RexxOption

```
RexxContextExit exits[2];
RexxOption options[2];

exits[0].handler = functionExit;
exits[0].sysexit_code = RXOFNC;
```

```
exits[1].sysexit_code = 0;

options[0].optionName = DIRECT_EXITS;
options[0].option = (void *)exits;
options[1].optionName = NULL;
```

DIRECT_ENVIRONMENTS

Registers one or more subcommand handler environments with the interpreter instance. The handlers are a list of RexxContextEnvironment structs. Each enabled handler is specified in a single RexxContextEnvironment struct identifying the handler name and entry point. The list is terminated by an instance using a handler name of NULL. The direct environment handlers are called using the calling convention described in *Section 8.15, "Command Handler Interface"*.

Example 8.11. API - RexxOption

```
RexxContextEnvironment environments[2];
RexxOption options[2];


environments[0].handler = editorHandler;
environments[0].name = "EDITOR";
environments[1].name = NULL;

options[0].optionName = DIRECT_ENVIRONMENTS;
options[0].option = (void *)environments;
options[1].optionName = NULL;
```

REGISTERED_EXITS

Specifies a list of system exits that will be used with this interpreter instance. The exits are a list of RexxContextExit structs. Each enabled exit is specified in a single RexxContextExit struct identifying the type of the exit and the name of the registered exit handler. The list is terminated by an instance using an exit type of 0. The registered exits are called using the RexxExitHandler calling convention. See *Section 9.6, "Registered System Exit Interface"* for details.

Example 8.12. API - RexxOption

```
RXSYSEXIT exits[2];
RexxOption options[2];


exits[0].sysexit_name = "MyFunctionExit";
exits[0].sysexit_code = RXOFNC;
exits[1].sysexit_code = 0;

options[0].optionName = REGISTERED_EXITS;
options[0].option = (void *)exits;
options[1].optionName = NULL;
```

REGISTERED_ENVIRONMENTS

Registers one or more subcommand handler environments with the interpreter instance. The handlers are a list of RexxRegisteredEnvironment structs. Each enabled handler is specified in a single RexxRegisteredEnvironment struct identifying the name of the environment and the registered subcom handler name. The list is terminated by an instance using a handler name

of NULL. The direct environment handlers are called using the calling convention described in *Section 9.4, "Subcommand Interface"*.

Example 8.13. API - RexxOption

```
RexxRegisteredEnvironment environments[2];
RexxOption options[2];


environments[0].registeredName = "MyEditorName";
environments[0].name = "EDITOR";
environments[1].name = NULL;

options[0].optionName = REGISTERED_ENVIRONMENTS;
options[0].option = (void *)environments;
options[1].optionName = NULL;
```

8.2. Data Types Used in APIs

The ooRexx APIs rely on a variety of special C++ types for interfacing with the interpreter. Some of these types are specific to the Rexx language, while others are standard types defined by C++. Many of the APIs involve conversion between types, while others require values of a specific type as arguments. This section explains the different types and the rules for using these types.

8.2.1. Rexx Object Types

Open Object Rexx is fundamentally an object-oriented language. All data in the language (including strings and numbers) are represented by object instances. The ooRexx APIs use a number of opaque types that represent instances of Rexx built-in objects. The defined object types are:

RexxObjectPtr   a reference to a Rexx object instance. This is the root of object hierarchy and can represent any type of object.

RexxStringObject   an instance of the Rexx String class. The API set allows String objects to be created and manipulated.

RexxBufferStringObject   an instance of the Rexx String class that can be written into. Buffer strings are used for constructing String objects "in-place" to avoid needing to create a String from a separate buffer. RexxBufferStringObject instances must be finalized to be converted into a usable Rexx String object.

RexxArrayObject   An instance of a Rexx single-dimension array. Arrays are used in many places, and there are interfaces provided for direct array manipulation.

RexxDirectoryObject   An instance of Rexx Directory class. Like arrays, there are APIs provided for access and manipulating data stored in a directory.

RexxStemObject   An instance of the Rexx Stem class. The APIs include a number of utility routines for accessing and manipulating data in Stem objects.

RexxSupplierObject   An instance of the Rexx Supplier class.

RexxClassObject   An instance of the Rexx Class class.

RexxPackageObject   An instance of the Rexx Package class.

RexxMethodObject   An instance of the Rexx Method class.

RexxRoutineObject   An instance of the Rexx Routine class. Routine objects can be invoked directly from C++ code.

RexxPointerObject   A wrapper around a pointer value. Pointer objects are designed for constructing Rexx classes that interface with native code subsystems.

RexxBufferObject   An allocatable storage object that can be used for storing native C++ data. Buffer objects and the contained data are managed using the Rexx object garbage collector.

RexxMutableBufferObjectAn instance of the Rexx MutableBuffer class.

8.2.2. Rexx Numeric Types

The Routine and Method interfaces support a very complete set of C numeric types as arguments and return values. In addition, there are also APIs provided for converting between Rexx Objects and numeric types (and the reverse transformation as well). It is recommended that you allow the Rexx runtime and APIs to handle conversions between Rexx strings and numeric types to give behavior consistent with the Rexx built-in methods and functions.

In addition to a full set of standard numeric types, there are special types provided that implement the standard Rexx rules for numbers used internally by Rexx. These types are:

wholenumber_t   conversions involving the wholenumber_t conform to the Rexx whole number rules. Values are converted using the same internal digits value used by the built-in functions. For 32-bit versions, this is numeric digits 9, giving a range of 999,999,999 to -999,999,999. On 64-bit systems, numeric digits 18 is used, giving a range of 999,999,999,999,999,999 to -999,999,999,999,999,999.

stringsize_t   stringsize_t conversions also conform to the Rexx whole number rules, with the added restriction that the value must be a non-negative whole number value. The stringsize_t type is useful for arguments such as string lengths where only a non-negative value is allowed. The range for 32-bit versions is 999,999,999 to 0, and 999,999,999,999,999,999 to 0 on 64-bit platforms.

logical_t   a Rexx logical value. On conversion from a string value, this must be either '1' (true) or '0' (false). On conversion back to a string value, a non-zero binary value will be converted to '1' (true) and zero will become '0' (false).

A subset of the integer numeric types are of differing sizes depending on the addressing mode of the system you are compiling on. These types will be either 32-bits or 64-bits. The variable size types are:

size_t   An unsigned "size" value. This is the value type returned by pointer subtraction.

ssize_t   The signed equivalent to size_t.

uintptr_t   An unsigned integer value that's guaranteed to be the same size as a pointer value. Use an uintptr_t type if you wish to return a pointer value as a Rexx number.

intptr_t   A signed equivalent to uintptr_t.

The remainder of the numeric types have fixed sizes regardless of the addressing mode.

int   A 32-bit signed integer.

int32_t   A 32-bit signed integer. This is equivalent to int.

uint32_t   An unsigned 32-bit integer.

int64_t   A signed 64-bit integer.

uint64_t   An unsigned 64-bit integer.

int16_t	A signed 16-bit integer.
uint16_t	An unsigned 16-bit integer.
int8_t	A signed 8-bit integer.
uint8_t	An unsigned 8-bit integer.
float	A 32-bit floating point number. When used as an argument to a routine or method, the strings "nan", "+infinity", and "-infinity" will be converted into the appropriate floating-point values. The reverse conversion is used when converting floating-point values back into Rexx objects.
double	A 64-bit floating point number. The Rexx runtime applies the same special processing for nan, +infinity, and -infinity values as float types.

8.3. Introduction to API Vectors

The Rexx APIs operate through a set of interface vectors that define a set of interpreter services that are available. There are different interface vectors used for different contexts, but they use very similar calling concepts.

The first interface vector you'll encounter with the programming interfaces is the RexxInstance value returned by RexxCreateInterpreter. The RexxInstance type is defined as a struct when compiled for C code, or a C++ class when compiled for ++. The struct version looks like this:

Example 8.14. API - RexxInstance

```
struct RexxInstance_
{
    RexxInstanceInterface *functions;   // the interface function vector
    void *applicationData;              // creator defined data pointer
};
```

The field *applicationData* contains any value that was specified via the APPLICATION_DATA option on the RexxCreateInterpreter call. This provides easy access any application-specific data needed to interact with the interpreter. All other interface contexts will include a pointer to the RexxInstance structure, so it is always possible to recover this data pointer.

The *functions* field is a pointer to a second structure that defines the RexxInstance programming interfaces. The RexxInstance services are ones that may be called from any thread and in any context. The services are called using C function pointer fields in the interface structure. The RexxInstanceInterface looks like this:

Example 8.15. API - RexxInstanceInterface

```
typedef struct
{
    wholenumber_t interfaceVersion;    // The interface version identifier

    void        (RexxEntry *Terminate)(RexxInstance *);
    logical_t   (RexxEntry *AttachThread)(RexxInstance *, RexxThreadContext **);
    size_t      (RexxEntry *InterpreterVersion)(RexxInstance *);
    size_t      (RexxEntry *LanguageLevel)(RexxInstance *);
    void        (RexxEntry *Halt)(RexxInstance *);
    void        (RexxEntry *SetTrace)(RexxInstance *, logical_t);
} RexxInstanceInterface;
```

The first thing to note is the interface struct contains a field named *interfaceVersion*. The interfaceVersion field is a version marker that defines the services the interpreter version supports. This interface version is incremented any time new functions are added to the interface. Using the interface version allows application code to reliably check that required interface functions are available.

The remainder of the fields are functions that can be called to perform RexxInstance operations. Note that the first argument to all of the functions is a pointer to a RexxInstance structure. A call to the InterpreterVersion API from C code would look like this:

```
size_t version = context->functions->InterpreterVersion(context);
```

When using C++ code, the RexxThreadContext struct has convenience methods that simplify calling these functions:

```
size_t version = context->InterpreterVersion();
```

Note that in the C++ call, it is no longer necessary to pass the RexxInstance as the first object. That's handled automatically by the C++ method.

The RexxThreadContext pointer returned from RexxCreateInterpreter() functions the same way. RexxThreadContext looks like this:

Example 8.16. API - RexxThreadContext

```
struct RexxThreadContext_
{
    RexxInstance *instance;          // the owning instance
    RexxThreadInterface *functions;  // the interface function vector
}
```

The RexxThreadContext struct contains an embedded RexxInstance pointer for the associated interpreter instance. It also contains an interface vector for the functions available with a RexxThreadContext. The RexxThreadInterface vector has its own version identifier and function pointer for each of the defined services. The RexxThreadContext functions all require a RexxThreadContext pointer as the first argument. The RexxThreadContext class also defines C++ convenience methods for accessing its own functions and the functions for the RexxInstance as well. For example, to call the InterpreterVersion() API using a RexxThreadContext from C code, it is necessary to code

```
size_t version = context->instance->functions->InterpreterVersion(context->instance);
```

The C++ version is simply

```
// context is a RexxThreadContext *
size_t version = context->InterpreterVersion();
```

When the Rexx interpreter makes calls to native code routines and methods, or invokes exit handlers, the calls use context structures specific to the call context. These are the RexxCallContext

(*Section 8.9, "Rexx Call Context Interface"*), RexxMethodContext (*Section 8.8, "Rexx Method Context Interface"*), and RexxExitContext (*Section 8.10, "Rexx Exit Context Interface"* structures. Each structure contains a pointer to a RexxThreadContext instance that's valid until the call returns. Through the embedded RexxThreadContext, each call may use any of the RexxThreadContext or RexxInstance functions in addition to the context-specific functions. Each context defines C++ methods for the embedded RexxInstance and RexxThreadContext functions.

Note that the RexxInstance interface can be used at any time and on any thread. The RexxThreadContext returned by RexxCreateInterpreter() can only be used on the same thread as the RexxCreateInterpreter() call, but is not valid for use in the context of a method, routine, or exit call-out. In those contexts, the RexxThreadContext instance passed to the call-out must be used. A RexxThreadContext instance created for a call-out is only valid until the call returns to the interpreter.

8.4. Threading Considerations

When using RexxCreateInterpreter() (*Section 8.1.1, "RexxCreateInterpreter"*) to create a new interpreter instances, a RexxThreadContext pointer is returned with the interpreter instance. The thread context vector allows you to perform operations such as running Rexx programs while in the same thread context as the RexxCreateInterpreter() call.

A given interpreter instance can process calls from multiple threads, but a RexxThreadContext instance must be obtained for each additional thread you wish to use. A new thread context is obtained by calling AttachThread() using the RexxInstance API vector returned from RexxCreateInterpreter(). Once a valid RexxThreadContext interface has been created for the thread, any of the thread context operations may be used from that thread. Before the thread terminates, the DetachThread() (*Section 8.16.24, "DetachThread"*) API must be called to remove the attached thread from the interpreter instance.

The interpreter is capable of asynchronous calls to interpreter APIs from signal or event handlers. When called in this manner, it is possible that AttachThread will be called while running on a thread that is already attached to the interpreter instance. When a nested AttachThread() (*Section 8.16.13, "AttachThread"*) call is made, the previous thread context is suspended and the newly created thread context is the active context for the source thread. It is very important that DetachThread() be called to restore the original thread context before you return from the signal handler.

8.5. Garbage Collection Considerations

When any context API has a return result that is a Rexx object instance, the source API context will protect that object instance from garbage collection for as long as the context is valid. Once the API context is destroyed, the accessed objects might become eligible for garbage collection and be reclaimed by the interpreter runtime. These object references are only valid until the current context is destroyed. They cannot be stored in native code control blocks and be used in other thread contexts. If you wish to store object references so that they can be accessed by other thread contexts, you can create a globally valid object reference using the RequestClobalResource() (*Section 8.16.141, "RequestGlobalReference"* API. A global reference will protect the object from the garbage collector until the interpreter instance is terminated. Protecting the object will also protect any objects referenced by the protected object. For example, using RequestGlobalReference() to protect a Directory object will also protect all of the directory keys and values. The global reference can be used with any API context valid for the same interpreter instance. Once you are finished with a locked object, ReleaseGlobalResource() (*Section 8.16.139, "ReleaseGlobalReference"*) removes the object lock and makes the object eligible for garbage collection.

On the flip side of this, sometimes it is desirable to remove the local API context protection from an object. For example, if you use the ArrayAt() API to iterate through all of the elements of an Array, each object ArrayAt() returns will be added to the API context's protection table. There is a

small overhead associated with each protected reference, so iterating through a large array would accumulate that overhead for each array element. Using ReleaseLocalReference() (*Section 8.16.140, "ReleaseLocalReference"* on an object reference you no longer require will remove the local lock, and thus limit the overhead associated with tracking the object references.

8.6. Rexx Interpreter Instance Interface

The Interpreter Instance API is defined by the RexxInstance interface vector. The RexxInstance defines methods that affect the global state of the interpreter instance. Most of the instance APIs can be called from any thread without requiring any extra steps to access the instance. The two most important instance operations are AttachThread() (*Section 8.16.13, "AttachThread"*) and Terminate() (*Section 8.16.169, "Terminate"*). AttachThread() allows additional externally identified threads to be included in the interpreter instance threadpool. AttachThread returns a RexxThreadContext (*Section 8.7, "Rexx Thread Context Interface"*) interface vector that enables a wider range of capability for the attached thread. The Terminate() API shuts down an interpreter instance when it is no longer needed.

8.7. Rexx Thread Context Interface

The RexxThreadContext interface vector provides a very wide range of functions to your application code. There are roughly 125 functions defined on a RexxThreadContext. Among the services provided are:

- Running Rexx programs

- Loading Rexx packages

- Invoking methods of Rexx objects

- Converting between objects and various C++ types

- Creating and manipulating common Rexx object types

- Raising/handling Rexx syntax errors

The C++ methods defined on a RexxThreadContext C++ object include the methods defined by the RexxInstance (*Section 8.6, "Rexx Interpreter Instance Interface"*) class, so the single context vector is used to access both thread context and interpreter instance APIs.

A RexxThreadContext instance is returned with the original RexxCreateInterpreter() (*Section 8.1.1, "RexxCreateInterpreter"*) call that created the interpreter instance. The AttachThread() (*Section 8.16.13, "AttachThread"* method will create a RexxThreadContext instance for additional threads that you add to an interpreter instance. Additionally, the RexxMethodContext (*Section 8.8, "Rexx Method Context Interface"*), RexxCallContext (*Section 8.9, "Rexx Call Context Interface"*), and RexxExitContext (*Section 8.10, "Rexx Exit Context Interface"*) objects embed a RexxThreadContext object the same way that a RexxThreadContext object embeds a RexxInstance object.

8.8. Rexx Method Context Interface

A RexxMethodContext object is included as an argument to any native C++ methof (*Section 8.13, "Defining Library Methods"*) defined in external libraries. The method context provides services that are specific to a method call, including:

- Accessing method-specific values such as SELF, SUPER, etc.

- Manipulating object instance variables

- Forwarding messages

- Manipulating GUARD state

- Locating classes defined in the method's package scope

In addition to the method-specific functions, the RexxMethodContext object has an embedded a RexxThreadContext (*Section 8.7, "Rexx Thread Context Interface"*) object created specifically for this environment. The RexxThreadContext provides a large number of additional methods to the method environment.

API calls made using the RexxMethodContext APIs may cause Rexx syntax errors or other condition to be raised. These calls are invoked as if the current context is operating with SIGNAL ON ALL enabled. Any conditions will be trapped and held in a pending condition until the current context returns. At the return, if a condition is still pending, the appropriate condition is reraised in the caller's context. These errors can be checked using the CheckCondition() (*Section 8.16.20, "CheckCondition"*) API, and pending conditions can be cancelled using ClearCondition() (*Section 8.16.21, "ClearCondition"*).

8.9. Rexx Call Context Interface

A RexxCallContext object is included as an argument to any native C++ routine (*Section 8.12, "Defining Library Routines"*) defined in external libraries. The call context provides services that are specific to a routine call, including:

- Accessing caller context specific values such as the current numeric settings

- Manipulating variables in the caller's variable context

- Locating classes defined in the routine's package scope

In addition to the call-specific functions, the RexxCallContext object has an embedded a RexxThreadContext (*Section 8.7, "Rexx Thread Context Interface"*) object created specifically for this environment. The RexxThreadContext provides a large number of additional methods to the call environment.

API calls made using the RexxCallContext APIs may cause Rexx syntax errors or other condition to be raised. These calls are invoked as if the current context is operating with SIGNAL ON ALL enabled. Any conditions will be trapped and held in a pending condition until the current context returns. At the return, if a condition is still pending, the appropriate condition is reraised in the caller's context. These errors can be checked using the CheckCondition() (*Section 8.16.20, "CheckCondition"*) API, and pending conditions can be cancelled using ClearCondition() (*Section 8.16.21, "ClearCondition"*).

8.10. Rexx Exit Context Interface

A RexxExitContext object is included as an argument to any systemexit (*Section 8.14.2, "Context Exit Definitions"*) or command handler (*Section 8.15, "Command Handler Interface"*). The exit context provides services that are specific to a exit call, including:

- Accessing caller context specific values such as the current numeric settings

- Manipulating variables in the caller's variable context

In addition to the exit-specific functions, the RexxExitContext object has an embedded a RexxThreadContext (*Section 8.7, "Rexx Thread Context Interface"*) object created specifically for this environment. The RexxThreadContext provides a large number of additional methods to the exit environment.

API calls made using the RexxExitContext APIs may cause Rexx syntax errors or other condition to be raised. These calls are invoked as if the current context is operating with SIGNAL ON ALL enabled. Any conditions will be trapped and held in a pending condition until the current context returns. At the return, if a condition is still pending, the appropriate condition is reraised in the caller's context. These errors can be checked using the CheckCondition() (*Section 8.16.20, "CheckCondition"*) API, and pending conditions can be cancelled using ClearCondition() (*Section 8.16.21, "ClearCondition"*).

8.11. Building an External Native Library

External libraries written in compiled languages (typically C or C++) provide a means to interface Rexx programs with other subsystems intended for compiled languages. These libraries are packaged as Dynamic Link Libraries on Windows or shared libraries on Unix-based systems. A named library can be loaded using the ::REQUIRES directive, the loadLibrary() method on the Package class, or by using the EXTERNAL keyword on a ::ROUTINE, ::METHOD, or ::ATTRIBUTE directive.

When the library is loaded, the interpreter searches for an entry point in the library named RexxGetPackage(). An external library package is required to provide a RexxGetPackage() function that returns a pointer to the descriptor structure defining the methods and routines contained within the library. The RexxGetPackage() routine takes no arguments and has a RexxPackageEntry *return value. This is normally created using the OOREXX_GET_PACKAGE() macro defined in the oorexxapi.h include file.

```
// package loading stub.
OOREXX_GET_PACKAGE(package);
```

Where *package* is the name of the RexxPackageEntry table for this library. The package entry table is a descriptor contained within the library. Note that on Windows, it is necessary to explicitly export the RexxPackageEntry() function when the library is linked. This is the only name you are required to export. Calls are made to the library routines and methods using addresses stored in the RexxPackageEntry table.

The RexxPackageEntry structure contains information about the package and descriptors of any methods and/or routines defined within the package. The structure looks like this:

Example 8.17. API - RexxPackageEntry

```
typedef struct _RexxPackageEntry
{
    int size;                     // size of the structure...helps compatibility
    int apiVersion;               // version this was compiled with
    int requiredVersion;          // minimum required interpreter version (0 means any)
    const char *packageName;      // package identifier
    const char  *packageVersion;  // package version #
    RexxPackageLoader loader;     // the package loader
    RexxPackageUnloader unloader; // the package unloader
    struct _RexxRoutineEntry *routines; // routines contained in this package
    struct _RexxMethodEntry *methods;   // methods contained in this package
} RexxPackageEntry;
```

The fields in the RexxPackageEntry have the following functions:

size and *apiVersion*
    these fields give the size of the received table and identify the interpreter level this library has been compiled against. These indicators will allow additional information to be added to the

RexxPackageEntry in the future without causing compatibility issues for older libraries. Normally, these two fields are defined using the STANDARD_PACKAGE_HEADER macro, which sets both values.

requiredVersion

a library can specify the minimum interpreter level it requires. The interpreter will only load libraries that match the minimum compatibility requirement of the library package. A zero value in this field indicates there is no minimum level requirement. The macro REXX_CURRENT_INTERPRETER_VERSION will set the level of interpreter you are compiling against. If REXX_CURRENT_INTERPRETER_VERSION is specified, then the library package will not load with older releases. The API header files will be updated with a macro for each interpreter version. The version macros are of the form REXX_INTERPRETER_*version_level_revision*, where *version*, *level*, and *revision* refer to the corresponding values in an interpreter release number. For example, REXX_INTERPRETER_4_0_0 would indicate that the 4.0.0 interpreter level is the minimum this library requires.

packageName

a descriptive name for this library package.

packageVersion

a version string for this package. The version can be in whatever form is appropriate for the package.

packageLoader

a function that will be called when the library package is first loaded by the interpreter. The package loader function is passed a RexxThreadContext pointer, which will give the package access to Rexx interpreter services at initialization time. The package loader is optional and is indicated by a NULL value in the descriptor.

packageUnloader

a function that will be called when the library package is unloaded by the interpreter. The unloading process happens when the last interpreter instance is destroyed during the last cleanup stages. This gives the loaded library an opportunity to clean up any global resources such as cached Rexx object references. The package loader is optional and is indicated by a NULL value in the descriptor.

routines

a pointer to an array of RexxRoutineEntry structures that define the routines provided by this package. If there are no routines, this field should be NULL. See *Section 8.12, "Defining Library Routines"* for details on creating the exported routine table.

method

a pointer to an array of RexxMethodEntry structures that define the methods provided by this package. If there are no methods, this field should be NULL. See *Section 8.13, "Defining Library Methods"* for details on creating the exported method table.

Here is an example of a RexxPackageEntry table taken from the rxmath library package:

Example 8.18. API - RexxPackageEntry and RexxRoutineEntry

```
// now build the actual entry list
RexxRoutineEntry rxmath_functions[] =
{
    REXX_TYPED_ROUTINE(MathLoadFuncs, MathLoadFuncs),
    REXX_TYPED_ROUTINE(MathDropFuncs, MathDropFuncs),
```

```
        REXX_TYPED_ROUTINE(RxCalcPi,      RxCalcPi),
        REXX_TYPED_ROUTINE(RxCalcSqrt,    RxCalcSqrt),
        REXX_TYPED_ROUTINE(RxCalcExp,     RxCalcExp),
        REXX_TYPED_ROUTINE(RxCalcLog,     RxCalcLog),
        REXX_TYPED_ROUTINE(RxCalcLog10,   RxCalcLog10),
        REXX_TYPED_ROUTINE(RxCalcSinH,    RxCalcSinH),
        REXX_TYPED_ROUTINE(RxCalcCosH,    RxCalcCosH),
        REXX_TYPED_ROUTINE(RxCalcTanH,    RxCalcTanH),
        REXX_TYPED_ROUTINE(RxCalcPower,   RxCalcPower),
        REXX_TYPED_ROUTINE(RxCalcSin,     RxCalcSin),
        REXX_TYPED_ROUTINE(RxCalcCos,     RxCalcCos),
        REXX_TYPED_ROUTINE(RxCalcTan,     RxCalcTan),
        REXX_TYPED_ROUTINE(RxCalcCotan,   RxCalcCotan),
        REXX_TYPED_ROUTINE(RxCalcArcSin,  RxCalcArcSin),
        REXX_TYPED_ROUTINE(RxCalcArcCos,  RxCalcArcCos),
        REXX_TYPED_ROUTINE(RxCalcArcTan,  RxCalcArcTan),
        REXX_LAST_ROUTINE()
};

RexxPackageEntry rxmath_package_entry =
{
    STANDARD_PACKAGE_HEADER
    REXX_INTERPRETER_4_0_0,               // anything after 4.0.0 will work
    "RXMATH",                             // name of the package
    "4.0",                                // package information
    NULL,                                 // no load/unload functions
    NULL,
    rxmath_functions,                     // the exported functions
    NULL                                  // no methods in rxmath.
};

// package loading stub.
OOREXX_GET_PACKAGE(rxmath);
```

8.12. Defining Library Routines

The RexxRoutineEntry table defines routines that are exported by a library package. This table is an array of RexxRoutineEntry structures, terminated by an entry that contains nothing but zero values in the fields. The REXX_LAST_ROUTINE() macro will generate a suitable table terminator entry.

The remainder of the table will be entries generated via either the REXX_CLASSIC_ROUTINE() or REXX_TYPED_ROUTINE() macros. REXX_CLASSIC_ROUTINE() entries are for routines created using the older string-oriented function style. The classic routines allow packages to be migrated to the new package loading system without requiring a rewrite of all of the contained functions. See *Section 9.5, "External Function Interface"* for details on creating the functions in the classic style.

Routine table entries defined using REXX_TYPED_ROUTINE() use the new object-oriented interfaces for creating routines. These routines can use the interpreter runtime to convert call arguments from Rexx objects into primitive types and return values converted from primitive types back into Rexx objects. These routines are also given access to a rich set of services through the RexxCallContext (*Section 8.9, "Rexx Call Context Interface"*) interface vector.

The REXX_CLASSIC_ROUTINE() and REXX_TYPED_ROUTINE() macros take two arguments. The first entry is the package table name for this routine. The second argument is the entry point name of the real native code routine that implements the function. These names are frequently the same, but need not be. The package table name is the name this routine will be called with from Rexx code.

Smaller function packages frequently place all of the contained functions and the package definition tables in the same file, with the package tables placed near the end of the source file so all of the functions are visible. For larger packages, it may be desirable to place the functions in more than one source file. For functions packaged as multiple source files, it is necessary to create prototype

declarations so the routine entry table can be generated. The oorexxapi.h header file includes REXX_CLASSIC_ROUTINE_PROTOTYPE() and REXX_TYPED_ROUTINE_PROTOTYPE() macros to generate the appropriate declarations. For example,

Example 8.19. API - REXX_TYPED_ROUTINE_PROTYPE

```
// create function declarations for the linker
REXX_TYPED_ROUTINE_PROTOTYPE(RxCalcPi);
REXX_TYPED_ROUTINE_PROTOTYPE(RxCalcSqrt);

// now build the actual entry list
RexxRoutineEntry rxmath_functions[] =
{
    REXX_TYPED_ROUTINE(RxCalcPi,      RxCalcPi),
    REXX_TYPED_ROUTINE(RxCalcSqrt,    RxCalcSqrt),
    REXX_LAST_ROUTINE()
};
```

8.12.1. Routine Declarations

Library routines are created using a series of macros that create the body of the function. These macros define the routine arguments and return value in a form that allows the Rexx runtime to perform argument checking and conversions before calling the target routine. These macros are named "RexxRoutine*n*, where *n* is the number of arguments passed to your routine. For example,

Example 8.20. API - RexxRoutine2

```
RexxRoutine2(int, beep, wholenumber_t, frequency, wholenumber_t, duration)
{
    return Beep(frequency, duration);  /* sound beep                */
}
```

defines a *beep* routine that will be passed two *wholenumber_t* arguments (*frequency* and *duration*). The return value is an *int* value.

An argument can be made optional by prefixing the type with "OPTIONAL_". For example,

Example 8.21. API - RexxRoutine2

```
RexxRoutine2(int, beep, wholenumber_t, frequency, OPTIONAL_wholenumber_t, duration)
{
    return Beep(frequency, duration);  /* sound beep                */
}
```

would define a routine that takes two arguments. The first argument is required, but the second is optional. Any optional arguments, when omitted on a call, will be passed using a zero value appropriate to the type. The macros argumentExists(n) or argumentOmitted(n) can reliably test if an argument was passed. For example, argumentExists(2) tests if the *duration* argument was specified when beep() was called. The *n* value is origin 1.

In addition to the arguments passed by the caller, there are some special argument types that provide your routine with additional information. These special types will add additional arguments

to your native routine implementation. The argument value specified with argumentExists() or argumentOmitted() maps to the arguments passed to your C++ routine rather than the arguments in the originating Rexx call. See *Section 8.12.2, "Routine Argument Types"* for details on the special argument types.

All routine declarations have an undeclared special argument passed to the routine. This special argument is named *context*. The *context* is a pointer to a RexxCallContext value and provides access to all API functions valid from a routine context.

> **Note**
>
> void is not a valid return type for a routine. There must be a real return type specified on the routine declaration. If you wish to have a routine without a return value, declare the routine with a return type of RexxObjectPtr and return the value NULLOBJECT. Routines that do not return a real value may not be invoked as functions. Only the CALL instruction allows a return without a value.

8.12.2. Routine Argument Types

A routine argument or return value may be a numeric type (*Section 8.2.2, "Rexx Numeric Types"*) or an object type (*Section 8.2.1, "Rexx Object Types"*). For numeric types, the call arguments must be convertible from a Rexx object equivalent into the primitive value or an error will be raised. For optional numeric arguments, a zero value is passed for omitted values. When used as a return type, the numeric values are translated into an appropriate Rexx object value.

If an argument is an object type, some additional validation is performed on the arguments being passed. If an argument does not meet the requirements for a given object type, an error will be raised. If an object-type argument is optional and a value is not specified on the call, the value NULLOBJECT is passed to your routine. The supported object types and the special processing rules are as follows:

RexxObjectPtr

a reference to any Rexx object instance. Any arbitrary object type may be passed for a RexxObjectPtr argument.

RexxStringObject

an instance of the Rexx String class. The argument value must be a Rexx String value or convertible to a Rexx String value using the request('String') mechanism.

RexxArrayObject

An instance of a Rexx single-dimension array.

RexxClassObject

An instance of Rexx Class class.

RexxMutableBufferObject

An instance of Rexx MutableBuffer class.

RexxStemObject

An instance of the Rexx Stem class. For routine calls, a stem argument may be specified either using the stem variable name directly or giving the stem variable name as a quoted string. For example, for a routine defined using

```
RexxRoutine1(int, MyRoutine, RexxStemObject, stem)
```

the following calls are equivalent:

```
x = MyRoutine(a.)
x = MyRoutine('a.')
```

This special processing allows routines that currently access stem variables using the RexxVariablePool API to be more easily converted to the newer API set.

In addition to the numeric and object types, there are additional special types that provide additional information to the calling routine or perform common special conversions on argument values. The special types available to routines are:

CSTRING

The argument is passed as an ASCII-Z string. The source argument must be one that is valid as a RexxStringObject value. The RexxStringObject is converted into a pointer to an ASCII-Z string. This is equivalent to the value returned from the StringValue() (*Section 8.16.126, "ObjectToStringValue"*) API from a RexxStringObject value. For an optional CSTRING argument, a NULL pointer is provided when the argument is omitted.

When CSTRING is used as a return value, the ASCII-Z string value will be converted into a Rexx String object. The Rexx runtime does not free any memory associated with a CSTRING return value, so care must be taken to avoid memory leaks. Also, locally declared character buffers cannot be returned as the storage associated with buffer is no longer valid once your routine returns to the Rexx interpreter. CSTRING return values are best confined to returning C literal values. For example, the following is not valid:

Example 8.22. API - CString

```
RexxRoutine0(CSTRING, MyRoutine)
{
    ....
    char buffer[32];
    sprintf(buffer, "%d:%d", major, minor);
    return buffer;     // buffer is not valid once return executes
}
```

A RexxStringObject return value and the String() (*Section 8.16.157, "String"*) API is more appropriate in this situation.

Example 8.23. API - RexxStringObject

```
RexxRoutine0(RexxStringObject, MyRoutine)
{
    ....
    char buffer[32];
    sprintf(buffer, "%d:%d", major, minor);
    return context->String(buffer);     // creates a string object and returns it.
}
```

POINTER

> an "unwrapped" Pointer or Buffer string object. If the argument is a Pointer object, the wrapped
> pointer value is returned as a void * value.. If the argument is a Buffer object, then a pointer to the
> buffer's data area is returned. A NULL pointer is returned for an omitted OPTIONAL_POINTER
> argument.
>
> When POINTER is used as a routine return value, any pointer value can be returned. The Rexx
> runtime will wrap the pointer value in a Rexx Pointer object.

POINTERSTRING

> a pointer value that has been encoded in string form. The string value must be in the format
> "0xnnnnnnnn", where the digits are valid hexadecimal digits. On 64-bit platforms, the pointer
> value must be 16 digits long. The string value is converted into a void * value. A NULL pointer is
> returned for an omitted optional POINTERSTRING argument.
>
> When POINTERSTRING is used as a routine return value, any pointer value can be returned. The
> Rexx runtime will convert the pointer value back into an encoded string value.

NAME

> The name of the invoked routine, passed as a CSTRING. NAME is not valid as a return value.

ARGLIST

> A RexxArrayObject containing all arguments passed to the routine. This is equivalent to using
> Arg(1, 'A') from Rexx code. The returned array contains all of the routine arguments that were
> specified in the original call. Omitted arguments are empty slots in the returned array. In addition,
> if a routine has an ARGLIST argument specified, the normal check for the maximum number of
> arguments is bypassed. This makes possible routines with an open-ended number of arguments.
> ARGLIST is not valid as a return value.

8.13. Defining Library Methods

The RexxMethodEntry table defines method that are exported by a library package. This table is an
array of RexxMethodEntry structures, terminated by an entry that contains nothing but zero values in
the fields. The REXX_LAST_METHOD() macro will generate a suitable table terminator entry.

The remainder of the table will be entries generated via the REXX_METHOD() macro. Routine table
entries defined using REXX_METHOD() use the object-oriented interfaces for creating methods
that can be defined on Rexx classes. These methods can use the interpreter runtime to convert call
arguments from Rexx objects into primitive types and return values from primitive types back into Rexx
objects. Native methods are also given access to a rich set of services via the RexxMethodContext
interface vector.

The REXX_METHOD() macro take two arguments. The first entry is the package table name for this
method. The second argument is the entry point name of the real native code method that implements
the function. These names are frequently the same, but need not be.

Smaller function packages frequently place all of the contained functions and the package definition
tables in the same file, with the package tables placed near the end of the source file so all of the
methods are visible. For larger packages, it may be desirable to place the methods in more than
one source file. For libraries packaged as multiple source files, it is necessary to create a prototype
declarations so the method entry table can be generated. The oorexxapi.h header file includes a
REXX_METHOD_PROTOTYPE() macro to generate the appropriate declarations. For example,

Example 8.24. API - REXX_METHOD_PROTOTYPE

```
// create function declarations for the linker
REXX_METHOD_PROTOTYPE(point_init);
REXX_METHOD_PROTOTYPE(point_add);

// now build the actual entry list
RexxMethodEntry point_methods[] =
{
    REXX_METHOD(point_init, point_init),
    REXX_METHOD(point_add,  point_add),
    REXX_LAST_METHOD()
};
```

8.13.1. Method Declarations

Library methods are created using a series of macros that create the body of the method. These macros define the method arguments and return value in a form that allows the Rexx runtime to perform argument checking and conversions before calling the target method. These macros are named "RexxMethod*n*, where *n* is the number of arguments you wish to be passed to your method. For example,

Example 8.25. API - RexxMethod2

```
RexxMethod2(int, beep, wholenumber_t, frequency, wholenumber_t, duration)
{
    return Beep(frequency, duration);  /* sound beep                */
}
```

defines a *beep* method that will be passed two *wholenumber_t* arguments (*frequency* and *duration*). The return value is an *int* value.

An argument can be made optional by prefixing the type with "OPTIONAL_". For example,

Example 8.26. API - RexxMethod2

```
RexxMethod2(int, beep, wholenumber_t, frequency, OPTIONAL_wholenumber_t, duration)
{
    return Beep(frequency, duration);  /* sound beep                */
}
```

would define a method that takes two arguments. The first argument is required, but the second is optional. Any omitted optional arguments will be passed using a zero value appropriate for the type. The macros argumentExists(n) or argumentOmitted(n) can reliably test if an argument was passed. For example, argumentExists(2) tests if the *duration* argument was specified when calling the beep() method. The *n* value is origin 1.

In addition to the arguments passed by the caller, there are some special argument types that provide your routine with additional information. These special types will add additional arguments to your native routine implementation. The argument position specified with argumentExists() or argumentOmitted() maps to the arguments passed to your C++ routine rather than the arguments in the originating Rexx call. See below for details on the special argument types.

All method declarations have an undeclared special argument passed to the routine. This special argument is named *context*. The *context* is a pointer to a RexxMethodContext (*Section 8.8, "Rexx Method Context Interface"*) value and provides access to all APIs valid from a method context.

> **Note**
>
> void is not a valid return type for a method. There must be a real return type specified on the method declaration. If you wish to have a method without a return value, declare the method with a return type of RexxObjectPtr and return the value NULLOBJECT. Methods that do not return a real value may not be invoked within expression, but may be used as standalone message instructions.

8.13.2. Method Argument Types

A method argument or return value may be a numeric type (*Section 8.2.2, "Rexx Numeric Types"*) or an object type (*Section 8.2.1, "Rexx Object Types"*). For numeric types, the arguments must be convertible from a Rexx object equivalent into the primitive value or an error will be raised. For optional numeric arguments, a zero value is passed for omitted values. When used as a return type, the numeric values are translated into an appropriate Rexx object value.

If an argument is an object type, some additional validation is performed on the arguments being passed. If an argument does not meet the requirements for a given object type, an error will be raised. If an object-type argument is optional and a value is not specified on the call, the value NULLOBJECT is passed to your routine. The supported object types and the special processing rules are as follows:

RexxObjectPtr

    a reference to any Rexx object instance. Any arbitrary object type may be passed for a RexxObjectPtr argument.

RexxStringObject

    an instance of the Rexx String class. The argument value must be a Rexx String value or convertible to a Rexx String value using the request('String') mechanism.

RexxArrayObject

    An instance of a Rexx single-dimension array.

RexxClassObject

    An instance of Rexx Class class.

RexxMutableBufferObject

    An instance of Rexx MutableBuffer class.

RexxStemObject

    An instance of Rexx Stem class. To pass a Stem to a method, a stem argument must be specified using a stem variable name directly. For example, for a method defined using

```
RexxMethod1(int, MyMethod, RexxStemObject, stem)
```

    the following call passes a stem object associated with a stem variable to the method:

```
x = o~myMethod(a.)
```

In addition to the numeric and object types, there are additional special types that provide additional information to the calling routine or perform common special conversions on argument values. The special types available to routines are:

CSTRING

The argument is passed as an ASCII-Z string. The source argument must be one that is valid as a RexxStringObject value. The RexxStringObject is converted into a pointer to an ASCII-Z string. This is equivalent to the value returned from the StringValue() (*Section 8.16.126, "ObjectToStringValue"*) API from a RexxStringObject value. For an optional CSTRING argument, a NULL pointer is provided when the argument is omitted.

When CSTRING is used as a return value, the ASCII-Z string value will be converted into a Rexx String object. CSTRING return values are best confined to returning C literal values. The Rexx runtime does not free any memory associated with a CSTRING return value, so care must be taken to avoid memory leaks. Also, locally declared character buffers cannot be returned as the storage associated with buffer is no longer valid once your method returns to the Rexx interpreter. CSTRING return values are best confined to returning C literal values. For example, the following is not valid:

Example 8.27. API - CString

```
RexxMethod0(CSTRING, MyMethod)
{
   ....
   char buffer[32];
   sprintf(buffer, "%d:%d", major, minor);
   return buffer;     // buffer is not valid once return executes
}
```

A RexxStringObject return value and the String() API is more appropriate in this situation.

Example 8.28. API - RexxStringObject

```
RexxMethod0(RexxStringObject, MyMethod)
{
   ....
   char buffer[32];
   sprintf(buffer, "%d:%d", major, minor);
   return context->String(buffer);     // creates a string object and returns it.
}
```

POINTER

an "unwrapped" Pointer or Buffer string object. If the argument is a Pointer object, the wrapped pointer value is returned as a void * value.. If the argument is a Buffer object, then a pointer to buffer's storage area is returned. A NULL pointer is returned for an omitted optional POINTER argument.

When POINTER is used as a method return value, any pointer value can be returned. The Rexx runtime will wrap the pointer value in a Rexx Pointer object.

POINTERSTRING

a pointer value that has been encoded in string form. The string value must be in the format "0xnnnnnnnn", where the digits are valid hexadecimal digits. On 64-bit platforms, the pointer

value must be 16 digits long. The string value is converted into a void * value. A NULL pointer is returned for an omitted optional POINTERSTRING argument.

When POINTERSTRING is used as a method return value, any pointer value can be returned. The Rexx runtime will convert the pointer value back into an encoded string value.

NAME

The name of the invoked method, passed as a CSTRING. This is the message name that was used to invoke the method. NAME is not valid as a return value.

ARGLIST

A RexxArrayObject containing all arguments passed to the method. This is equivalent to using Arg(1, 'A') from Rexx code. The returned array contains all of the method arguments that were specified in the original call. Omitted arguments are empty slots in the returned array. In addition, if a method has an ARGLIST argument specified, the normal check for the maximum number of arguments is bypassed. This makes possible methods with an open-ended number of arguments. ARGLIST is not valid as a return value.

OSELF

A RexxObjectPtr containing a reference to the object that was the message target for the current method. This is equivalent to the SELF variable that is available in Rexx method code. OSELF is not valid as a return value.

SUPER

A RexxClassObject containing a reference to the super scope object for the current method. This is equivalent to the SUPER variable that is set in Rexx method code. SUPER is not valid as a return value.

SCOPE

A RexxObjectPtr containing a reference to the current method's owning scope. This is normally the class that defined the method currently being executed. SCOPE is not valid as a return value.

CSELF

CSELF is a special argument type used for classes to store native pointers or structures inside an object instance. When a CSELF type is encountered, the runtime will search all of the object's variable scopes for an instance variable named CSELF. If a CSELF variable is located and the value is an instance of either the Pointer or Buffer class, the POINTER value will be passed to the method as a void * value. Objects that rely on CSELF values typically set the variable CSELF inside an init method for the object. For example:

Example 8.29. API - CSELF

```
RexxMethod2(RexxObjectPtr, stream_init, OSELF, self, CSTRING, name)
{
    // create a new stream info member
    StreamInfo *stream_info = new StreamInfo(self, name);
    RexxPointerObject streamPtr = context->NewPointer(stream_info);
    context->SetObjectVariable("CSELF", streamPtr);

    return NULLOBJECT;
}
```

Then, within other methods for the object, when the CSELF variable is used as an argument to the method, the void * is retrieved and cast to the correct type:

Example 8.30. API - CSELF

```
RexxMethod3(size_t, stream_charout, CSELF, streamPtr, OPTIONAL_RexxStringObject, data,
 OPTIONAL_int64_t, position)
{
    StreamInfo *stream_info = (StreamInfo *)streamPtr;
    stream_info->setContext(context, context->False());

    ...
}
```

CSELF is not valid as a return value.

8.13.3. Pointer, Buffer, and CSELF

Methods written in C++ frequently need to acquire access to data that is associated with an object instance. ooRexx provides two classes, Buffer and Pointer, that allow these associations to be made. Both classes are real Rexx classes that can be passed as arguments, returned as method results, and assigned to object instance variables. For the Rexx programmer who might encounter one of these instances, these are opaque objects that don't appear to be of much use. To the native library writer, the usefulness derives from what is stored inside these objects.

8.13.3.1. The Buffer class

The Buffer class allows the library writer to allocate blocks of memory from the Rexx object space. The memory is a part of the Buffer object instance, and will be reclaimed automatically when the Buffer object is garbage collected. This means the programmer does not need to explicitly release a Buffer object. It does, however, require that steps be taken to protect the Buffer object from garbage collection while it is still needed. The usual protection mechanism it to store the buffer object in an object instance variable using SetObjectVariable() (*Section 8.16.152, "SetObjectVariable"*). Once assigned to a variable, the Buffer is protected from garbage collection until its associated object instance is also reclaimed. The buffer is part of the internal state of the the object.

Buffer objects are allocated using the NewBuffer() function (*Section 8.16.105, "NewBuffer"*) that's part of the RexxThreadContext interface. Once created, you can access the Buffer's data area using BufferData() (*Section 8.16.14, "BufferData"*), which returns a pointer to the beginning of the data buffer. The data buffer area is writeable storage, into which any data may be placed. This is frequently used to allocate a C++ struct or class instance that is the native embodiment of the class implementation. For example

Example 8.31. API - RexxBufferObject

```
RexxMethod0(RexxObjectPtr, myclass_init)
{
    // create a buffer for my internal data.
    RexxBufferObject data = context->NewBuffer(sizeof(MyDataClass));
    // store this someplace safe
    context->SetObjectVariable("MYDATA", data);
    // get access to the data area
    void *dataPtr = context->BufferData(data);
    // construct a C++ object to place in the buffer
    MyDataClass *myData = new (dataPtr) MyDataClass();
    // initialize the data below
```

```
        ...

        return NULLOBJECT;
    }
```

This example allocates a Buffer object instance, creates a C++ class in its data area, and stores a reference to the Buffer in the MYDATA object variable. Other C++ methods can access this instance by using the C++ equivalent to the Rexx EXPOSE instruction.

Example 8.32. API - RexxBufferObject

```
RexxMethod0(RexxObjectPtr, myclass_dosomething)
{
    // retrieve my instance buffer
    RexxBufferObject data = (RexxBufferObject)context->GetObjectVariable("MYDATA");
    // Get the data pointer and cast it back to my class type
    MyDataClass *myData = (MyDataClass *)context->BufferData(data);
    // perform the operation below
    ...
}
```

Since Buffer object instances are reclaimed automatically when the object is garbage collected, no additional steps are required to cleanup that memory. However, if there are additional dymanically allocated resources associated with the Buffer, such as pointers to system allocated resources or dynamically allocated memory, it may be necessary to add an UNINIT method to your class to ensure the resources are not leaked.

Example 8.33. API - RexxBufferObject

```
RexxMethod0(RexxObjectPtr, myclass_uninit)
{
    // retrieve my instance buffer
    RexxBufferObject data = context->GetObjectVariable("MYDATA");
    // Get the data pointer and cast it back to my class type
    MyDataClass *myData = (MyDataClass *)context->BufferData(data);
    // delete any resources I've obtained (but not the MyDataClass
    // instance itself
    delete ((void *)myData) myData;
}
```

8.13.3.2. The Pointer class

The Pointer class has uses similar to the Buffer class, but Pointer instances only hold a single pointer value to native C/C++ resources. A Pointer instance is effectively a Buffer object where the buffer data area is a single void * pointer. Like Buffer objects, Pointers can be stored in Rexx variables and retrieved in native methods. Pointer object instances are garbage collected just like Buffer objects, but when a Pointer is reclaimed, whatever value refererenced by the Pointer instance are not cleaned up. If additional cleanup is required, then it will be necessary to implement an UNINIT method to handle the cleanup. Here are the Buffer examples above reworked for the Pointer class:

Example 8.34. API - RexxObjectPtr

```
RexxMethod0(RexxObjectPtr, myclass_init)
{
    // construct a C++ object to associate with the object
    MyDataClass *myData = new MyDataClass();
    // create a Pointer to store this in the object
    RexxPointerObject data = context->NewPointer(myData);
    // store this someplace safe
    context->SetObjectVariable("MYDATA", data);
    // initialize the data below
    ...

    return NULLOBJECT;
}


RexxMethod0(RexxObjectPtr, myclass_dosomething)
{
    // retrieve my instance data
    RexxPointerObject data = (RexxPointerObject)context->GetObjectVariable("MYDATA");
    // Get the data pointer and cast it back to my class type
    MyDataClass *myData = (MyDataClass *)context->PointerValue(data);
    // perform the operation below
    ...
}


RexxMethod0(RexxObjectPtr, myclass_uninit)
{
    // retrieve my instance data
    RexxPointerObject data = (RexxPointerObject)context->GetObjectVariable("MYDATA");
    // Get the data pointer and cast it back to my class type
    MyDataClass *myData = (MyDataClass *)context->PointerValue(data);
    // delete the backing instance
    delete myData;
}
```

8.13.3.3. The POINTER method type

The Rexx runtime has some special support for Pointer and Buffer objects when they are passed as method arguments and also when used as return values. The RexxMethod macros used to define method instances support the POINTER special argument type. When an argument is defined as a POINTER, then the argument value must be either a Buffer object or a Pointer object. The Rexx runtime will automatically pass this argument to the native method as the Buffer BufferData() value or the Pointer PointerValue() value, thus removing the need to unwrapper these in the method code. The POINTER type is generally used for private methods of a class where the Rexx versions of the methods pass Pointer or Buffer references to the private native code. For example, the Rexx code might look like this:

Example 8.35. API - RexxObjectPtr

```
::method setTitle
  expose title prefix handle
  use arg title
  // set the title to the title concatenated to the prefix
  self~privateSetTitle(handle, prefix title)
```

```
::method privateSetTitle PRIVATE EXTERNAL "LIBRARY mygui setTitle"
```

The corresponding C++ method would look like this:

Example 8.36. API - RexxObjectPtr

```
RexxMethod2(RexxObjectPtr, setTitle, POINTER, handle, CSTRING, title)
{
    // the pointer object was unwrapped for me
    MyWindowHandle *myHandle = (MyWindowHandle *)handle;

    // other stuff here
}
```

When POINTER is used as a method return type, the runtime will automatically create a Pointer object instance that wrappers the returned void *value. The created Pointer instance is the result returned to the Rexx code.

8.13.3.4. The CSELF method type

There's one additional concept using Pointer and Buffer objects supported by the C++ APIs. When a method definition specifies the special type CSELF, the runtime will look for an object variable named CSELF. If the variable is found, and if the variable is assigned to an instance of Pointer or Buffer, then the corresponding data pointer is returned as the argument. The CSELF type is most useful when just a single anchor to native C++ data is backing an object instance and the backing data is created in the object INIT method. Here's the Pointer example above reworked to use CSELF:

Example 8.37. API - CSELF

```
RexxMethod0(RexxObjectPtr, myclass_init)
{
    // construct a C++ object to associate with the object
    MyDataClass *myData = new MyDataClass();
    // create a Pointer to store this in the object
    RexxPointerObject data = context->NewPointer(myData);
    // assign this to the special CSELF variable
    context->SetObjectVariable("CSELF", data);
    // initialize the data below
    ...

    return NULLOBJECT;
}


RexxMethod1(RexxObjectPtr, myclass_dosomething, CSELF, cself)
{
    // We can just cast this to our data value
    MyDataClass *myData = (MyDataClass *)cself;
    // perform the operation below
    ...
}
```

```
RexxMethod1(RexxObjectPtr, myclass_uninit, CSELF, cself)
{
    // We can just cast this to our data value
    MyDataClass *myData = (MyDataClass *)cself;
    // delete the backing instance
    delete myData;
}
```

Using the CSELF argument type eliminates the need to directly access the Rexx variable used to anchor the value in every method except the INIT method. This produces generally smaller, more reliable code, since the runtime is managing the retrieval.

There are other advantages to using the CSELF convention. The example above is equivalent to the examples using Pointer and Buffer objects. If, however, you were to create a subclass of the Buffer example and try to access the value stored in MYDATA from a subclass method, you'll find that GetObjectVariable("MYDATA") will return NULLOBJECT. The GetObjectVariable() method retrieves variables from the current method's variable scope. Since the INIT method that set MYDATA originally and the subclass method that wishes to access the data are defined at different class scopes, GetObjectVariable() will access different variable pools and MYDATA will not be found. One solution would be to create a private attribute method in the base class:

```
::attribute mydata get private
```

The subclass method can then access the method using SendMessage() (*Section 8.16.144, "SendMessage0"*) to access the value.

```
RexxObjectPtr self = context->GetSelf()
RexxPointerObject = context->SendMessage0(self, "MYDATA");
```

The CSELF type handles this detail automatically. When used as an argument, all variable scopes of the object's class hiearchy are searched for a variable named CSELF. if one is located, it will be used for the value passed to the method. This allows all subclasses of a class using the CSELF convention to access the backing native data.

Frequently, one class instance might need access to the native information associated with another object instance. The other object instance might be of the same class or another class that is designed to interoperate with the current class. The ObjectToCSelf() (*Section 8.16.117, "ObjectToCSelf"*) allows the CSELF information for an object other than the current active object to be retrieved.

8.14. Rexx Exits 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

Direct exit handlers are specified when the interpreter instance is created, and reside as entry points within the application that creates the interpreter instance.

8.14.1. Writing Context Exit Handlers

The following is a sample exit handler declaration:

Example 8.38. API - Rexx_IO_Exit

```
int REXXENTRY Rexx_IO_exit(
    RexxExitContext *context,    // the exit context API vector
    int   exitNumber,            // code defining the exit function
    int   subfunction,           // code defining the exit subfunction
    PEXIT parmBlock);            // function-dependent control block
```

where:

context
    is the RexxExitContext vector that provides access to interpreter services for this exit handler.

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.

8.14.1.1. 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"). Other errors can be raised using the RaiseException() (*Section 8.16.134, "RaiseException"*) API provided by the exit context.

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".

8.14.1.2. Exit Parameters

Each exit subfunction has a different parameter list. All RXSTRING exit subfunction parameters are passed as null-terminated strings. The terminating null is not included in the length stored in the RXSTRING structures. The string values pointed to by the RXSTRING structs may 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 **RexxAllocateMemory(size)**. The Rexx interpreter will release the allocated storage after the exit handler returns.

8.14.1.3. Identifying Exit Handlers to Rexx

System exit handlers are specified using the DIRECT_EXITS option when the interpreter instance is created. The exits are specified using a RexxContextExit structure identifying which exits will be enabled.

8.14.2. Context Exit Definitions

The Rexx interpreter supports the following system exits:

RXFNC

    External function call exit.

    RXFNCCAL

        Call an external function. This exit is called at the beginning of the search for external functions, allowing external functions calls to be intercepted. The RXFNCCAL converts all function arguments to RXSTRING values and can only return RXSTRING values as a function result. For full object access, the RXOFNC exit is also provided.

RXOFNC

    Object oriented external function call exit.

    RXOFNCCAL

        Call an external function. This exit is called at the beginning of the search for external functions, allowing external functions calls to be intercepted. This is an extended version of the RXFNC exit that passes arguments as object references and allows object return values.

RXEXF

Scripting external function call exit.

RXEXFCAL

Call an external function. This exit is called at the end of the search for external functions if no suitable call target has been found. This allows applications to extend the external function search order. Like the RXOFNC exit, the RXEXF exit will pass function arguments and return values as Rexx objects.

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.

RXNOVAL

NOVALUE exit.

RXNOVALCALL

Process a variable NOVALUE condition.

RXVALUE

VALUE built-in function extension.

RXVALUECALL

Process a VALUE() built-in function call for an unknown named environment.

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

8.14.2.1. RXOFNC

Processes calls to external functions.

RXOFNCCAL

Processes calls to external functions.

- When called: At beginning of the search for an external routine 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:

Example 8.39. API - Rexx_IO_Exit parameter list

```
typedef  struct _RXOFNC_FLAGS {          /* fl */
    unsigned rxfferr  : 1;                /* Invalid call to routine.   */
    unsigned rxffnfnd : 1;                /* Function not found.        */
```

```
    unsigned rxffsub  : 1;                /* Called as a subroutine    */
}   RXOFNC_FLAGS ;

typedef  struct _RXOFNCCAL_PARM {      /* fnc */
    RXOFNC_FLAGS       rxfnc_flags ;    /* function flags              */
    CONSTRXSTRING      rxfnc_name;      // the called function name
    size_t             rxfnc_argc;      /* Number of args in list.     */
    RexxObjectPtr     *rxfnc_argv;      /* Pointer to argument list.   */
    RexxObjectPtr      rxfnc_retc;      /* Return value.               */
}   RXOFNCCAL_PARM;
```

The name of the external function is defined by the *rxfnc_name* CONSTRXSTRING (*Section 9.2, "RXSTRINGs"*) value. The arguments to the function are in *rxfnc_argv* array and *rxfnc_argc* gives the number of arguments. 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* RexxObjectPtr. 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.

8.14.2.2. RXEXF

Processes calls to external functions.

RXEXFCAL
    Processes calls to external functions.

- When called: At end of the search for an external routine or function when no suitable call target has been located.

- Default action: Raise error 43 ("Routine not found").

- 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:

Example 8.40. API - Rexx_IO_Exit parameter list

```
typedef  struct _RXEXF_FLAGS {         /* fl */
```

```
    unsigned rxfferr  : 1;                  /* Invalid call to routine.   */
    unsigned rxffnfnd : 1;                  /* Function not found.        */
    unsigned rxffsub  : 1;                  /* Called as a subroutine     */
}   RXEXF_FLAGS ;

typedef  struct _RXEXFCAL_PARM {       /* fnc */
    RXEXF_FLAGS       rxfnc_flags ;    /* function flags              */
    CONSTRXSTRING     rxfnc_name;      // the called function name
    size_t            rxfnc_argc;      /* Number of args in list.    */
    RexxObjectPtr    *rxfnc_argv;      /* Pointer to argument list.  */
    RexxObjectPtr     rxfnc_retc;      /* Return value.              */
}   RXEXFCAL_PARM;
```

The name of the external function is defined by the *rxfnc_name* CONSTRXSTRING value. The arguments to the function are in *rxfnc_argv* array and *rxfnc_argc* gives the number of arguments. 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* RexxObjectPtr. 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.

8.14.2.3. RXFNC

Processes calls to external functions.

RXFNCCAL
    Processes calls to external functions.

- When called: At beginning of the search for an external routine 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:

    Example 8.41. API - Rexx_IO_Exit 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 ;

      const char *      rxfnc_name;         /* Pointer to function name.   */
      unsigned short    rxfnc_namel;        /* Length of function name.    */
      const char *      rxfnc_que;          /* Current queue name.         */
      unsigned short    rxfnc_quel;         /* Length of queue name.       */
      unsigned short    rxfnc_argc;         /* Number of args in list.     */
      PCONSTRXSTRING    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.

The RXFNC translates all call arguments to string values and only allows a string value as a return value. To access call arguments as Rexx objects, use the RXOFNC exit.

8.14.2.4. RXCMD

Processes calls to subcommand 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:

Example 8.42. API - Rexx_IO_Exit 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;
   const char *       rxcmd_address;  /* Pointer to address name.    */
   unsigned short     rxcmd_addressl; /* Length of address name.     */
   const char *       rxcmd_dll;      /* dll name for command.       */
   unsigned short     rxcmd_dll_len;  /* Length of dll name.  0 ==>  */
                                      /* executable file.            */
   CONSTRXSTRING      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.

8.14.2.5. 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:

Example 8.43. API - Rexx_IO_Exit 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:

Example 8.44. API - Rexx_IO_Exit parameter list

```
typedef struct {
   struct {                             /* Operation flag             */
      unsigned rxfmlifo : 1;         /* Stack entry LIFO when TRUE, */
                                     /* FIFO when FALSE.           */
   } rxmsq_flags;
   CONSTRXSTRING     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:

Example 8.45. API - Rexx_IO_Exit parameter list

```
typedef struct {
   size_t           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:
>
>   > Example 8.46. API - Rexx_IO_Exit parameter list
>   >
>    > ```
>    > typedef struct {
>    >    RXSTRING     rxmsq_name;       /* RXSTRING containing        */
>    >                                  /* queue name.                */
>    > } RXMSQNAM_PARM;
>    > ```
>
>   *rxmsq_name* contains the new queue name.

8.14.2.6. 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:
>
>   > Example 8.47. API - Rexx_IO_Exit parameter list
>   >
>    > ```
>    > typedef struct {
>    >    CONSTRXSTRING     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:

> Example 8.48. API - Rexx_IO_Exit parameter list
>
> ```
> typedef struct {
>     CONSTRXSTRING       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:

> Example 8.49. API - Rexx_IO_Exit parameter list
>
> ```
> typedef struct {
>     RXSTRING        rxsiotrd_retc;    /* RXSTRING for input.        */
> } 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:

Example 8.50. API - Rexx_IO_Exit parameter list

```
typedef struct {
   RXSTRING          rxsiodtr_retc;    /* RXSTRING for input.        */
} RXSIODTR_PARM;
```

The input stream line is returned in the *rxsiodtr_retc* RXSTRING.

8.14.2.7. RXNOVAL

Processes NOVALUE variable conditions.

RXNOVALCALL
   Processes a Rexx NOVALUE condition.

- When called: Before the interpreter raises a NOVALUE condition. The exit is given the opportunity to provide a value to the unassigned variable.

- Default action: Raise a NOVALUE condition for an unassigned variable.

- Exit action: Return an initial value for an unassigned variable.

- Continuation: If the exit provides a value for the unassigned variable, that value is assigned to the indicated variable. The exit will not be called for the same variable on the next reference unless the variable is dropped. If a value is not returned, a NOVALUE condition will be raised. If SIGNAL ON NOVALUE is not enabled, the variable name will be returned as the value.

- Parameter list:

Example 8.51. API - Rexx_IO_Exit parameter list

```
typedef  struct _RXVARNOVALUE_PARM {   /* var */
   RexxStringObject  variable_name;    // the request variable name
   RexxObjectPtr     value;            // returned variable value
}  RXVARNOVALUE_PARM;
```

8.14.2.8. RXVALUE

Extends the environments available to the VALUE() built-in function.

RXVALUECALL
   Processes an extended call to the VALUE() built-in function.

- When called: When the VALUE() built-in function is called with an unknown environment name. The exit is given the opportunity to provide a value for the given environment selector.

- Default action: Raise a SYNTAX error for an unknown environment name.

- Exit action: Return a value for the given name/environment pair.

- Continuation: If the exit provides a value for the VALUE() call, that value is returned as a result. .

- Parameter list:

Example 8.52. API - Rexx_IO_Exit parameter list

```
typedef  struct _RXVALCALL_PARM {      /* val */
   RexxStringObject  selector;         // the environment selector name
   RexxStringObject  variable_name;    // the request variable name
   RexxObjectPtr     value;            // returned variable value
}  RXVALCALL_PARM;
```

If the newValue argument is specified on the VALUE() built-in function, that value is assigned to *value* on the call to the exit.

8.14.2.9. RXHLT

HALT condition processing.

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:

Example 8.53. API - Rexx_IO_Exit 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.

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.

8.14.2.10. 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 SetThreadTrace() (*Section 8.16.155, "SetThreadTrace"*) method 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:

Example 8.54. API - Rexx_IO_Exit 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.

8.14.2.11. RXINI

Initialization processing. This exit is called as the last step of Rexx program initialization.

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 SetContextVariable() (*Section 8.16.147, "SetContextVariable"*) API to initialize application-specific variables.

      - Use SetThreadTrace() (*Section 8.16.155, "SetThreadTrace"*) API to switch on the interactive Rexx debug mode.

   - Parameters: None.

8.14.2.12. RXTER

Termination processing.

The RXTER exit is called as the first step of Rexx program termination.

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 SetContextVariable() (*Section 8.16.147, "SetContextVariable"*) to retrieve the Rexx variable values.

   - Parameters: None.

8.15. Command Handler Interface

Applications can create custom command handlers that function like operating command shell environments. These named evironments can be invoked with the Rexx ADDRESS instuction and applications can create Rexx instances that direct commands to custom application command handlers by default.

Command handlers are registered with an interpreter instance when it is created. See *Section 8.1.2, "Interpreter Instance Options"* for how to register a handler with an interpreter instance.

The command handlers are registered as a function pointer to a handler routine. When a Rexx program issues a command to the named ADDRESS target, the handler is called with the evaluated

command string and the name of the address environment. The handler is responsible for executing the command and returning a return code value back to the Rexx program. Handlers are called using the following function signature:

```
RexxObjectPtr RexxEntry TestCommandHandler(RexxExitContext *context,
      RexxStringObject address, RexxStringObject command)
```

Arguments

context   A RexxExitContext (*Section 8.10, "Rexx Exit Context Interface"*) interface vector for the handler call. The RexxExitContext provides access to runtime services appropriate to a command handler. For example, the exit context can set or get Rexx variables, invoke methods on objects, and raise ERROR or FAILURE conditions.

address   A String object containing the target command environment name.

command   A String object containing the issued command string.

Returns

Any object that should be used as the command return code. This value will be assigned to the variable RC upon return. If NULLOBJECT is returned, a 0 is used as the return code. The return code value is traditionally a numeric value, but any value can be returned, including more complex object return values, if desired.

For normal commands, the command is processed and a return code is given back to the Rexx program. The interpreter recognizes two different abnormal return states for commands, ERROR and FAILURE. An ERROR condition indicates there was some sort of error return state involved with executing a command. These could be command syntax errors, semantic errors, etc. FAILURE conditions are more serious conditions. One traditional FAILURE condition is the unknown command error.

Command handlers raise ERROR and FAILURE conditions using the RaiseCondition() (*Section 8.16.133, "RaiseCondition"*) API provided by the RexxExitContext. For example:

Example 8.55. API - Command handler interface

```
// if this was an unknown command, give our generic unknown command return code
if (errorStatus == COMMAND_FAILURE) {
    // Note:  The return code needs to be included with the FAILURE condition
    context->RaiseCondition("FAILURE", command, NULLOBJECT, context->WholeNumber(-1));
    // just return null...the RC value is picked up from the condition.
    return NULLOBJECT;
}
else if (errorStatus == COMMAND_ERROR) {
    // Note:  The return code needs to be included with the ERROR condition
    context->RaiseCondition("ERROR", command, NULLOBJECT, context->WholeNumber(rc));
    // just return null...the RC value is picked up from the condition.
    return NULLOBJECT;
}
// return the RC value for the command, which need not be 0
return context->WholeNumber(rc);
```

8.16. Rexx Interface Methods Listing

This section describes each available method and its associated context.

8.16.1. Array

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxArrayObject arr;
 RexxObjectPtr obj1, obj2, obj3, obj4;

 >>--arr = context->Array(obj1);-----------------------------------------><

 >>--arr = context->Array(obj1, obj2);-----------------------------------><

 >>--arr = context->Array(obj1, obj2, obj3);-----------------------------><

 >>--arr = context->Array(obj1, obj2, obj3, obj4);-----------------------><
```

This method has four forms. It create a new one-dimension Array with the specified objects.

Arguments

obj1	The first object to be added.
obj2	The second object to be added.
obj3	The third object to be added.
obj4	The fourth object to be added.

Returns

The new Array object.

8.16.2. ArrayAppend

Context
Thread
Method
Function
Exit

Interpreter

```
RexxArrayObject arr;
RexxObjectPtr obj;
size_t n;

>>--n = context->ArrayAppend(arr, obj);---------------------------------><
```

Append an Object to the end of an Array.

Arguments

arr	The target Array object.
obj	The object to be appended.

Returns

The index of the appended object.

8.16.3. ArrayAppendString

Context
Thread
Method
Function
Exit
Interpreter

```
RexxArrayObject arr;
CSTRING str;
size_t n, len;

>>--n = context->ArrayAppendString(arr, str, len);----------------------><
```

Append an object to the end of an Array. The appended object is a String object created from a pointer and length.

Arguments

arr	The target Array object.
str	A pointer to the string data to be appended.
len	The length of the string value in characters.

Returns

The Array index of the appended object.

8.16.4. ArrayAt

Context
Thread
Method
Function
Exit
Interpreter

```
RexxArrayObject arr;
RexxObjectPtr obj;
size_t idx;

>>--obj = context->ArrayAt(arr, idx);------------------------------------><
```

Retrieve an object from a specified Array index.

Arguments

arr	The source Array object.
idx	The index of the required object. This argument is 1-based.

Returns

The object at the specified index. Returns NULLOBJECT if there is no value at the specified index.

8.16.5. ArrayDimension

Context
Thread
Method
Function
Exit
Interpreter

```
RexxArrayObject arr;
size_t sz;

>>--sz = context->ArrayDimension(arr);------------------------------------><
```

Returns number of dimensions of an Array.

Arguments

arr   The target Array object.

Returns

The number of Array dimensions.

8.16.6. ArrayItems

Context
Thread
Method
Function
Exit
Interpreter

```
RexxArrayObject arr;
size_t sz;

>>--sz = context->ArrayItems(arr);---------------------------------------><
```

Returns number of elements in an Array.

Arguments

arr   The source Array object.

Returns

The number of Array elements.

8.16.7. ArrayOfFour

Context
Thread
Method
Function
Exit
Interpreter

```
RexxArrayObject arr;
RexxObjectPtr obj1, obj2, obj3, obj4;

>>--arr = context->ArrayOfFour(obj1, obj2, obj3, obj4);------------------><
```

Create a new one-dimension Array with the specified objects.

Arguments

obj1	The first object to be added.
obj2	The second object to be added.
obj3	The third object to be added.
obj4	The fourth object to be added.

Returns

The new Array object.

8.16.8. ArrayOfThree

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxArrayObject arr;
 RexxObjectPtr obj1, obj2, obj3;

 >>--arr = context->ArrayOfThree(obj1, obj2, obj3);---------------------><
```

Create a new one-dimension Array with the specified objects.

Arguments

obj1	The first object to be added.
obj2	The second object to be added.
obj3	The third object to be added.

Returns

The new Array object.

8.16.9. ArrayOfTwo

Context
Thread
Method
Function

Exit
Interpreter

```
 RexxArrayObject arr;
 RexxObjectPtr obj1, obj2;

>>--arr = context->ArrayOfTwo(obj1, obj2);------------------------------><
```

Create a new one-dimension Array with the specified objects..

Arguments

obj1   The first object to be added.

obj2   The second object to be added.

Returns

The new Array object.

8.16.10. ArrayOfOne

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxArrayObject arr;
 RexxObjectPtr obj;

>>--arr = context->ArrayOfOne(obj);-------------------------------------><
```

Create a new one-dimension Array with the specified object.

Arguments

obj   The object to be added.

Returns

The new Array object.

8.16.11. ArrayPut

Context
Thread
Method
Function
Exit
Interpreter

```
RexxArrayObject arr;
RexxObjectPtr obj;
size_t idx;

>>--context->ArrayPut(arr, obj, idx);-------------------------------------><
```

Replace/add an Object to an Array.

Arguments

arr	The target Array object.
obj	The object to be added.
idx	The index into the Array object. This argument is 1-based.

Returns

Void.

8.16.12. ArraySize

Context
Thread
Method
Function
Exit
Interpreter

```
RexxArrayObject arr;
size_t sz;

>>--sz = context->ArraySize(arr);----------------------------------------><
```

Returns the size of an Array.

Arguments

arr	The source Array object.

Returns

The Array size.

8.16.13. AttachThread

Context
Thread
Method
Function
Exit
Interpreter

```
RexxThreadContext *tc

>>--success = context->AttachThread(&tc);------------------------------><
```

Attaches the current thread to the Rexx interpreter instance *context* pointer.

Arguments

tc   Pointer to a RexxThreadContext pointer used to return a RexxThreadContext for the attached thread.

Returns

Boolean value. 1 = success, 0 = failure. If the call was successful, a RexxThreadContext object valid for the current context is returned via the *tc* argument.

8.16.14. BufferData

Context
Thread
Method
Function
Exit
Interpreter

```
RexxBufferObject obj;
POINTER str;

>>--str = context->BufferData(obj);--------------------------------------------><
```

Returns a pointer to a Buffer object's data area.

Arguments

obj   The source Buffer object.

Returns

The C pointer to the Buffer object's data area.

8.16.15. BufferLength

Context
Thread
Method
Function
Exit
Interpreter

```
RexxBufferObject obj;
size_t sz;

>>--sz = context->BufferLength(obj);-------------------------------------><
```

Return the length of a Buffer object's data area.

Arguments

obj   The source Buffer object.

Returns

The length of the Buffer object's data area.

8.16.16. BufferStringData

Context
Thread
Method
Function
Exit
Interpreter

```
RexxBufferStringObject obj;
POINTER str;

>>--str = context->BufferStringData(obj);--------------------------------------><
```

Returns a pointer to a RexxBufferString object's data area.

Arguments

obj   The source object.

Returns

The C pointer to the RexxBufferString's data area. This is a writable data area, but the RexxBufferString must be finalized using FinishBufferString() (*Section 8.16.40, "FinishBufferString"*) before it can be used in any other context.

8.16.17. BufferStringLength

Context
Thread
Method
Function
Exit
Interpreter

```
RexxBufferStringObject obj;
size_t sz;

>>--sz = context->BufferStringLength(obj);------------------------------><
```

Return the length of a RexxBufferStringObject instance.

Arguments

obj   The source RexxBufferStringObject.

Returns

The length of the RexxBufferStringObject.

8.16.18. CallProgram

Context
Thread
Method

Function
Exit
Interpreter

```
CSTRING name;
RexxObjectPtr ret;
RexxArrayObject arr;

>>--ret = context->CallProgram(name, arr);-------------------------><
```

Returns the result object of the routine.

Arguments

name   The ASCII-Z path/name of the Rexx program to call.

arr   An Array of object program arguments.

Returns

Any result object returned by the program. NULLOBJECT is returned if the program does not return a value. Any errors involved with calling the program will will return a NULLOBJECT result. The CheckCondition() (*Section 8.16.20, "CheckCondition"*). can be used to check if any errors occurred during the call.

8.16.19. CallRoutine

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj, ret;
RexxArrayObject arr;

>>--ret = context->CallRoutine(obj, arr);-----------------------------><
```

Returns the result object of the routine.

Arguments

obj   The routine object to call.

arr   An Array of routine argument objects.

Returns

Any result object returned by the Routine. NULLOBJECT is returned if the program does not return a value. Any errors involved with calling the program will will return a NULLOBJECT result. The CheckCondition() (*Section 8.16.20, "CheckCondition"*). can be used to check if any errors occurred during the call.

8.16.20. CheckCondition

Context
Thread
Method
Function
Exit
Interpreter

```
logical_t flag;

>>--flag = context->CheckCondition();------------------------------------><
```

Checks to see if any conditions have resulted from a call to a Rexx API. .

Arguments

None.

Returns

1 = if a condition has been raised, 0 = no condition raised.

8.16.21. ClearCondition

Context
Thread
Method
Function
Exit
Interpreter

```
>>--context->ClearCondition();------------------------------------------------><
```

Clears any pending condition status.

Arguments

None.

Returns

Void.

8.16.22. CString

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
RexxStringObject ostr;
CSTRING str;

>>--str = context->CString(obj);-----------------------------------------><

>>--ostr = context->CString(str);-----------------------------------------><
```

There are two forms of this method. The first converts an Object into a C ASCII-Z string. The second converts C ASCII-Z string into a String object.

Arguments

obj   The source object for the conversion.

str   The source C ASCII-Z string for the conversion.

Returns

For the first method form, a CSTRING representation of the object is returned. For the second form, a String object is created from the ASCII-Z string data..

8.16.23. DecodeConditionInfo

Context
Thread
Method
Function
Exit
Interpreter

```
RexxDirectoryObject dirj;
RexxCondition cond;

>>--context->DecodeConditionInfo(dir, &cond);--------------------------><
```

Decodes the condition information into a RexxCondition structure.

Arguments

dir	The source Directory object containing the condition information.
cond	A pointer to the RexxCondition structure.

Returns

Void. The **cond** structure is updated with information from *dir*.

8.16.24. DetachThread

Context
Thread
Method
Function
Exit
Interpreter

```
>>--context->DetachThread()-------------;------------------------------><
```

Detaches the thread represented by the RexxThreadContext object from it's interpreter instance. Once DetachThread() is called, the RexxThreadContext object issuing the call is no longer a valid, active interface.

Arguments

None

Returns

Void.

8.16.25. DirectoryAt

Context
Thread
Method

Function
Exit
Interpreter

```
RexxDirectoryObject dirobj;
RexxObjectPtr obj;
CSTRING str;

>>--obj = context->DirectoryAt(dirobj, str);---------------------------><
```

Return the object at the specified index.

Arguments

dirobj   The source Directory object.

str   The index into the Directory object.

Returns

The object at the specified index. Returns NULLOBJECT if the given index does not exist.

8.16.26. DirectoryPut

Context
Thread
Method
Function
Exit
Interpreter

```
RexxDirectoryObject dirobj;
RexxObjectPtr item;
CSTRING index;

>>--context->DirectoryPut(dirobj, item, index);--------------------------><
```

Replace/add an Object at the specified Directory index.

Arguments

dirobj   The source Directory object.

item   The object instance to be stored at the index.

index   The ASCII-Z string index into the Directory object.

Returns

Void.

8.16.27. DirectoryRemove

Context
Thread
Method
Function
Exit
Interpreter

```
RexxDirectoryObject dirobj;
RexxObjectPtr obj;
CSTRING str;

>>--obj = context->DirectoryRemove(dirobj, str);------------------------><
```

Removes and returns the object at the specified Directory index.

Arguments

dirobj	The source Directory object.
str	The ASCII-Z index into the Directory object.

Returns

The object removed at the specified index. Returns NULLOBJECT if the index did not exist in the target Directory.

8.16.28. DisplayCondition

Context
Thread
Method
Function
Exit
Interpreter

```
wholenumber_t rc;

>>--rc = context->DisplayCondition();------------------------------------><
```

If any syntax conditions are currently pending in the Rexx context, then error information will be output to the current .error stream.

Arguments

None.

Returns

If there is syntax information to display, the return code will be the major error number for the syntax error. Returns 0 if there is no current syntax condition.

8.16.29. Double

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 double n;
 logical_t flag;

 >>--obj = context->Double(n);-------------------------------------><

 >>--flag = context->Double(obj, &n);--------------------------------><
```

There are two forms of this method. The first form converts C double value to an Object. The second form converts an Object to a C double value.

Arguments

n	For the first method form, the double value to be converted. For the second method form, the target of the conversion.
obj	The object to be converted..

Returns

For the first method form, returns an Object version of the double value. For the second method form, 0 - success, 1 = failure. If successful, the converted value is placed in **n**.

8.16.30. DoubleToObject

Context
Thread

Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 double n;

 >>--obj = context->DoubleToObject(n);------------------------------------><
```

Converts C double value to an Object.

Arguments

n   The double value to be converted.

Returns

An Object representation of the double value.

8.16.31. DoubleToObjectWithPrecision

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 size_t p;
 double n;

 >>--obj = context->DoubleToObject(n, p);--------------------------------><
```

Converts C double value to an Object with a specific precision.

Arguments

n   The double value to be converted.
p   The precision to be used for the conversion.

Returns

An Object representation of the double value.

8.16.32. DropContextVariable

Context
Thread
Method
Function
Exit
Interpreter

```
CSTRING name;

>>--context->DropContextVariable(name);-------------------------------><
```

Drops a Rexx variable in the current routine's caller variable context.

Arguments

name   The name of the Rexx variable.

Returns

Void.

8.16.33. DropObjectVariable

Context
Thread
Method
Function
Exit
Interpreter

```
CSTRING str;

>>--context->DropObjectVariable(str);------------------------------------><
```

Drops an instance variable in the current method's scope.

Arguments

str   The name of the object variable.

Returns

Void.

8.16.34. DropStemArrayElement

Context
Thread
Method
Function
Exit
Interpreter

```
RexxStemObject sobj;
size_t n;

>>--context->DropStemArrayElement(sobj, n);----------------------------><
```

Drops an element of the Stem object.

Arguments

sobj	The target Stem object.
n	The Stem object element number.

Returns

Void.

8.16.35. DropStemElement

Context
Thread
Method
Function
Exit
Interpreter

```
RexxStemObject sobj;
CSTRING name;

>>--context->DropStemElement(sobj, name);------------------------------><
```

Drops an element of the Stem object.

Arguments

sobj	The target Stem object.
name	The Stem object element name.

Returns

Void.

8.16.36. False

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;

>>--obj = context->False();---------------------------------------><
```

This method returns the Rexx .false (0) object.

Arguments

None.

Returns

The Rexx .false object.

8.16.37. FindClass

Context
Thread
Method
Function
Exit
Interpreter

```
RexxClassObject class;
CSTRING name;
```

```
>>--class = context->FindClass(name);--------------------------------><
```

Locates a Class object in the current thread context.

Arguments

name   An ASCII-Z string containing the name of the class.

Returns

The located Class object. Returns NULLOBJECT if the class is not found.

8.16.38. FindContextClass

Context
Thread
Method
Function
Exit
Interpreter

```
 CSTRING name;
 RexxClassObject obj;

>>--obj = context->FindContextClass(name);------------------------------><
```

Locate a Class object in the current Method or Routine Package context.

Arguments

name   The class name to be located.

Returns

The located Class object. Returns NULLOBJECT if the class is not found.

8.16.39. FindPackageClass

Context
Thread
Method
Function
Exit
Interpreter

```
RexxPackageObject pkg;
RexxClassObject class;
CSTRING name;

>>--class = context->FindPackageClass(pkg, name);--------------------><
```

Locate a class object in a given Package object's context.

Arguments

pkg	The Package object used to resolve the class.
name	An ASCII-Z string containing the name of the class.

Returns

The located Class object. Returns NULLOBJECT if the class is not found.

8.16.40. FinishBufferString

Context
Thread
Method
Function
Exit
Interpreter

```
RexxBufferStringObject obj;
RexxStringObject strobj;
size_t len;

>>--str = context->FinishBufferString(obj, len);--------------------------><
```

Converts a RexxBufferStringObject into a completed, immutable String object of the given length and returns a reference to the completed String object.

Arguments

obj	The working RexxBufferStringObject.
len	The final length of the constructed string.

Returns

The finalized Rexx string object.

8.16.41. ForwardMessage

Context
Thread
Method
Function
Exit
Interpreter

```
 CSTRING str;
 RexxObjectPtr obj, ret;
 RexxClassObject sobj;
 RexxArrayObject arr;

 >>--ret = context->ForwardMessage(obj, str, cobj, arr);------------------><
```

Forwards a message to a different object or method. This is equivalent to using a FORWARD CONTINUE instruction from Rexx code.

Arguments

obj	The object to receive the message. If NULL, the object that is the target of the current method call is used.
str	The message name to use. If NULL, then the name of the current method is used.
cobj	The class scope used to locate the method. If NULL, this will be an unscoped method call.
arr	The Array of message arguments. If NULL, the same arguments that were used on the current method invocation will be used.

Returns

The invoked message result. NULLOBJECT will be returned if the target method does not return a result.

8.16.42. GetAllContextVariables

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxDirectoryObject obj;

 >>--obj = context->GetAllContextVariables();--------------------------><
```

Returns all the Rexx variables in the current routine's caller's context as a Directory. Only simple variables and stem variables are included in the Directory. Stem variable entries will have a Stem object as the value. Compound variables may be accessed via the Stem object values.

Arguments

None.

Returns

A RexxDirectoryObject with the variable names and values.

8.16.43. GetAllStemElements

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxStemObject sobj;
 RexxDirectoryObject obj;

 >>--obj = context->GetAllStemElements(sobj);--------------------------->< 
```

Returns all elements of a Stem object as a Directory object. Each assigned Stem tail element will be an entry in the Directory.

Arguments

sobj   The source Stem object.

Returns

The Directory object containing the Stem variable values.

8.16.44. GetApplicationData

Context
Thread
Method
Function
Exit

Interpreter

```
 >>--ptr = context->GetApplicationData();------------------------------><
```

Returns the application data pointer that was set via the APPLICATION_DATA option when the intepreter instance was created.

Arguments

None.

Returns

The application instance data set when the interpreter instance was created.

8.16.45. GetArgument

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 size_t n;

 >>--obj = context->GetArgument(n);------------------------------------><
```

Returns the specified argument to the method or routine. This is equivalent to calling Arg(n) from within Rexx code.

Arguments

n The argument number (1-based).

Returns

The object corresponding to the given argument position. Returns NULLOBJECT if the argument was not specified.

8.16.46. GetArguments

Context
Thread

Method
Function
Exit
Interpreter

```
RexxArrayObject arr;

>>--arr = context->GetArguments();-------------------------------------><
```

Returns an Array object of the arguments to the method or routine. This is the same argument Array returned by the ARGLIST argument type.

Arguments

None.

Returns

The Array object containing the method or routine arguments.

8.16.47. GetCallerContext

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;

>>--obj = context->GetCallerContext();-------------------------------><
```

Get the RexxContext object corresponding to the routine or exit's calling context.

Arguments

None.

Returns

The current exit or routine caller's RexxContext object.

8.16.48. GetConditionInfo

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxDirectoryObject dir;

 >>--dir = context->GetConditionInfo();-------------------------------><
```

Returns a Directory object containing the condition information. This is equivalent to calling Condition('O') from within Rexx code.

Arguments

None.

Returns

The RexxDirectoryObject containing the condition information. If there are no pending conditions, NULLOBJECT is returned.

8.16.49. GetContextDigits

Context
Thread
Method
Function
Exit
Interpreter

```
 stringsize_t sz;

 >>--sz = context->GetContextDigits();-------------------------------><
```

Get the routine caller's current NUMERIC DIGITS setting.

Arguments

None.

Returns

The current NUMERIC DIGITS setting.

8.16.50. GetContextForm

Context
Thread
Method
Function
Exit
Interpreter

```
stringsize_t sz;

>>--sz = context->GetContextForm();----------------------------------><
```

Get the routine caller's current NUMERIC FORM setting.

Arguments

None.

Returns

The current NUMERIC FORM setting.

8.16.51. GetContextFuzz

Context
Thread
Method
Function
Exit
Interpreter

```
stringsize_t sz;

>>--sz = context->GetContextFuzz();----------------------------------><
```

Get the routine caller's current NUMERIC FUZZ setting.

Arguments

None.

Returns

The current NUMERIC FUZZ setting.

8.16.52. GetContextVariable

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
CSTRING name;

>>--obj = context->GetContextVariable(name);--------------------------><
```

Gets the value of a Rexx variable in the routine or exit caller's variable context. Only simple variables and stem variables can be retrieved with GetContextVariable(). The value returned for a stem variable will be the corresponding Stem object. Compound variable values can be retrieved from the corresponding Stem values.

Arguments

name   The name of the Rexx variable.

Returns

The value of the named variable. Returns NULLOBJECT if the variable has not been assigned a value.

8.16.53. GetGlobalEnvironment

Context
Thread
Method
Function
Exit
Interpreter

```
RexxDirectoryObject dir;

>>--dir = context->GetGlobalEnvironment();--------------------------><
```

Returns a reference to the .environment Directory.

Arguments

None.

Returns

A RexxDirectoryObject pointer to the .environment Directory.

8.16.54. GetLocalEnvironment

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxDirectoryObject dir;

 >>--dir = context->GetLocalEnvironment();---------------------------->< 
```

Returns a reference to the interpreter instance .local Directory.

Arguments

None.

Returns

A RexxDirectoryObject pointer to the .local Directory.

8.16.55. GetMessageName

Context
Thread
Method
Function
Exit
Interpreter

```
 CSTRING str;
```

```
>>--str = context->GetMessageName(obj);-------------------------------><
```

Returns the message name used to invoke the current method.

Arguments

None.

Returns

The current method message name.

8.16.56. GetMethod

Context
Thread
Method
Function
Exit
Interpreter

```
RexxMethodObject obj;

>>--obj = context->GetMethod();---------------------------------------><
```

Returns the Method object for the currently executing method.

Arguments

None.

Returns

The current Method object.

8.16.57. GetMethodPackage

Context
Thread
Method
Function
Exit
Interpreter

```
RexxMethodObject obj;
RexxPackageObject pkg;

>>--pkg = context->GetMethodPackage(obj);--------------------------><
```

Returns the Package object associated with the specified Method instance.

Arguments

obj   The source Method object..

Returns

The Method's defining Package object.

8.16.58. GetObjectVariable

Context
Thread
Method
Function
Exit
Interpreter

```
CSTRING str;
RexxObjectPtr obj;

>>--obj = context->GetObjectVariable(str);------------------------------><
```

Retrieves a Rexx instance variable value from the current object's method scope context. Only simple variables and stem variables can be retrieved with GetObjectVariable(). The value returned for a stem variable will be the corresponding Stem object. Compound variable values can be retrieved from the corresponding Stem values.

Arguments

str   The name of the object variable.

Returns

The object assigned to the named object variable. Returns NULLOBJECT if the variable has not been assigned a value.

8.16.59. GetPackageClasses

Context

| Thread |
| Method |
| Function |
| Exit |
| Interpreter |

```
 RexxDirectoryObject dir;
 RexxPackageObject pkg;

 >>--dir = context->GetPackageClasses(pkg);---------------------------><
```

Returns a Directory object containing the Package public and private classes, indexed by class name.

Arguments

obj   The package object to query.

Returns

A Directory object containing the package classes.

8.16.60. GetPackageMethods

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxDirectoryObject dir;
 RexxPackageObject pkg;

 >>--dir = context->GetPackageMethods(pkg);---------------------------><
```

Returns a Directory object containing the Package unattached methods, indexed by Method name. This is equivalent to using the .methods environment symbol from Rexx code.

Arguments

obj   The package routine object to query.

Returns

A Directory object containing the Package's unattached methods.

8.16.61. GetPackagePublicClasses

Context
Thread
Method
Function
Exit
Interpreter

```
RexxDirectoryObject dir;
RexxPackageObject pkg;

>>--dir = context->GetPackagePublicClasses(pkg);---------------------><
```

Returns a Directory object containing the Package public classes, indexed by class name.

Arguments

obj   The package object to query.

Returns

A Directory object containing the public classes.

8.16.62. GetPackagePublicRoutines

Context
Thread
Method
Function
Exit
Interpreter

```
RexxDirectoryObject dir;
RexxPackageObject pkg;

>>--dir = context->GetPackagePublicRoutines(pkg);--------------------><
```

Returns a Directory object containing the Package public routines, indexed by routine name.

Arguments

obj   The package object to query.

Returns

A Directory object containing the public routines.

8.16.63. GetPackageRoutines

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxDirectoryObject dir;
 RexxPackageObject pkg;

 >>--dir = context->GetPackageRoutines(pkg);--------------------------><
```

Returns a Directory object containing the Package public and private routines, indexed by routine name.

Arguments

  obj   The package routine object to query.

Returns

A Directory object containing the routines.

8.16.64. GetRoutine

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxRoutineObject obj;

 >>--obj = context->GetRoutine();-------------------------------------><
```

Returns current Routine object.

Arguments

None

Returns

The current Routine object.

8.16.65. GetRoutineName

Context
Thread
Method
Function
Exit
Interpreter

```
CSTRING name;

>>--name = context->GetRoutineName();--------------------------------><
```

Returns the name of the current routine.

Arguments

None

Returns

A pointer ASCII-Z routine name.

8.16.66. GetRoutinePackage

Context
Thread
Method
Function
Exit
Interpreter

```
RexxRoutineObject obj;
RexxPackageObject pkg;

>>--pkg = context->GetRoutinePackage(obj);---------------------------><
```
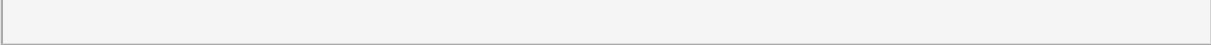
Returns Routine object's associated Package object.

Arguments

obj   The routine object to query.

Returns

The Package object instance.

8.16.67. GetScope

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;

>>--obj = context->GetScope();---------------------------------------><
```

Return the current active method's scope.

Arguments

None.

Returns

The current Method's scope.

8.16.68. GetSelf

Context
Thread
Method
Function
Exit
Interpreter
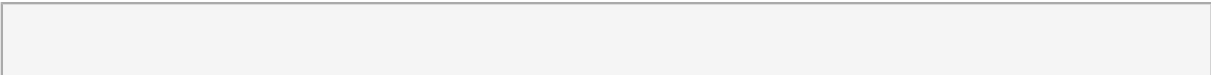
```
   RexxObjectPtr obj;

>>--obj = context->GetSelf();----------------------------------------><
```

Returns the Object that is the current method's message target. This is equivalent to the SELF variable in a Rexx method. The same value can be accessed as a method argument using the OSELF type.

Arguments

None.

Returns

The current SELF object.

8.16.69. GetStemArrayElement

Context
Thread
Method
Function
Exit
Interpreter

```
   RexxStemObject sobj;
   RexxObjectPtr obj;
   size_t n;

>>--obj = context->GetStemArrayElement(sobj, n);------------------------><
```

Retrieves an element of a Stem object using a numeric index.

Arguments

sobj	The source Stem object.
n	The Stem object element number. The numeric index is translated into the corresponding String tail.

Returns

The Object stored at the target index or NULLOBJECT if the target index has not been assigned a value.

8.16.70. GetStemElement

Context

| Thread |
| Method |
| Function |
| Exit |
| Interpreter |

```
RexxStemObject sobj;
RexxObjectPtr obj;
CSTRING name;

>>--obj = context->GetStemElement(sobj, name);-------------------------><
```

Retrieves an element of a Stem object.

Arguments

sobj   The source Stem object.

name   The Stem object element name. This is a fully resolved tail name, taken as a
                constant. No variable substitution is performed on the tail.

Returns

The the object at the target index or NULLOBJECT if the target index has not been assigned a value.

8.16.71. GetStemValue

Context
Thread
Method
Function
Exit
Interpreter

```
RexxStemObject sobj;
RexxObjectPtr obj;
CSTRING name;

>>--obj = context->GetStemValue(sobj);----------------------------------><
```

Retrieves the base name value of a Stem object.

Arguments

sobj   The source Stem object.

Returns

The Stem object's default base value.

8.16.72. GetSuper

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;

>>--obj = context->GetSuper();----------------------------------------><
```

Returns the current method's super class scope. This is equivalent to the SUPER variable used from Rexx code. This value can also be obtained via the SUPER method argument type.

Arguments

None.

Returns

The current method's SUPER scope.

8.16.73. Halt

Context
Thread
Method
Function
Exit
Interpreter

```
>>--context->Halt();-------------------------------------------------------><
```

Raise a HALT condition on all threads associated with the interpreter instance.

Arguments

None.

Returns

Void.

8.16.74. HaltThread

Context
Thread
Method
Function
Exit
Interpreter

```
>>--context->HaltThread();----------------------------------------><
```

Raises a HALT condition on the thread corresponding to the current *context* pointer.

Arguments

None

Returns

Void.

8.16.75. HasMethod

Context
Thread
Method
Function
Exit
Interpreter

```
logical_t flag;
RexxObjectPtr obj;
CSTRING name;

>>--flag = context->HasMethod(obj, name);-------------------------------><
```

Tests if an object supports the specified method name.

Arguments

obj	The target object.
name	An ASCII-Z method name.

Returns

1 = the method exists, 0 = the method does not exist.

8.16.76. InvalidRoutine

Context
Thread
Method
Function
Exit
Interpreter

```
RexxDirectoryObject obj;

>>--context->InvalidRoutine();---------------------------------------><
```

Raises the standard Error 40, "Invalid call to routine" synatx error for the current routine. This error will be raised by the Rexx runtime once the routine returns.

Arguments

None.

Returns

Void.

8.16.77. Int32

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;
int32_t n;
```

```
>>--obj = context->Int32(n);-------------------------------------------><

>>--logical_t = context->Int32(obj, &n);-------------------------------><
```

There a two forms of this method. The first form converts a C 32-bit integer *n* to an Object. The second form converts an Object to a C 32-bit integer, returning it in *n*.

Arguments

n	For the first form, the value to be converted. For the second form, the converted result.
obj	The object to be converted.

Returns

For the first form, n Object representation of the integer value. For the second form, returns 1 = success, 0 = failure. If successful, the converted value is placed in *n*.

8.16.78. Int32ToObject

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
int32_t n;

>>--obj = context->Int32ToObject(n);-----------------------------------><
```

Convert a C 32-bit integer *n* to an Object.

Arguments

n	The integer to be converted.

Returns

An Object representation of the integer value.

8.16.79. Int64

Context

Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;
int64_t n;

>>--obj = context->Int64(n);---------------------------------------------><

>>--logical_t = context->Int64(obj, &n);---------------------------------><
```

There a two forms of this method. The first form converts a C 64-bit integer *n* to an Object. The second form converts an Object to a C 64-bit integer and returns in *n*.

Arguments

n	For the first form, the integer to be converted. For the second form, the converted integer.
obj	The object to be converted.

Returns

For the first form, an Object representation of the integer value. For the second form, returns 1 = success, 0 = failure. If successful, the converted value is placed in *n*.

8.16.80. Int64ToObject

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
int64_t n;

>>--obj = context->Int64ToObject(n);-------------------------------------><
```

Convert the C 64-bit integer *n* to an Object.

Arguments

| *n* | The integer to be converted. |

Returns

An Object representing the integer value.

8.16.81. InterpreterVersion

Context
Thread
Method
Function
Exit
Interpreter

```
size_t version;

>>--version = context->InterpreterVersion();---------------------------><
```

Returns the version of the interpreter. The returned version is encoded in the 3 least significant bytes of the returned value, using 1 byte each for the interpreter version, release, and revision values. For example, on a 32-bit platform, this value would be 0x00040000 for version 4.0.0. The oorexxapi.h header file will have a define matching these values using the naming convention REXX_INTERPRETER_4_0_0 and the macro REXX_CURRENT_INTERPRETER_VERSION will give the interpreter version used to compile your code.

Arguments

None.

Returns

The interpreter version number.

8.16.82. Intptr

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;
intptr_t n;

>>--obj = context->Intptr(&n);------------------------------------------><

>>--flag = context->Intptr(obj, &n);------------------------------------><
```

There are two forms of this method. The first form converts the C signed integer *n* to an Object. The second form converts an Object to a C signed integer and returns it in *n*.

Arguments

n	For the first form, the value to be converted. For the second form, the conversion result.
obj	The object to be converted.

Returns

For the first form, an Object version of the integer value. For the second form, returns 1 = success, 0 = failure. If successful, the converted value is placed in *n*.

8.16.83. IntptrToObject

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
intptr_t n;

>>--obj = context->IntptrToObject(&n);----------------------------------><
```

Convert the C signed integer *n* to an Object.

Arguments

n	The signed integer to be converted.

Returns

An Object representing the integer value.

8.16.84. IsArray

Context	
Thread	
Method	
Function	
Exit	
Interpreter	

```
RexxObjectPtr obj;
logical_t flag;

>>--flag = context->IsArray(obj);---------------------------------------><
```

Tests if an Object is an Array. A true result indicates the RexxObjectPtr value may be safely cast to a RexxArrayObject.

Arguments

  obj   The object to be tested.

Returns

1 = is an Array object, 0 = not an Array object.

8.16.85. IsBuffer

Context	
Thread	
Method	
Function	
Exit	
Interpreter	

```
RexxObjectPtr obj;
logical_t flag;

>>--flag = context->IsBuffer(obj);---------------------------------------><
```

Tests if an Object is a Buffer object. A true result indicates the RexxObjectPtr value may be safely cast to a RexxBufferObject.

Arguments

  obj   The object to be tested.

Returns

1 = is a Buffer object, 0 = not a Buffer object.

8.16.86. IsDirectory

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;

>>--flag = context->IsDirectory(obj);------------------------------------>< 
```

Tests if an Object is a Directory object. A true result indicates the RexxObjectPtr value may be safely cast to a RexxDirectoryObject.

Arguments

 obj   The object to be tested.

Returns

1 = is a Directory object, 0 = not a Directory object.

8.16.87. IsInstanceOf

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
RexxClassObj class;
logical_t flag;

>>--flag = context->IsInstanceOf(obj, class);-------------------------->< 
```

Tests if an Object is an instance of the specified class.

Arguments

obj	The Object to be tested.
class	The Class object for the instance test.

Returns

1 = is an instance, 0 = not an instance.

8.16.88. IsMethod

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;

>>--flag = context->IsMethod(obj);---------------------------------------><
```

Tests if an Object is a Method object. A true result indicates the RexxObjectPtr value may be safely cast to a RexxMethodObject.

Arguments

obj	The object to be tested.

Returns

1 = is a Method object, 0 = not a Method object.

8.16.89. IsMutableBuffer

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;

>>--flag = context->IsMutableBuffer(obj);------------------------------><
```

Tests if an Object is a MutableBuffer object. A true result indicates the RexxObjectPtr value may be safely cast to a RexxMutableBufferObject.

Arguments

obj   The object to be tested.

Returns

1 = is a MutableBuffer object, 0 = not a MutableBuffer object.

8.16.90. IsOfType

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
CSTRING class;
logical_t flag;

>>--flag = context->IsOfType(obj, class);------------------------------><
```

Tests an object to see if it is an instance of the named class. This method combines the operations of the FindClass() and IsInstanceOf() methods in a single call.

Arguments

obj   The object to be tested.

class   An ASCII-Z string containing the name of the Rexx class. The named class will be located in the current context and used in an IsInstanceOf() test.

Returns

1 = is an instance, 0 = not an instance or the named class cannot be located.

8.16.91. IsPointer

Context

Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;

>>--flag = context->IsPointer(obj);------------------------------------><
```

Tests if an Object is a Pointer object. A true result indicates the RexxObjectPtr value may be safely cast to a RexxPointerObject.

Arguments

obj   The object to be tested.

Returns

1 = is a Pointer object, 0 = not a Pointer object.

8.16.92. IsRoutine

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;

>>--flag = context->IsRoutine(obj);------------------------------------><
```

Tests if an Object a Routine object. A true result indicates the RexxObjectPtr value may be safely cast to a RexxPointerObject.

Arguments

obj   The object to be tested.

Returns

1 = is a Routine object, 0 = not a Routine object.

8.16.93. IsStem

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;

>>--flag = context->IsStem(obj);---------------------------------------><
```

Tests if an Object is a Stem object. A true result indicates the RexxObjectPtr value may be safely cast to a RexxStemObject.

Arguments

obj   The object to be tested.

Returns

1 = is a Stem object, 0 = not a Stem object.

8.16.94. IsString

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;

>>--flag = context->IsString(obj);---------------------------------------><
```

Tests if an Object is a String object. A true result indicates the RexxObjectPtr value may be safely cast to a RexxStringObject.

Arguments

obj   The object to be tested.

Returns

1 = is a String object, 0 = not a String object.

8.16.95. LanguageLevel

Context
Thread
Method
Function
Exit
Interpreter

```
 size_t langlevel;

 >>--langlevel = context->LanguageLevel();-------------------------------><
```

Returns the language level of the interpreter. The returned language level is encoded in the 2 least significant bytes of the returned value, using 1 byte each for the interpreter version, release, and revision values. For example, on a 32-bit platform, this value would be 0x00000603 for language level 6.03. The oorexxapi.h header file will have a define matching these values using a the naming convention REXX_LANGUAGE_6_03 and the macro REXX_CURRENT_LANGUAGE_LEVEL will give the interpreter version used to compile your code.

Arguments

None.

Returns

The interpreter language level number.

8.16.96. LoadLibrary

Context
Thread
Method
Function
Exit
Interpreter

```
CSTRING name;
logical_t success;

>>--logical_t = context->LoadLibrary(name);-------------------------------><
```

Loads an external library with the given name and adds it to the global Rexx environment.

Arguments

name   The ASCII-Z path/name of the library package, in format required by
                the ::REQUIRES LIBRARY directive.

Returns

True if the library was successfully loaded or the library had been previously loaded. False is returned for any errors in loading the package.

8.16.97. LoadPackage

Context
Thread
Method
Function
Exit
Interpreter

```
CSTRING name;
RexxPackageObject pkg;

>>--pkg = context->LoadPackage(name);-------------------------------><
```

Returns the Package object loaded from the specified file path/name.

Arguments

name   The ASCII-Z path/name of the Rexx package source file.

Returns

The loaded Package object. Any errors resulting from loading the package will return a NULLOBJECT value. Information about errors can be retrieved using GetConditionInfo() (*Section 8.16.48, "GetConditionInfo"*).

8.16.98. LoadPackageFromData

Context
Thread

Method
Function
Exit
Interpreter

```
CSTRING name, data;
size_t sz;
RexxPackageObject pkg;

>>--pkg = context->LoadPackageFromData(name, data, sz);--------------><
```

Returns the loaded package object from the specified file path/name.

Arguments

name	The ASCII-Z name assigned to the package.
data	Data buffer containing the package Rexx.
sz	The size of the *data* buffer.

Returns

The loaded Package object. Any errors resulting from loading the package will return a NULLOBJECT value. Information about errors can be retrieved using GetConditionInfo() (*Section 8.16.48, "GetConditionInfo"*).

8.16.99. Logical

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag, n;

>>--flag = context->Logical(obj, &n);-------------------------------------><

>>--obj = context->Logical(n);-------------------------------------------><
```

This method has two forms. The first form converts an Object to a C logical value (0 or 1). The second form converts a C logical value to an Object.

Arguments

obj	The object to be converted.
n	For the first method form, a C pointer to a logical_t to receive the conversion result. For the second form, a logical_t to be converted to an Object.

Returns

For the first method form, 1 = success and 0 = conversion error, with the converted value placed in *n*
For the second form, an Object version of the logical value.

8.16.100. LogicalToObject

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 logical_t flag, n;

 >>--obj = context->LogicalToObject(n);----------------------------------><
```

Converts a C logical value to an Object.

Arguments

n	The logical_t value to be converted..

Returns

Either the .false or .true object is returned.

8.16.101. MutableBufferCapacity

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxMutableBufferObject obj;
 size_t sz;

 >>--sz = context->MutableBufferCapacity(obj);------------------------------------><
```

Return the current buffer size of the MutableBuffer. The capacity is the total size of the buffer. The length value is the amount of data currently contained in the buffer.

Arguments

obj   The source MutableBuffer object.

Returns

The size of the MutableBuffer object's data area.

8.16.102. MutableBufferData

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxMutableBufferObject obj;
 POINTER str;

 >>--str = context->MutableBufferData(obj);-------------------------------------><
```

Returns a pointer to a MutableBuffer object's data area.

Arguments

obj   The source MutableBuffer object.

Returns

The C pointer to the MutableBuffer object's data area.

8.16.103. MutableBufferLength

Context
Thread
Method
Function

Exit
Interpreter

```
RexxMutableBufferObject obj;
size_t sz;

>>--sz = context->MutableBufferLength(obj);-------------------------------------><
```

Return the current length of the data in a MutableBuffer object's data area.

Arguments

obj   The source MutableBuffer object.

Returns

The length of data in the MutableBuffer object's data area.

8.16.104. NewArray

Context
Thread
Method
Function
Exit
Interpreter

```
RexxArrayObject obj;
size_t len;

>>--obj = context->NewArray(d);---------------------------------------------><
```

Create an Array object of the specified size.

Arguments

d   The size of the Array.

Returns

The new Array object.

8.16.105. NewBuffer

Context

Thread
Method
Function
Exit
Interpreter

```
RexxBufferObject obj;
size_t len;

>>--obj = context->NewBuffer(len);---------------------------------------><
```

Create a Buffer object with a specific data size.

Arguments

  len   The maximum length of the buffer.

Returns

The new Buffer object.

8.16.106. NewBufferString

Context
Thread
Method
Function
Exit
Interpreter

```
RexxBufferStringObject obj;
size_t len;

>>--obj = context->NewBufferString(len);--------------------------------><
```

Create a RexxBufferString with the indicated buffer size. A RexxBufferString is a mutable String object that can be used to construct return values. You must use FinishBufferString() (*Section 8.16.40, "FinishBufferString"*) to transform this into a completed String object.

Arguments

  len   The maximum length of the final String object.

Returns

A new RexxBufferString value.

8.16.107. NewDirectory

Context
Thread
Method
Function
Exit
Interpreter

```
RexxDirectoryObject obj;

>>--obj = context->NewDirectory();----------------------------------------><
```

Create a Directory object.

Arguments

None

Returns

The new Directory object.

8.16.108. NewMethod

Context
Thread
Method
Function
Exit
Interpreter

```
RexxMethodObject obj;
CSTRING name, code;
size_t sz;

>>--obj = context->NewMethod(name, code, sz);---------------------------><
```

Create a new Method object from an in-memory buffer.

Arguments

name   ASCII-Z name of the method.

code	A data buffer containing the new method's Rexx code.
sz	Size of the *code* buffer.

Returns

The created Method object. Any errors resulting from creating the method will return a NULLOBJECT value. Information about any error can be retrieved using GetConditionInfo() (*Section 8.16.48, "GetConditionInfo"*).

8.16.109. NewMutableBuffer

Context
Thread
Method
Function
Exit
Interpreter

```
RexxMutableBufferObject obj;
size_t len;

>>--obj = context->NewMutableBuffer(len);-------------------------------><
```

Create a MutableBuffer object with a specific initial capacity. The new buffer will have an initial length of 0.

Arguments

len	The initial capacity of the buffer.

Returns

The new MutableBuffer object.

8.16.110. NewPointer

Context
Thread
Method
Function
Exit
Interpreter

```
RexxPointerObject obj;
POINTER p;

>>--obj = context->NewPointer(p);---------------------------------------><
```

Create a new Pointer object from a C pointer.

Arguments

p   The source C pointer.

Returns

The created Pointer object.

8.16.111. NewRoutine

Context
Thread
Method
Function
Exit
Interpreter

```
RexxRoutineObject obj;
CSTRING name, code;
size_t sz;

>>--obj = context->NewRoutine(name, code, sz);---------------------------><
```

Create a new Routine object from an in-memory buffer.

Arguments

name   ASCII-Z name of the routine.

code   Buffer containing the routine Rexx code.

sz   Size of the *code* buffer.

Returns

The new Routine object. Any errors resulting from creating the the routine will return a NULLOBJECT value. Information about errors can be retrieved using GetConditionInfo() (*Section 8.16.48, "GetConditionInfo"*).

8.16.112. NewStem

Context
Thread
Method
Function
Exit
Interpreter

```
RexxStemObject obj;
CSTRING str;

>>--obj = context->NewStem(str);----------------------------------------><
```

Create an new Stem object with the specified base name.

Arguments

str   The base name for the new Stem object.

Returns

The new Stem object.

8.16.113. NewString

Context
Thread
Method
Function
Exit
Interpreter

```
RexxStringObject obj;
CSTRING str;
size_t len;

>>--obj = context->NewString(str, len);----------------------------------><

>>--obj = context->NewString(str);----------------------------------------><
```

There are two forms of this method. Both create a new String object from program data.

Arguments

str   For the first form, a pointer to a null-terminated ASCII-Z string. For the second
                form, a pointer to a data buffer containing the string data.

len   Length of the *str* string.

Returns

The new String object.

8.16.114. NewSupplier

Context
Thread
Method
Function
Exit
Interpreter

```
RexxSupplierObject obj;
RexxArrayObject arr1, arr2;

>>--obj = context->NewSupplier(arr1, arr2);------------------------------><
```

This method returns a Supplier object based on the supplied argument Arrays.

Arguments

arr1   The Array of supplier items.

arr2   The Array of supplier item indexes.

Returns

The new Supplier object.

8.16.115. Nil

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;

>>--obj = context->Nil();-------------------------------------------------><
```

Returns the Rexx Nil object.

Arguments

None.

Returns

The Rexx Nil object.

8.16.116. NullString

Context
Thread
Method
Function
Exit
Interpreter

```
RexxStringObject obj;

>>--obj = context->NullString();---------------------------------------><
```

This method returns a string object of zero length.

Arguments

None.

Returns

A null String object.

8.16.117. ObjectToCSelf

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 POINTER ptr;

 >>--ptr = context->ObjectToCSelf(obj);-------------------------------><
```

Returns a pointer to the CSELF value for another object. CSELF is a special argument type used for classes to store native pointers or structures inside an object instance. ObjectToCSelf() will search all of the object's variable scopes searching for a variable named CSELF. If a CSELF variable is located and the value is an instance of either the Pointer or the Buffer class, the corresponding POINTER value will be returned as a void * value. Objects that rely on CSELF values typically set the variable CSELF inside an INIT method for the class.

Arguments

   obj   The source object.

Returns

The CSELF value for the object. Returns NULL if no CSELF value was found in the target object.

8.16.118. ObjectToCSelfScoped

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 POINTER ptr;

 >>--ptr = context->ObjectToCSelfScoped(obj,scope);--------------------><
```

Returns a pointer to the CSELF value for another object. CSELF is a special argument type used for classes to store native pointers or structures inside an object instance. ObjectToCSelfScoped() will search scopes searching for a variable named CSELF, beginning with the indicated scope level. If a CSELF variable is located and the value is an instance of either the Pointer or the Buffer class, the corresponding POINTER value will be returned as a void * value. Objects that rely on CSELF values typically set the variable CSELF inside an INIT method for the class.

Arguments

   obj   The source object.
   scope   A class object indicating the starting scope.

Returns

The CSELF value for the object. Returns NULL if no CSELF value was found in the target object.

8.16.119. ObjectToDouble

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
double n;
logical_t flag;

>>--flag = context->ObjectToDouble(obj, &n);---------------------------><
```

Converts an Object to a C double value.

Arguments

obj	The source object for the conversion.
n	A returned converted value.

Returns

1 = success, 0 = conversion error. The converted value is placed in *n*.

8.16.120. ObjectToInt32

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
int32_t n;
logical_t flag;

>>--flag = context->ObjectToInt32(obj, &n);----------------------------><
```

Convert an Object into a 32-bit integer.

Arguments

obj	The object to convert.
n	The conversion result.

Returns

1 = success, 0 = conversion error. The converted value is placed in *n*.

8.16.121. ObjectToInt64

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
int64_t n;
logical_t flag;

>>--flag = context->ObjectToInt64(obj, &n);----------------------------->< 
```

Convert an Object into a 64-bit integer.

Arguments

obj	The object to be converted.
n	The conversion result.

Returns

1 = success, 0 = conversion error. The converted value is placed in *n*.

8.16.122. ObjectToIntptr

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
intptr_t n;
logical_t flag;

>>--flag = context->ObjectToIntptr(obj, &n);---------------------------><
```

Convert an Object to an intptr_t value.

Arguments

obj	The object to convert.
n	The conversion result.

Returns

1 = success, 0 = conversion error. The converted value is placed in *n*.

8.16.123. ObjectToLogical

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag, n;

>>--flag = context->ObjectToLogical(obj, &n);---------------------------><
```

Converts an Object to a C logical value (0 or 1).

Arguments

obj	The object to convert.
n	The conversion result.

Returns

1 = success, 0 = conversion error. The converted value is placed in *n*.

8.16.124. ObjectToString

Context
Thread

Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
RexxStringObject str;

>>--str = context->ObjectToString(obj);----------------------------------><
```

Convert an Object to a String object.

Arguments

obj   The source object for the conversion.

Returns

The String object.

8.16.125. ObjectToStringSize

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
size_t n;
logical_t flag;

>>--flag = context->ObjectToStringSize(obj, &n);------------------------><
```

Convert an Object to a stringsize_t number value.

Arguments

obj   The object to convert.
n   The conversion result.

Returns

1 = success, 0 = conversion error. The converted value is placed in *n*.

8.16.126. ObjectToStringValue

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
CSTRING str;

>>--str = context->ObjectToStringValue(obj);---------------------------><
```

Convert an Object to a C ASCII-Z string.

Arguments

obj   The source object for the conversion.

Returns

The C ASCII-Z string representation of the object.

8.16.127. ObjectToUintptr

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
uintptr_t n;
logical_t flag;

>>--flag = context->ObjectToUintptr(obj, &n);--------------------------><
```

Convert an Object to an uintptr_t value.

Arguments

obj   The object to convert.

n	The conversion result.

Returns

1 = success, 0 = conversion error. The converted value is placed in *n*.

8.16.128. ObjectToUnsignedInt32

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 uint32_t n;
 logical_t flag;

 >>--flag = context->ObjectToUnsignedInt32(obj, &n);----------------------><
```

Convert an Object to an uint32_t value.

Arguments

obj	The object to convert.
n	The conversion result.

Returns

1 = success, 0 = conversion error. The converted value is placed in *n*.

8.16.129. ObjectToUnsignedInt64

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 uint64_t n;
 logical_t flag;
```

```
>>--flag = context->ObjectToUnsignedInt64(obj, &n);--------------------><
```

Convert an Object to an uint64_t value.

Arguments

obj	The object to convert.
n	The conversion result.

Returns

1 = success, 0 = conversion error. The converted value is placed in *n*.

8.16.130. ObjectToValue

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
ValueDescriptor desc;
logical_t flag;

>>--flag = context->ObjectToValue(obj, &desc);--------------------------><
```

Convert a Rexx object to another type. The target type is identified by the ValueDescriptor
structure, and can by any of the types that may be used as a method or routine return type.
For many conversions, it may be more appropriate to use more targeted routines such as
ObjectToWholeNumber(). ObjectToValue() is capable of conversions to types such as int8_t for which
there are no specific conversion APIs.

Arguments

obj	The object to be converted.
desc	A C pointer to a ValueDescriptor struct that identifies the conversion type. The converted value will be stored in the ValueDescriptor if successful.

Returns

1 = success, 0 = conversion error. If successful, *desc* is updated with the converted value of the
requested type.

8.16.131. ObjectToWholeNumber

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
wholenumber_t wn;
logical_t flag;

>>--flag = context->ObjectToWholeNumber(obj, &wn);----------------------><
```

Convert an Object to a whole number value.

Arguments

obj	The object to convert.
n	The conversion result.

Returns

1 = success, 0 = conversion error. The converted value is placed in *n*.

8.16.132. PointerValue

Context
Thread
Method
Function
Exit
Interpreter

```
RexxPointerObject obj;
POINTER p;

>>--p = context->PointerValue(obj);---------------------------------------><
```

Return the wrappered C pointer value from a RexxPointerObject.

Arguments

obj	The source RexxPointerObject.

Returns

The wrappered C pointer value.

8.16.133. RaiseCondition

Context
Thread
Method
Function
Exit
Interpreter

```
CSTRING str;
RexxStringObject sobj;
RexxArrayObject arr;
RexxObjectPtr obj;

>>--context->RaiseCondition(str, sobj, add, obj);------------------------><
```

Raise a condition. The raised condition is held in a pending state until the method, routine, or exit returns to the Rexx runtime. This is similar to using the RAISE instruction to raise a condition from Rexx code.

Arguments

str   The condition name.
sobj   The optional condition description as a String object.
add   A optional object containing additional condition information.
obj   A Object that will be returned as a routine or method result if the raised condition is
             not trapped by the caller.

Returns

Void.

8.16.134. RaiseException

Context
Thread
Method
Function
Exit
Interpreter

```
size_t n;
RexxObjectPtr obj;

>>--context->RaiseException(n, obj);-------------------------------------><
```

Raise a SYNTAX condition. The raised condition is held in a pending state until the method, routine, or exit returns to the Rexx runtime. This is similar to using the RAISE instruction to raise a SYNTAX condition from Rexx code.

Arguments

n	The exception condition number. There are #defines for the recognized condition errors in the oorexxerrors.h include file.
obj	An Array of error message substitution values.

Returns

Void.

8.16.135. RaiseException0

Context
Thread
Method
Function
Exit
Interpreter

```
size_t n;

>>--context->RaiseException0(n);-----------------------------------------><
```

Raise an exception condition with no message substitution values.

Arguments

n	The exception condition number.

Returns

Void.

8.16.136. RaiseException1

Context
Thread

Method
Function
Exit
Interpreter

```
size_t n;
RexxObjectPtr obj;

>>--context->RaiseException1(n, obj);------------------------------------><
```

Raise an exception condition with a single message substitution value.

Arguments

| *n* | The exception condition number. |
| *obj* | A substitution value for the condition error message. |

Returns

Void.

8.16.137. RaiseException2

Context
Thread
Method
Function
Exit
Interpreter

```
size_t n;
RexxObjectPtr obj1, obj2;

>>--context->RaiseException2(n, obj1, obj2);----------------------------><
```

Raise an exception condition with two message substitution values.

Arguments

n	The exception condition number.
obj1	The first substitution value.
obj2	The second substitution value.

Returns

Void.

8.16.138. RegisterLibrary

Context
Thread
Method
Function
Exit
Interpreter

```
CSTRING name;
logical_t success;

>>--logical_t = context->RegisterLibrary(name, table);--------------------------------><
```

Registers an in-process library package with the global Rexx environment. The package is processed as if it is loaded from an external library, but without requiring the library packaging.

Arguments

name   The ASCII-Z path/name of the library package, in format required by the ::REQUIRES LIBRARY directive.

table   A pointer to a RexxPackageEntry (*Section 8.11, "Building an External Native Library"*) table defining the contents of the package.

Returns

True if the library was successfully registered. False is returned if a package has already be loaded or registered with the given name.

8.16.139. ReleaseGlobalReference

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr ref;

>>--context->ReleaseGlobalReference(ref);-------------------------------><
```

Release access to a global object reference. This removes the global garbage collection protection from the object reference. Once released, *ref* should no longer be used for object operations.

Arguments

  ref   A global Rexx object reference.

Returns

Void.

8.16.140. ReleaseLocalReference

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr ref;

 >>--context->ReleaseLocalReference(ref);---------------------------------><
```

Removes local context protection from an object reference. Once released, *ref* should no longer be used for object operations.

Arguments

  ref   The local Rexx object reference.

Returns

Void.

8.16.141. RequestGlobalReference

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr ref, obj;

>>--ref = context->RequestGlobalReference(obj);-------------------------><
```

Requests global garbage collection protection for an object reference. The returned value may be saved in native code control blocks and used as an object reference in any API context. The *obj* will be protected from garbage collection until the global reference is released with ReleaseGlobalReference() (*Section 8.16.139, "ReleaseGlobalReference"*).

Arguments

obj   The Rexx object to be protected.

Returns

A global reference to this object that can be saved and used in any API context.

8.16.142. ResolveStemVariable

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
RexxStemObject stem;

>>--stem = context->ResolveStemVariable(obj);-------------------------><
```

Resolves a stem variable object using the same mechanism applied to RexxStemObject arguments passed to routines. If *obj* is a Stem object, the same Stem object will be returned. If *obj* is a String object, the string value is used to resolve a stem variable from the caller's variable context. The Stem object value of the referenced stem variable is returned as a result.

Arguments

obj   The source object to be resolved to a Stem object.

Returns

The resolved Stem object.

8.16.143. SendMessage

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj, ret;
RexxArrayObject arr;

>>--ret = context->SendMessage(obj, meg, arr);--------------------------><
```

Send a message to an Object.

Arguments

obj	The object to receive the message.
msg	An ASCII-Z string containing the message name. This argument will be converted to upper case automatically.
arr	The Array of message arguments.

Returns

The returned object. If the method does not return an object then NULLOBJECT is returned. Any errors resulting from invoking the method will return a NULLOBJECT value. The CheckCondition() (*Section 8.16.20, "CheckCondition"*). can be used to check if an error occurred during the call.

8.16.144. SendMessage0

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj, ret;

>>--ret = context->SendMessage0(obj, meg);----------------------------><
```

Send a message to an Object. This is a short cut method when no arguments are needed by the receiving object method.

Arguments

obj	The object to receive the message.
msg	An ASCII-Z string containing the message name. This argument will be converted to upper case automatically.

Returns

The returned object. If the method does not return an object then NULLOBJECT is returned. Any errors resulting from invoking the method will return a NULLOBJECT value. The CheckCondition() (*Section 8.16.20, "CheckCondition"*). can be used to check if an error occurred during the call.

8.16.145. SendMessage1

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj, ret, arg1;

 >>--ret = context->SendMessage2(obj, meg, arg1);------------------------><
```

Send a message to an Object. This is a short cut method when only one argument is needed by the receiving object method.

Arguments

obj	The object to receive the message.
msg	An ASCII-Z string containing the message name. This argument will be converted to upper case automatically.
arg1	The first argument to the receiving method.

Returns

The returned object. If the method does not return an object then NULLOBJECT is returned. Any errors resulting from invoking the method will return a NULLOBJECT value. The CheckCondition() (*Section 8.16.20, "CheckCondition"*). can be used to check if an error occurred during the call.

8.16.146. SendMessage2

Context
Thread
Method

Function
Exit
Interpreter

```
 RexxObjectPtr obj, ret, arg1, arg2;

 >>--ret = context->SendMessage2(obj, meg, arg1, arg2);------------------><
```

Send a message to an Object. This is a short cut method when only two arguments are needed by the receiving object method.

Arguments

obj	The object to receive the message.
msg	An ASCII-Z string containing the message name. This argument will be converted to upper case automatically.
arg1	The first argument to the receiving method.
arg2	The second argument to the receiving method.

Returns

The returned object. If the method does not return an object then NULLOBJECT is returned. Any errors resulting from invoking the method will return a NULLOBJECT value. The CheckCondition() (*Section 8.16.20, "CheckCondition"*). can be used to check if an error occurred during the call.

8.16.147. SetContextVariable

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 CSTRING name;

 >>--context->SetContextVariable(name, obj);--------------------------><
```

Sets the value of a Rexx variable in the current function context. Only simple and stem variables may be set using SetContextVariable(). Compound variable values may be set by retrieving the Stem object associated with a stem variable and using SetStemElement() (*Section 8.16.154, "SetStemElement"*) to set the associated compound variable.

Arguments

name	The name of the Rexx variable.
obj	The object to assign to the variable.

Returns

Void.

8.16.148. SetGuardOff

Context
Thread
Method
Function
Exit
Interpreter

```
>>--context->SetGuardOff();---------------------------------------><
```

Release the guard lock for this method scope.

Arguments

None.

Returns

Void.

8.16.149. SetGuardOn

Context
Thread
Method
Function
Exit
Interpreter

```
>>--context->SetGuardOn();---------------------------------------><
```

Obtain the guard lock for this object scope.

Arguments

None.

Returns

Void.

8.16.150. SetMutableBufferCapacity

Context
Thread
Method
Function
Exit
Interpreter

```
RexxMutableBufferObject obj;
size_t len;
POINTER data;

>>--data = context->SetMutableBufferCapacity(obj,len);-----------------------------------><
```

Ensure the MutableBuffer object's data area is at least the indicated size. If necessary, the internal data area will be reallocated. SetMutableBufferCapacity will only change the capacity if *len* is larger than the current buffer capacity.

Arguments

obj   The source MutableBuffer object.

len   The required buffer capacity. If *len* is larger than the current data area, the internal data area will be reallocated to the larger size and any existing buffer data will be copied to the new data area.

Returns

A pointer to the MutableBuffer's data area. Because SetMutableBufferCapacity() may reallocate the data area, the return value should replace any previous buffer pointers.

8.16.151. SetMutableBufferLength

Context
Thread
Method
Function
Exit
Interpreter

```
RexxMutableBufferObject obj;
size_t len;
size_t newLen;

>>--newLen = context->SetMutableBufferLength(obj,len);---------------------------------><
```

Sets the length of the data in the MutableBuffer's data area. If the length is greater than the current capacity, then it will be capped at the current capacity. If *len* is longer than the buffer's current data length, data will be padded with '00'x characters for the additional length. When adding characters to the buffer's data area, you should call SetMutableBufferLength() before copying the additional data into the buffer. If additional capacity is required, SetMutableBufferCapacity to increase the buffer size.

Arguments

obj	The source MutableBuffer object.
len	The new data length. If *len* is larger than the current data area, the new length will be capped at the length of the data area.

Returns

The new data length, which may be less than the indicated length if the buffer capacity is smaller.

8.16.152. SetObjectVariable

Context
Thread
Method
Function
Exit
Interpreter

```
CSTRING str;
RexxObjectPtr obj;

>>--context->SetObjectVariable(str, obj);------------------------------><
```

Sets an instance variable in the current method's variable scope to a new value. Only simple and stem variables may be set using this API.

Arguments

str	The name of the object variable.
obj	The object to assign to the object variable.

Returns

Void.

8.16.153. SetStemArrayElement

Context
Thread
Method
Function
Exit
Interpreter

```
RexxStemObject sobj;
RexxObjectPtr obj;
size_t n;

>>--context->SetStemArrayElement(sobj, n, obj);------------------------><
```

Sets an element of the Stem object. If the element exists it is replaced. This method uses a numeric index as the element name.

Arguments

sobj	The target Stem object.
n	The Stem object element number.
obj	The object value assigned to the Stem object element.

Returns

Void.

8.16.154. SetStemElement

Context
Thread
Method
Function
Exit
Interpreter

```
RexxStemObject sobj;
RexxObjectPtr obj;
CSTRING name;

>>--context->SetStemElement(sobj, name, obj);--------------------------><
```

Sets an element of the Stem object. If the element exists it is replaced.

Arguments

sobj	The target Stem object.
name	The Stem object element name. This is a fully resolve Stem tail element.
obj	The object value assigned to the Stem object element.

Returns

Void.

8.16.155. SetThreadTrace

Context
Thread
Method
Function
Exit
Interpreter

```
logical_t flag;

>>--context->SetThreadTrace(flag);---------------------------------------><
```

Sets the interactive trace state for the current thread.

Arguments

flag	New state for interactive trace.

Returns

Void.

8.16.156. SetTrace

Context
Thread
Method
Function
Exit
Interpreter

```
 logical_t flag;

 >>--context->SetTrace(flag);--------------------------------------------><
```

Sets the interactvive trace state for the interpreter instance. This will enable tracing in all active threads for the interpreter instance.

Arguments

flag	The new trace state.

Returns

Void.

8.16.157. String

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxRoutineObject obj;
 CSTRING str;
 size_t len;

 >>--obj = context->String(str, len);-------------------------------------><

 >>--obj = context->String(str);------------------------------------------><
```

There are two forms of this method. Both create a new String object from a C string.

Arguments

str	The ASCII-Z string to be converted.
len	Length of the *str* string.

Returns

A new String object.

8.16.158. StringData

Context

Thread	
Method	
Function	
Exit	
Interpreter	

```
RexxObjectPtr obj;
CSTRING str;

>>--str = context->StringData(obj);-------------------------------------><
```

Returns a read-only pointer to the String object's string data.

Arguments

obj   The source String object for the data.

Returns

A pointer to the String object's string data.

8.16.159. StringGet

Context	
Thread	
Method	
Function	
Exit	
Interpreter	

```
RexxObjectPtr obj;
POINTER str;
size_t c, len1, len2;

>>--c = context->StringGet(obj, len1, str, len2);-----------------------><
```

Copies all or part of the String object to a C string buffer.

Arguments

obj   The source String object.

len1   The starting position within the String. This argument is 1-based

str   A pointer to the target buffer for the copy. Note that the buffer is NOT zero-terminated.

len2 The number of characters to copy. This argument should be less than or equal the size of the *str* buffer or a buffer overrun will result.

Returns

The number of characters actually copied.

8.16.160. StringLength

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
size_t sz;

>>--sz = context->StringLength(obj);-------------------------------------><
```

Return the length a String object.

Arguments

obj The source String object.

Returns

The string length of the String object.

8.16.161. StringLower

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr srcobj, newobj;

>>--newobj = context->StringLower(srcobj);------------------------------><
```

Convert a String object to lower case, returning a new String object.

Arguments

srcobj　　　　The source String object to be converted to lower case.

Returns

A new String object with the string value lower cased.

8.16.162. StringSize

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxObjectPtr obj;
 logical_t flag;
 stringsize_t n;

 >>--obj = context->StringSize(&n);------------------------------------------><

 >>--flag = context->StringSize(obj, &n);--------------------------------------><
```

There are two forms of this method. The first converts the stringsize_t value *n* to an Object. The second converts an Object to a stringsize_t value and returns it in *n*.

Arguments

n　　　　For the first form, the stringsize_t value to be converted. For the second form, the target of the conversion.

obj　　　　The object to be converted.

Returns

For the first form, an Object representation of the integer value. For the second form, 1 = success, 0 = failure. If successful, the converted value is placed in *n*.

8.16.163. StringSizeToObject

Context
Thread

Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
size_t sz;

>>--obj = context->StringSizeToObject(sz);------------------------------><
```

Convert a stringsize_t value to an Object.

Arguments

sz   The stringsize_t value to be converted.

Returns

an Object that represents the C stringsize_t value.

8.16.164. StringUpper

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr srcobj, newobj;

>>--newobj = context->StringUpper(srcobj);------------------------------><
```

Convert a String object upper case, returning a new String object.

Arguments

srcobj   The source String object.

Returns

A new String object with the string value upper cased.

8.16.165. SupplierAvailable

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxSupplierObjectPtr sobj;
 logical_t flag;

 >>--flag = context->SupplierAvailable(sobj);----------------------------><
```

Returns 1 if there is another supplier item available.

Arguments

  sobj   The source supplier object.

Returns

1 = another item available, 0 = no item available.

8.16.166. SupplierIndex

Context
Thread
Method
Function
Exit
Interpreter

```
 RexxSupplierObjectPtr sobj;
 RexxObjectPtr obj;

 >>--obj = context->SupplierIndex(sobj);----------------------------------><
```

Return the current supplier object index value.

Arguments

  sobj   The source supplier object.

Returns

The index object at the current supplier position.

8.16.167. SupplierItem

Context
Thread
Method
Function
Exit
Interpreter

```
RexxSupplierObjectPtr sobj;
RexxObjectPtr obj;

>>--obj = context->SupplierItem(sobj);----------------------------------><
```

Return the current supplier item object.

Arguments

sobj The source supplier object.

Returns

The object item at the current supplier position.

8.16.168. SupplierNext

Context
Thread
Method
Function
Exit
Interpreter

```
RexxSupplierObjectPtr sobj;

>>--context->SupplierNext(sobj);----------------------------------------><
```

Advance a Supplier object to the next enumeration position.

Arguments

sobj The source supplier object.

Returns

Void.

8.16.169. Terminate

Context
Thread
Method
Function
Exit
Interpreter

```
>>--context->Terminate();---------------------------------------><
```

Terminates the current Rexx interpreter instance. Terminate() may only be called from the thread context that originally created the interpreter instance. This call will wait for all threads to complete processing before returning.

Arguments

None.

Returns

Void.

8.16.170. True

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;

>>--obj = context->True();--------------------------------------><
```

This method returns the Rexx .true object.

Arguments

None.

Returns

The Rexx .true object.

8.16.171. Uintptr

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;
uintptr_t n;

>>--obj = context->Uintptr(&n);-----------------------------------------><

>>--flag = context->Uintptr(obj, &n);------------------------------------><
```

There are two forms of this method. The first converts the uintptr_t value *n* to an Object. The second converts an Object to a uintptr_t value and returns it in *n*.

Arguments

n	For the first form, the uintptr_t value to be converted. For the second form, the target of the conversion.
obj	The object to be converted.

Returns

For the first form, an Object version of the integer. The second form returns 1 = success, 0 = failure. If successful, the converted value is placed in **n**.

8.16.172. UintptrToObject

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
uintptr_t n;

>>--obj = context->UintptrToObject(&n);-------------------------------><
```

Convert a uintptr_t value *n* to an Object.

Arguments

n   The uintptr_t value to be converted.

Returns

An Object that represents the uintptr_t value.

8.16.173. UnsignedInt32

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
logical_t flag;
uint32_t n;

>>--obj = context->UnsignedInt32(n);------------------------------------><

>>--flag = context->UnsignedInt32(obj, &n);----------------------------><
```

There are two forms of this method. The first converts a C 32-bit unsigned integer *n* to an Object. The second converts an Object to a uint32_t value and returns it in *n*.

Arguments

n   For the first form, the uint32_t value to be converted. For the second form, the target of the conversion.

n   The object to be converted to a uint32_t value.

Returns

For the first form, an Object version of the integer value. For the second form, returns 1 = success, 0 = failure. If successful, the converted value is placed in *n*.

8.16.174. UnsignedInt32ToObject

Context	
Thread	
Method	
Function	
Exit	
Interpreter	

```
RexxObjectPtr obj;
uint32_t n;

>>--obj = context->UnsignedInt32ToObject(n);----------------------------><
```

Convert a C 32-bit unsigned integer *n* to an Object.

Arguments

n   The uint32_t value to be converted.

Returns

An Object that represents the C unsigned integer.

8.16.175. UnsignedInt64

Context	
Thread	
Method	
Function	
Exit	
Interpreter	

```
RexxObjectPtr obj;
logical_t flag;
uint64_t n;

>>--obj = context->UnsignedInt64(n);-------------------------------------><

>>--flag = context->UnsignedInt64(obj, &n);-----------------------------><
```

There are two forms of this method. The first converts a C 64-bit unsigned integer *n* to an Object. The second converts an Object to a uint64_t value and returns it in *n*.

Arguments

n   For the first form, the uint64_t value to be converted. For the second form, the target of the conversion.

n	The object to be converted.

Returns

For the first form, an Object version of the integer value. For the second form, returns 1 = success, 0 = failure. If successful, the converted value is placed in *n*.

8.16.176. UnsignedInt64ToObject

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
uint64_t n;

>>--obj = context->UnsignedInt64ToObject(n);----------------------------><
```

Convert a C 64-bit unsigned integer *n* to an Object.

Arguments

n	The uint64_t value to be converted.

Returns

An Object that represents the C unsigned integer.

8.16.177. ValuesToObject

Context
Thread
Method
Function
Exit
Interpreter

```
RexxArrayObj obj;
ValueDescriptor desc[3];
```

```
>>--obj = context->ValuesToObject(desc);-------------------------------><
```

Converts an array of ValueDescriptor structs to an Array of objects.

Arguments

desc   A C pointer to the ValueDescriptor struct array to be converted. The end of the
                array is marked by a ValueDescriptor struct with all fields set to zero.

Returns

A Array object containing the converted objects.

8.16.178. ValueToObject

Context	
Thread	
Method	
Function	
Exit	
Interpreter	

```
RexxObjectPtr obj;
ValueDescriptor desc;;

>>--obj = context->ValueToObject(&desc);-------------------------------><
```

Convert a type to an Object representation. The source type is identified by the ValueDescriptor
structure, and can by any of the types that may be used as a method or routine return types.
For many conversions, it may be more appropriate to use more targeted routines such as
WholeNumberToObject(). ValueToObject() is capable of converting to types such as int8_t for which
there are no specific conversion APIs.

Arguments

desc   A C pointer to the ValueDescriptor struct describing the source value.

Returns

The object representing the converted value.

8.16.179. WholeNumber

Context	
Thread	

Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
wholenumber_t n;
logical_t flag;

>>--obj = context->WholeNumber(n);---------------------------------------><

>>--flag = context->WholeNumber(obj, &n);--------------------------------><
```

There are two forms of this method. The first form converts a wholenumber_t value to an Object. The second form converts an Object to a wholenumber_t value and returns it in *n*.

Arguments

n	For the first form, the wholenumber_t value to be converted. For the second form, the target of the conversion.
obj	The source object for the conversion.

Returns

For the first form, an Object version of the integer value. For the second form, returns 1 = success, 0 = failure. If successful, the converted value is placed in *n*.

8.16.180. WholeNumberToObject

Context
Thread
Method
Function
Exit
Interpreter

```
RexxObjectPtr obj;
wholenumber_t n;

>>--obj = context->WholeNumberToObject(n);--------------------------------><
```

Convert a C wholenumber_t value to an Object.

Arguments

n	The C whole number to be converted.

Returns

An Object that represents the C whole number.

Classic 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 *Section 9.7, "Variable Pool Interface"*), and execute Rexx routines directly from memory (see *Section 9.11, "Macrospace Interface"*).

9.1. Handler Characteristics

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

- Rexx handlers must use the REXXENTRY 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 loadable library (dynamic-link library (DLL) on Windows, or shared library on Unix-based systems.).

  - An entry point within an executable (EXE) 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 dynamic-

link library external function identifies both the dynamic-link library and routine that contains the external function. Also note:

- Dynamic-link 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.

9.2. 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:

Example 9.1. RXSTRING

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

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

Many programming interfaces use RXSTRINGs for input-only operations. These APIs use a constant version of the RXSTRING, the CONSTRXSTRING.

Example 9.2. RXSTRING

```
typedef struct {
    size_t          strlength;    /* length of string          */
    const char *    strptr;       /* pointer to string         */
} RXSTRING;

typedef CONSTRXSTRING *PCONSTRXSTRING;     /* pointer to a CONSTRXSTRING     */
```

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 **RexxAllocateMemory(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.

9.3. Calling the Rexx Interpreter

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

9.3.1. From the Operating System

You can run a Rexx program directly from the operating system command prompt using Rexx followed by the program name. See *Section 2.2, "Running a Rexx Program"*.

9.3.2. From within an Application

The Rexx interpreter is a dynamic-link library (DLL) routine (or Unix/Linux shared object). 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.

9.3.3. The RexxStart Function

RexxStart calls the Rexx interpreter to run a Rexx procedure.

```
retc = RexxStart(ArgCount, ArgList, ProgramName, Instore, EnvName,
                 CallType, Exits, ReturnCode, Result);
```

9.3.3.1. Parameters

ArgCount (size_t) - 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 (PCONSTRXSTRING) - input

> is an array of CONSTRXSTRING structures that are the Rexx program arguments.

ProgramName (const char *) - 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, drive, and path. If you do not specify a file extension, the default is .REX. A Rexx program can use any extension. If you do not provide the path and the drive, the Rexx interpreter uses the usual file search order to locate the file.

> 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 *Section 9.11, "Macrospace Interface"*).

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 *Section 9.11, "Macrospace Interface"*). 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* is exactly the string "//T" and the *CallType* is RXCOMMAND, the interpreter performs a syntax check on the procedure source, but does not execute it and does not store any images.

> The program calling RexxStart must release *Instore[1]* using **RexxFreeMemory(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 (const char *) - 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 (int) - 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:

Example 9.3. RXSYSEXIT

```
typedef struct {
    const char *    sysexit_name;  /* name of exit handler       */
    int             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 *Section 9.6, "Registered System Exit Interface"* for exit code definitions. An RXENDLST entry identifies the system-exit list end. *Exits* must be null if exits are not used.

ReturnCode (short *) - 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 **RexxAllocateMemory(size)**. The caller of RexxStart is responsible for releasing the RXSTRING storage with **RexxFreeMemory(ptr)**.

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

9.3.3.2. Return Codes

The possible RexxStart return codes are:

negative

> Interpreter errors. See the Appendix in the *Open Object Rexx: 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.

When a macrospace Rexx procedure (see *Section 9.11, "Macrospace Interface"*) is not loaded in the macrospace, the return code is -3 ("Program is unreadable").

9.3.3.3. Example

Example 9.4. RexxStart

The following example for RexxStart should compile and execute on Linux. A few small changes as noted in the example, and it should compile and execute on Windows. This is dependent on having the build environment set up correctly. Note that you need to provide a **test.rex** program for the executable to pass to the interpreter:

```
/* rexxStartExample.c

  gcc -D_GNU_SOURCE -std=c99 -pedantic -ldl rexxStartExample.c -lrexx -lrexxapi -o
 rexxStartExample

*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <rexx.h>

int main(int argc, char *argv[]) {
    int           return_code; /* interpreter return code */
    short         rc;          /* converted return code */

    CONSTRXSTRING argr[1];     /* rexx program argument string */
    RXSTRING      retstr;      /* rexx program return value */

    char          return_buffer[250]; /* returned buffer */
    char          rexx_argument[] = "theargument";
```

```
    /* build the argument string */
    MAKERXSTRING(argr[0], rexx_argument, strlen(rexx_argument));

    /* set up default return */
    MAKERXSTRING(retstr, return_buffer, sizeof(return_buffer));

    retstr.strptr[0] = 0;

    return_code = RexxStart(1,             // one argument
                            argr,          // here it is
                            "./test.rex",  // name of program
                            NULL,          // use disk version
                            "bash",        // default address name
                            RXCOMMAND,     // called as a subcommand
                            NULL,          // no exits
                            &rc,           // where to put rc
                            &retstr);      // where to put returned string

    /* process return value */
    printf("rc %i\n", rc);
    if (retstr.strlength > 0) {
       printf("ret: %s\n", retstr.strptr);
    }

    /* need to return storage? */
    if (RXSTRPTR(retstr) !=  return_buffer) {
       RexxFreeMemory(RXSTRPTR(retstr)); /* release the RXSTRING */
    }
    return 0;
}

/*
  In the above code, change: "./test.rex" to: ".\test.rex" and
  change: "bash" to: "cmd".

  For the VC++ compiler, this command line should work:

  cl rexxStartExample.cpp rexx.lib rexxapi.lib

*/
```

When RexxStart is executed within an external program (usually a C program), the main Rexx thread runs on the same thread as the RexxStart invocation. When the main thread terminates, the interpreter will wait until all additional threads created from the main thread terminate before returning control to the invoking program.

9.3.4. The RexxWaitForTermination Function (Deprecated)

RexxWaitForTermination is not supported in 4.0 and will return immediately if called. This is maintained for binary compatibility with previous releases.

9.3.5. The RexxDidRexxTerminate Function (Deprecated)

RexxDidRexxTerminate always returns 1 for 4.0. This is maintained for binary compatibility with early releases.

```
retc = RexxDidRexxTerminate();
```

9.4. 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.

9.4.1. Registering Subcommand Handlers

A subcommand handler can reside in the same module (executable or DLL) as an application, or it can reside in a separate dynamic-link 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 dynamic-link library. Any Rexx program using the Rexx ADDRESS instruction can access a dynamic-link library subcommand handler. A dynamic-link library subcommand handler can also be registered directly from a Rexx program using the RXSUBCOM command.

9.4.1.1. Creating Subcommand Handlers

The following example is a sample subcommand handler definition.

Example 9.5. Command handler

```
RexxReturnCode REXXENTRY command_handler(
  PCONSTRXSTRING Command,    /* Command string from Rexx        */
  unsigned short *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 **RexxAllocateMemory(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.

9.4.1.1.1. Example

Example 9.6. Subcommand handler

```
RexxReturnCode REXXENTRY Edit_Commands(
  PCONSTRXSTRING Command,     /* Command string passed from the caller   */
  unsigned short *Flags,      /* pointer too short for return of flags    */
  PRXSTRING Retstr)           /* pointer to RXSTRING for RC return        */
{
  int        command_id;                  /* command to process         */
  int        rc;                          /* return code                */
  const char *scan_pointer;               /* current command scan       */
  const char *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      */
}
```

9.4.2. Subcommand Interface Functions

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

9.4.2.1. RexxRegisterSubcomDll

RexxRegisterSubcomDll registers a subcommand handler that resides in a dynamic-link library routine.

```
retc = RexxRegisterSubcomDll(EnvName, ModuleName, EntryPoint,
                             UserArea, DropAuth);
```

9.4.2.1.1. Parameters

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

ModuleName (const char *) - input
    is the address of an ASCII dynamic-link library name. *ModuleName* is the DLL file containing the subcommand handler routine.

EntryPoint (const char *) - input
    is the address of an ASCII dynamic-link library procedure name. *EntryPoint* is the name of the exported routine within *ModuleName* that Rexx calls as a subcommand handler.

UserArea (const char *) - input
    is the address of an area of user-defined information. The user-defined information is a buffer the size of two pointer values. The bytes *UserArea* buffer is 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 (size_t) - 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.

9.4.2.1.2. Return Codes

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 DLL handler with the same name registered in another DLL. (To address this subcommand, you must specify its library name.)
RXSUBCOM_NOTREG	30	Registration was unsuccessful due to duplicate handler and module names (RexxRegisterSubcomExe or RexxRegisterSubcomDll); the subroutine environment is not registered (other Rexx subcommand functions).
RXSUBCOM_NOEMEM	1002	There is insufficient memory available to complete this request.

9.4.2.2. RexxRegisterSubcomExe

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

```
retc = RexxRegisterSubcomExe(EnvName, EntryPoint, UserArea);
```

9.4.2.2.1. Parameters

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

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

UserArea (const char *) - input
>   is the address of an area of user-defined information. The user-defined information is a buffer the size of two pointer values. The bytes *UserArea* buffer is 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.

9.4.2.2.2. Return Codes

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 DLL handler with the same name

		registered in another DLL. (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_NOEMEM	1002	There is insufficient memory available to complete this request.

9.4.2.2.3. 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 dynamic-link library subcommand handlers with the RXSUBCOM command. For example:

Example 9.7. RXSUBCOM

```
                            /* 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 dynamic-link library.

9.4.2.2.4. Example

Example 9.8. RexxStart

```
const char  *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        */
```

9.4.2.3. RexxDeregisterSubcom

RexxDeregisterSubcom deregisters a subcommand handler.

```
retc = RexxDeregisterSubcom(EnvName, ModuleName);
```

9.4.2.3.1. Parameters

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

ModuleName (const char *) - input
> is the address of an ASCII dynamic-link library name. *ModuleName* is the name of the
> dynamic-link 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.

9.4.2.3.2. Return Codes

RXSUBCOM_OK	0	A subcommand has executed successfully.
RXSUBCOM_NOTREG	30	Registration was unsuccessful due to duplicate handler and dynalink 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."

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

9.4.2.4. RexxQuerySubcom

RexxQuerySubcom queries a subcommand handler and retrieves saved user information.

```
retc = RexxQuerySubcom(EnvName, ModuleName, Flag, UserWord);
```

9.4.2.4.1. Parameters

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

ModuleName (const char *) - input
> is the address of an ASCII dynamic-link library name. *ModuleName* restricts the query to a
> subcommand handler within the *ModuleName* dynamic-link 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 (unsigned short *) - 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 (char *) - output

> is the address of an area that receives the user information saved with RexxRegisterSubcomExe or RexxRegisterSubcomDll. The userarea must be large enough to store two pointer values. *UserWord* can be null if the saved user information is not required.

9.4.2.4.2. Return Codes

RXSUBCOM_OK	0	A subcommand has executed successfully.
RXSUBCOM_NOTREG	30	Registration was unsuccessful due to duplicate handler and dynalink names (RexxRegisterSubcomExe or RexxRegisterSubcomDll); the subroutine environment is not registered (other Rexx subcommand functions).

9.4.2.4.3. Example

Example 9.9. Command handlers

```
RexxReturnCode REXXENTRY Edit_Commands(
  PCONSTRXSTRING  Command,    /* Command string passed from the caller   */
  unsigned short *Flags,      /* pointer too short for return of flags    */
  PRXSTRING       Retstr)     /* pointer to RXSTRING for RC return        */
{
  char            *user_info[2];     /* saved user information    */
  char            *global_workarea;  /* application data anchor   */
  unsigned short   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      */
```

9.5. External Function Interface

There are two types of Rexx external functions:

- Routines written in Rexx

- Routines written in other platform-supported native code (compiled) 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.

There are two styles of native code routines supported by Open Object Rexx. Registered External Functions are an older style of routine that is only capable of dealing with String data. These routines do not have access to any of the object-oriented features of the language. The registered external functions are described here, but should be considered only if compatibility with older versions of Object Rexx or other Rexx interpreters is a consideration.

The newer style functions have access to Rexx objects and a fuller set of APIs for interfacing with the interpreter runtime. These functions are the preferred method for writing Open Object Rexx extensions are defined in *Section 8.11, "Building an External Native Library"*.

9.5.1. Registering External Functions

An external function can reside in the same module (executable or library) as an application, or in a separate loadable 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 dynamic-link 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.

9.5.1.1. Creating External Functions

The following is a sample external function definition:

Example 9.10. External functions

```
size_t REXXENTRY SysLoadFuncs(
     const char *Name,                 /* name of the function       */
     size_t     Argc,                  /* number of arguments        */
     CONSTRXSTRING  Argv[],            /* list of argument strings    */
     const char *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 CONSTRXSTRINGs 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 **RexxAllocateMemory(size)** if the returned string is longer than 256 bytes. The Rexx interpreter releases *Retstr* with **RexxFreeMemory(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.

9.5.2. 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.

9.5.2.1. Example

Example 9.11. External functions

```
size_t REXXENTRY SysMkDir(
      const char *Name,                   /* name of the function      */
      size_t      Argc,                   /* number of arguments       */
      CONSTRXSTRING  Argv[],              /* list of argument strings   */
      const char *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 = !CreateDirectory(Argv[0].strptr, NULL);

  sprintf(Retstr->strptr, "%d", rc);    /* result: <> 0 failed         */
                                         /* set proper string length   */
  Retstr->strlength = strlen(Retstr->strptr);
  return 0;                              /* successful completion       */
}
```

9.5.3. External Function Interface Functions

The following sections explain the functions for registering and using external functions.

9.5.3.1. RexxRegisterFunctionDll

RexxRegisterFunctionDll registers an external function that resides in a dynamic-link library routine.

```
retc = RexxRegisterFunctionDll(FuncName, ModuleName, EntryPoint);
```

9.5.3.1.1. Parameters

FuncName (const char *) - input
    is the address of an external function name. The function name must not exceed 1024 characters.

ModuleName (const char *) - input
    is the address of an ASCII library name. *ModuleName* is the library file containing the external
    function routine.

EntryPoint (const char *) - input
    is the address of an ASCII dynamic-link procedure name. *EntryPoint* is the name of the exported
    external function routine within *ModuleName*.

9.5.3.1.2. Return Codes

RXFUNC_OK	0	The call to the function completed successfully.
RXFUNC_NOEMEM	1002	Memory allocation failure, or related.

9.5.3.1.3. Remarks

On Windows, External functions that reside in a dynamic-link library routine must be exported. You
can do this by specifying a module-definition (.DEF) file that lists the external functions in the EXPORT
section. For example:

```
EXPORTS
   SYSMKDIR = SysMkDir
```

A Rexx procedure can register dynamic-link library-external functions with the RXFUNCADD built-in
function. For example:

Example 9.12. RXFUNCADD

```
                          /* register function SysLoadFuncs  */
                          /* in dynamic link library RexxUTIL*/
```

```
Call RxFuncAdd "SysLoadFuncs", "RexxUTIL", "SysLoadFuncs"
Call SysLoadFuncs              /* call to load other functions   */
```

RXFUNCADD registers the external function SysLoadFuncs as routine SysLoadFuncs in the rexxutil library. SysLoadFuncs registers additional functions in rexxutil with RexxRegisterFunctionDll. See the SysLoadFuncs routine below for a function registration example.

9.5.3.1.4. Example

Example 9.13. External functions

```
static const char *RxFncTable[] =     /* function package list     */
{
     "SysCls",
     "SysCurpos",
     "SysCurState",
     "SysDriveInfo",
}


size_t REXXENTRY SysLoadFuncs(
     const char *Name,                 /* name of the function      */
     size_t      Argc,                 /* number of arguments       */
     CONSTRXSTRING  Argv[],            /* list of argument strings  */
     const char *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(const char *);
                                       /* register each function in */
  for (j = 0; j < entries; j++) {      /* the table                 */
    RexxRegisterFunctionDll(RxFncTable[j],
         "RexxUTIL", RxFncTable[j]);
  }
  return 0;                            /* successful completion     */
}
```

9.5.3.2. RexxRegisterFunctionExe

RexxRegisterFunctionExe registers an external function that resides within the application code.

```
retc = RexxRegisterFunctionExe(FuncName, EntryPoint);
```

9.5.3.2.1. Parameters

FuncName (const char *) - input

    is the address of an external function name. The function name must not exceed 1024 characters.

EntryPoint (REXXPFN) - 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.

9.5.3.2.2. 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.

9.5.3.3. RexxDeregisterFunction

RexxDeregisterFunction deregisters an external function.

```
retc = RexxDeregisterFunction(FuncName);
```

9.5.3.3.1. Parameters

FuncName (const char *) - input

    is the address of an external function name to be deregistered.

9.5.3.3.2. Return Codes

RXFUNC_DEFINED	10	The requested function is already registered.
RXFUNC_NOTREG	30	The requested function is not registered.

9.5.3.4. RexxQueryFunction

RexxQueryFunction queries the existence of a registered external function.

```
retc = RexxQueryFunction(FuncName);
```

9.5.3.4.1. Parameters

FuncName (const char *) - input

    is the address of an external function name to be queried.

9.5.3.4.2. Return Codes

RXFUNC_OK	0	The call to the function completed successfully.

| RXFUNC_NOTREG | 30 | The requested function is not registered. |

9.5.3.4.3. 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.

9.6. Registered 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 dynamic-link libraries or within an executable application module.

9.6.1. Writing System Exit Handlers

The following is a sample exit handler definition:

Example 9.14. Rexx_IO_exit

```
int REXXENTRY Rexx_IO_exit(
    int   ExitNumber,    /* code defining the exit function   */
    int   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.

9.6.1.1. 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".

9.6.1.2. 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 **RexxAllocateMemory(size)**. The Rexx interpreter returns the RXSTRING storage for the exit handler.

9.6.1.3. 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.

Example 9.15. RXSYSEXIT

```
typedef struct {
   const char *     sysexit_name;      /* name of exit handler       */
   int              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.

9.6.1.3.1. Example

Example 9.16. RXSYSEXIT

```
...
{
const char *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      */
...
}

int REXXENTRY Init_exit(
     int   ExitNumber,    /* code defining the exit function    */
     int   Subfunction,   /* code defining the exit subfunction */
     PEXIT ParmBlock)     /* function dependent control block   */
{
  char            *user_info[2];       /* saved user information     */
  char            *global_workarea;    /* application data anchor     */
  unsigned short   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       */
}
```

9.6.2. 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

9.6.2.1. RXFNC

Processes calls to external functions.

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:

Example 9.17. RXFUNC 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 ;

    const char *      rxfnc_name;       /* Pointer to function name.  */
    unsigned short    rxfnc_namel;      /* Length of function name.   */
    const char *      rxfnc_que;        /* Current queue name.        */
    unsigned short    rxfnc_quel;       /* Length of queue name.      */
    unsigned short    rxfnc_argc;       /* Number of args in list.    */
    PCONSTRXSTRING    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.

9.6.2.2. RXCMD

Processes calls to subcommand 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:

Example 9.18. RXCMD 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;
    const char *     rxcmd_address;   /* Pointer to address name.    */
    unsigned short   rxcmd_addressl;  /* Length of address name.     */
    const char *     rxcmd_dll;       /* dll name for command.       */
    unsigned short   rxcmd_dll_len;   /* Length of dll name.  0 ==>  */
                                      /* executable file.            */
    CONSTRXSTRING    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.

9.6.2.3. 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:

Example 9.19. RXMSQ 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:

Example 9.20. RXMSQ parameter list

```
typedef struct {
   struct {                           /* Operation flag              */
      unsigned rxfmlifo : 1;          /* Stack entry LIFO when TRUE, */
                                      /* FIFO when FALSE.            */
   } rxmsq_flags;
   CONSTRXSTRING    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:

Example 9.21. RXMSQ parameter list

```
typedef struct {
   size_t           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:

Example 9.22. RXMSQ parameter list

```
typedef struct {
   CONSTRXSTRING      rxmsq_name;       /* RXSTRING containing      */
                                        /* queue name.              */
} RXMSQNAM_PARM;
```

rxmsq_name contains the new queue name.

9.6.2.4. 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:

Example 9.23. RXSIO parameter list

```
typedef struct {
   CONSTRXSTRING      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:

Example 9.24. RXSIO parameter list

```
typedef struct {
    CONSTRXSTRING      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:

Example 9.25. RXSIO parameter list

```
typedef struct {
    RXSTRING           rxsiotrd_retc;    /* RXSTRING for input.        */
} 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:

  Example 9.26. RXSIO parameter list

  ```
  typedef struct {
     RXSTRING          rxsiodtr_retc;    /* RXSTRING for input.       */
  } RXSIODTR_PARM;
  ```

  The input stream line is returned in the *rxsiodtr_retc* RXSTRING.

9.6.2.5. RXHLT

HALT condition processing.

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:

    Example 9.27. RXHLT 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.

    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.

9.6.2.6. 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:

  Example 9.28. RXTRC 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.

9.6.2.7. RXINI

Initialization processing. This exit is called as the last step of Rexx program initialization.

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.

9.6.2.8. RXTER

Termination processing.

The RXTER exit is called as the first step of Rexx program termination.

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.

9.6.3. System Exit Interface Functions

The system exit functions are similar to the subcommand handler functions. The system exit functions are:

9.6.3.1. RexxRegisterExitDll

RexxRegisterExitDll registers an exit handler that resides in a dynamic-link library routine.

```
retc = RexxRegisterExitDll(ExitName, ModuleName, EntryPoint,
                           UserArea, DropAuth);
```

9.6.3.1.1. Parameters

ExitName (const char *) - input
> is the address of an ASCII exit handler name.

ModuleName (const char *) - input
> is the address of an ASCII dynamic-link library name. *ModuleName* is the DLL file containing the exit handler routine.

EntryPoint (const char *) - input

>is the address of an ASCII dynamic-link procedure name. *EntryPoint* is the routine within *ModuleName* that Rexx calls as an exit handler.

UserArea (const char *) - input

>is the address of an area of user-defined information. The user-defined information is a buffer the size of two pointer values. The bytes *UserArea* buffer is saved with the subcommand 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 (size_t) - 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.

9.6.3.1.2. 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 DLL handler with the same name registered in another DLL. (To address this exit handler, you must specify its library name.)
RXEXIT_NOEMEM	1002	There is insufficient memory available to complete this request.

9.6.3.2. RexxRegisterExitExe

RexxRegisterExitExe registers an exit handler that resides within the application code.

```
retc = RexxRegisterExitExe(ExitName, EntryPoint, UserArea);
```

9.6.3.2.1. Parameters

ExitName (const char *) - input

>is the address of an ASCII exit handler name.

EntryPoint (REXXPFN) - input

>is the address of the exit handler entry point within the application executable file.

UserArea (const char *) - input

>is the address of an area of user-defined information. The user-defined information is a buffer the size of two pointer values. The bytes *UserArea* buffer is saved with the subcommand handler

registration. *UserArea* can be null if there is no user information to be saved. The RexxQueryExit function can retrieve the user information.

9.6.3.2.2. 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 DLL handler with the same name registered in another DLL. (To address this exit handler, you must specify its library name.)
RXEXIT_NOTREG	30	Registration was unsuccessful due to duplicate handler and DLL names (RexxRegisterExitExe or RexxRegisterExitDll); the exit handler is not registered (other Rexx exit handler functions).
RXEXIT_NOEMEM	1002	There is insufficient memory available to complete this request.

9.6.3.2.3. 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.

9.6.3.2.4. Example

Example 9.29. SYSEXIT

```
const char     *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       */
```

9.6.3.3. RexxDeregisterExit

RexxDeregisterExit deregisters an exit handler.

```
retc = RexxDeregisterExit(ExitName, ModuleName);
```

9.6.3.3.1. Parameters

ExitName (const char *) - input
>   is the address of an ASCII exit handler name.

ModuleName (const char *) - input
>   is the address of an ASCII dynamic-link 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.

9.6.3.3.2. Return Codes

RXEXIT_OK	0	The system exit function executed successfully.
RXEXIT_NOTREG	30	Registration was unsuccessful due to duplicate handler and DLL 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."

9.6.3.3.3. Remarks

The handler is removed from the exit handler list.

9.6.3.4. RexxQueryExit

RexxQueryExit queries an exit handler and retrieves saved user information.

```
retc = RexxQueryExit(ExitName, ModuleName, Flag, UserWord);
```

9.6.3.4.1. Parameters

ExitName (const char *) - input
>   is the address of an ASCII exit handler name.

ModuleName (const char *) - input
>   restricts the query to an exit handler within the *ModuleName* dynamic-link 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 (unsigned short *) - 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 (char *) - output
>   is the address of an area to receive the user information saved with RexxRegisterExitExe or RexxRegisterExitDll. The referenced area must be large enough to store two pointer values. *UserWord* can be null if the saved user information is not required.

9.6.3.4.2. Return Codes

RXEXIT_OK	0	The system exit function executed successfully.
RXEXIT_NOTREG	30	Registration was unsuccessful due to duplicate handler and DLL names (RexxRegisterExitExe or RexxRegisterExitDll); the exit handler is not registered (other Rexx exit handler functions).

9.6.3.4.3. Example

Example 9.30. Command handler

```
int REXXENTRY Edit_IO_Exit(
  int       Code,       /* Major exit code                    */
  int       SubCode     /* Minor exit code                    */
  PEXIT     Parms)      /* Exit-specific parameters           */
{
  char            *user_info[2];      /* saved user information    */
  char            *global_workarea;   /* application data anchor   */
  unsigned short  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      */
...
}
```

9.7. Variable Pool Interface

Application programs can use the Rexx Variable Pool Interface to manipulate the variables of a currently active Rexx procedure.

9.7.1. Interface Types

Three of the Variable Pool Interface functions (set, fetch, and drop) have dual interfaces.

9.7.1.1. 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.

9.7.1.2. 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.

9.7.2. RexxVariablePool Restrictions

The RexxVariablePool interface is only available from subcommand handlers, external functions, and exit handlers. The interface will access the variable context that initiated the call to the handler code and is only available if made from the same thread.

9.7.3. RexxVariablePool Interface Function

Rexx procedure variables are accessed using the RexxVariablePool function.

9.7.3.1. RexxVariablePool

RexxVariablePool accesses variables of a currently active Rexx procedure.

```
retc = RexxVariablePool(RequestBlockList);
```

9.7.3.1.1. 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:

Example 9.31. SHVBLOCK

```
typedef struct shvnode {
    struct shvnode    *shvnext;
    CONSTRXSTRING     shvname;
    RXSTRING          shvvalue;
    size_t            shvnamelen;
    size_t            shvvaluelen;
    unsigned char     shvcode;
    unsigned char     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
      RexxFreeMemory(ptr).

      > **Note**
      >
      > The RexxVariablePool does not add a terminating null character to the variable name.

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_FETCH, RXSHV_SYFET , RXSHV_PRIV ,RXSHV_NEXT
      shvvalue is a buffer that is used by the Rexx interpreter to return the value 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 `RexxFreeMemory(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:

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*.

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.

9.7.3.1.2. 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.

9.7.3.1.3. Example

Example 9.32. RexxVariablePool

```
/********************************************************************/
/*                                                                  */
/* SetRexxVariable - Set the value of a Rexx variable               */
/*                                                                  */
/********************************************************************/

int SetRexxVariable(
  const char *name,                   /* Rexx variable to set      */
  char      *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           */
}
```

9.8. Dynamically Allocating and De-allocating Memory

For several functions of the Rexx-API it is necessary or possible to dynamically allocate or free memory. Depending on the operating system, compiler and REXX interpreter, the method for these allocations and de- allocations vary. To write system independent code, Object REXX comes with two API function calls called RexxAllocateMemory() and RexxFreeMemory(). These functions are wrapper for the corresponding compiler or operating system memory functions.

9.8.1. The RexxAllocateMemory() Function

```
void * REXXENTRY RexxAllocateMemory( size_t size );
```

where:
size
    is the number of bytes of requested memory.

Return Codes

Returns a pointer to the newly allocated block of memory, or NULL if no memory could be allocated.

9.8.2. The RexxFreeMemory() Function

```
RexxReturnCode REXXENTRY RexxFreeMemory( void *MemoryBlock );
```

where:
MemoryBlock
> is a void pointer to the block of memory allocated by the Object REXX interpreter, or allocated by a previous call to RexxAllocateMemory().

Return Codes

RexxFreeMemory() always returns 0.

9.9. 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 Windows InterProcess Communications queues.

9.9.1. Queue Interface Functions

The following sections explain the functions for creating and using named queues.

9.9.1.1. RexxCreateQueue

RexxCreateQueue creates a new (empty) queue.

```
retc = RexxCreateQueue(Buffer, BuffLen, RequestedName, DupFlag);
```

9.9.1.1.1. Parameters

Buffer (char *) - input
> is the address of the buffer where the ASCII name of the created queue is returned.

BuffLen (size_t) - input
> is the size of the buffer.

RequestedName (const char *) - 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 (size_t *) - output
    is the duplicate name indicator. This flag is set when the requested name already exists.

9.9.1.1.2. 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.

9.9.1.1.3. Remarks

Queue names must conform to the same syntax rules as Rexx variable names. Lowercase characters in queue names are translated to uppercase.

9.9.1.2. RexxOpenQueue

RexxOpenQueue creates a new (empty) queue if a queue by the given name does not already exist. In contrast to RexxCreateQueue, RexxOpenQueue will not create a differently named queue if the indicated queue name already exists.

```
retc = RexxOpenQueue(RequestedName, CreatedFlag);
```

9.9.1.2.1. Parameters

RequestedName (const char *) - input
    is the address of an ASCII queue name. If no queue of that name exists, a queue is created with the requested name. and the CreatedFlag will be set to TRUE. If the name already exists, this will just return a successful return code. The maximum length for a queue name is 1024 characters.

CreateFlag (size_t *) - output
    indicates whether RexxOpenQueue created the indicated queue. If zero on return, then the named queue already existed.

9.9.1.2.2. 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.

9.9.1.2.3. Remarks

Queue names must conform to the same syntax rules as Rexx variable names. Lowercase characters in queue names are translated to uppercase.

9.9.1.3. RexxDeleteQueue

RexxDeleteQueue deletes a queue.

```
retc = RexxDeleteQueue(QueueName);
```

9.9.1.3.1. Parameters

QueueName (const char *) - input
    is the address of the ASCII name of the queue to be deleted.

9.9.1.3.2. Return Codes

RXQUEUE_OK	0	The system queue function completed successfully.
RXQUEUE_BADQNAME	5	The queue name is not valid, or you tried to create or delete a queue named SESSION.
RXQUEUE_NOTREG	9	The queue does not exist.
RXQUEUE_ACCESS	10	The queue cannot be deleted because it is busy.

9.9.1.3.3. Remarks

If a queue is busy (for example, wait is active), it is not deleted.

9.9.1.4. RexxQueueExists

RexxQueueExists tests if name queue exists.

```
retc = RexxQueueExists(QueueName);
```

9.9.1.4.1. Parameters

QueueName (const char *) - input
    is the address of the ASCII name of the queue to be queried.

9.9.1.4.2. Return Codes

RXQUEUE_OK	0	The system queue function completed successfully.
RXQUEUE_BADQNAME	5	The queue name is not valid, or you tried to create or delete a queue named SESSION.
RXQUEUE_NOTREG	9	The queue does not exist.

9.9.1.5. RexxQueryQueue

RexxQueryQueue returns the number of entries remaining in the named queue.

```
retc = RexxQueryQueue(QueueName, Count);
```

9.9.1.5.1. Parameters

QueueName (const char *) - input
    is the address of the ASCII name of the queue to be queried.

Count (size_t *) - output
    is the number of entries in the queue.

9.9.1.5.2. Return Codes

RXQUEUE_OK	0	The system queue function completed successfully.
RXQUEUE_BADQNAME	5	The queue name is not valid, or you tried to create or delete a queue named SESSION.
RXQUEUE_NOTREG	9	The queue does not exist.

9.9.1.6. RexxAddQueue

RexxAddQueue adds an entry to a queue.

```
retc = RexxAddQueue(QueueName, EntryData, AddFlag);
```

9.9.1.6.1. Parameters

QueueName (const char *) - input
    is the address of the ASCII name of the queue to which data is to be added.

EntryData (PCONSTRXSTRING) - input
    is the address of a CONSTRXSTRING containing the data to be added to the queue.

AddFlag (size_t) - 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).

9.9.1.6.2. Return Codes

RXQUEUE_OK	0	The system queue function completed successfully.
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_NOTREG	9	The queue does not exist.
RXQUEUE_MEMFAIL	12	There is insufficient memory available to complete the request.

9.9.1.7. RexxPullFromQueue

RexxPullFromQueue removes the top entry from the queue and returns it to the caller.

```
retc = RexxPullFromQueue(QueueName, DataBuf, DateTime, WaitFlag);
```

9.9.1.7.1. Parameters

QueueName (const char *) - 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 (REXXDATETIME *) - output
   is the address of the entry's date and time stamp. If the date and time stamp is not needed,
   DateTime may be NULL.

WaitFlag (size_t) - 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.

9.9.1.7.2. Return Codes

RXQUEUE_OK	0	The system queue function completed successfully.
RXQUEUE_BADQNAME	5	The queue name is not valid, or you tried to create or delete a queue named SESSION.
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_MEMFAIL	12	There is insufficient memory available to complete the request.

9.9.1.8. RexxClearQueue

RexxClearQueue clears all entries from a named queue.

```
retc = RexxClearQueue(QueueName);
```

9.9.1.8.1. Parameters

QueueName (const char *) - input
    is the address of the ASCII name of the queue to be cleared.

9.9.1.8.2. Return Codes

RXQUEUE_OK	0	The system queue function completed successfully.
RXQUEUE_BADQNAME	5	The queue name is not valid, or you tried to create or delete a queue named SESSION.
RXQUEUE_NOTREG	9	The queue does not exist.

9.9.1.9. RexxPullQueue (Deprecated)

RexxPullQueue removes the top entry from the queue and returns it to the caller. RexxPullQueue is deprecated in favor of its more portable replacement RexxPullFromQueue (*Section 9.9.1.7, "RexxPullFromQueue"*).

```
retc = RexxPullQueue(QueueName, DataBuf, DateTime, WaitFlag);
```

9.9.1.9.1. Parameters

QueueName (const char *) - 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 (size_t) - 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.

9.9.1.9.2. Return Codes

RXQUEUE_OK	0	The system queue function completed successfully.
RXQUEUE_BADQNAME	5	The queue name is not valid, or you tried to create or delete a queue named SESSION.
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_MEMFAIL	12	There is insufficient memory available to complete the request.

9.9.1.9.3. Remarks

The caller is responsible for freeing the returned memory that DataBuf points to.

9.10. 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.

9.10.1. Halt and Trace Interface Functions

The Halt and Trace functions are:

9.10.1.1. RexxSetHalt

RexxSetHalt raises a HALT condition in a running Rexx program.

```
retc = RexxSetHalt(ProcessId, ThreadId);
```

9.10.1.1.1. Parameters

ProcessId (process_id_t) - input
    is the process ID of the target Rexx procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId (thread_id_t) - 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.

9.10.1.1.2. 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.

9.10.1.1.3. Remarks

This call is not processed if the target Rexx program is running with the RXHLT exit enabled.

9.10.1.2. RexxSetTrace

RexxSetTrace turns on the interactive debug mode for a Rexx procedure.

```
retc = RexxSetTrace(ProcessId, ThreadId);
```

9.10.1.2.1. Parameters

ProcessId (process_id_t) - input
    is the process ID of the target Rexx procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId (thread_id_t) - 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.

9.10.1.2.2. 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.

9.10.1.2.3. Remarks
A RexxSetTrace call is not processed if the Rexx procedure is using the RXTRC exit.

9.10.1.3. RexxResetTrace

RexxResetTrace turns off the interactive debug mode for a Rexx procedure.

```
retc = RexxResetTrace(ProcessId,ThreadId);
```

9.10.1.3.1. Parameters

ProcessId (process_id_t) - input
    is the process ID of the target Rexx procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId (thread_id_t) - 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.

9.10.1.3.2. 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.

9.10.1.3.3. 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.

9.11. 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.

> **Note**
>
> The **TRACE** keyword instruction cannot be used in the Rexx macrospace. Since macrospace uses the tokenized format, it is not possible to get the source code from macrospace to trace a function.

9.11.1. 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.

9.11.2. Storage of Macrospace Libraries

The Rexx macrospace is stored in separate process using a daemon process. Macrospace routines are retrieved using interprocess call (IPC) mechanisms. A package file that is loaded in the local process might be preferable to loading routines in the macrospace.

9.11.3. Macrospace Interface Functions

The functions to manipulate macrospaces are:

9.11.3.1. RexxAddMacro

RexxAddMacro loads a Rexx procedure into the macrospace.

```
retc = RexxAddMacro(FuncName, SourceFile, Position);
```

9.11.3.1.1. Parameters

FuncName (const char *) - input
  is the address of the ASCII function name. Rexx procedures in the macrospace are called using the assigned function name.

SourceFile (const char *) - 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 (size_t) - 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.

9.11.3.1.2. Return Codes

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_SOURCE_NOT_FOUND	7	The requested file was not found.
RXMACRO_INVALID_POSITION	8	An invalid search-order position request flag was used.

9.11.3.2. RexxDropMacro

RexxDropMacro removes a Rexx procedure from the macrospace.

```
retc = RexxDropMacro(FuncName);
```

9.11.3.2.1. Parameter

FuncName (const char *) - input
    is the address of the ASCII function name.

9.11.3.2.2. Return Codes

RXMACRO_OK	0	The call to the function completed successfully.
RXMACRO_NOT_FOUND	2	The requested function was not found in the macrospace.

9.11.3.3. RexxClearMacroSpace

RexxClearMacroSpace removes all loaded Rexx procedures from the macrospace.

```
retc = RexxClearMacroSpace();
```

9.11.3.3.1. Return Codes

RXMACRO_OK	0	The call to the function completed successfully.
RXMACRO_NOT_FOUND	2	The requested function was not found in the macrospace.

9.11.3.3.2. Remarks

RexxClearMacroSpace must be used with care. This function removes all functions from the macrospace, including functions loaded by other processes.

9.11.3.4. RexxSaveMacroSpace

RexxSaveMacroSpace saves all or part of the macrospace Rexx procedures to a disk file.

```
retc = RexxSaveMacroSpace(FuncCount, FuncNames, MacroLibFile);
```

9.11.3.4.1. Parameters

FuncCount (size_t) - input
    Number of Rexx procedures to be saved.

FuncNames (const char **) - input
    is the address of a list of ASCII function names. *FuncCount* gives the size of the function list.

MacroLibFile (const char *) - input
    is the address of the ASCII macrospace file name. If *MacroLibFile* already exists, it is replaced with the new file.

9.11.3.4.2. Return Codes

RXMACRO_OK	0	The call to the function completed successfully.
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_FILE_ERROR	5	An error occurred accessing a macrospace file.

9.11.3.4.3. 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.

9.11.3.5. RexxLoadMacroSpace

RexxLoadMacroSpace loads all or part of the Rexx procedures from a saved macrospace file.

```
retc = RexxLoadMacroSpace(FuncCount, FuncNames, MacroLibFile);
```

9.11.3.5.1. Parameters

FuncCount (size_t) - input
    is the number of Rexx procedures to load from the saved macrospace.

FuncNames (const char **) - input
    is the address of a list of Rexx function names. *FuncCount* gives the size of the function list.

MacroLibFile (const char *) - input
    is the address of the saved macrospace file name.

9.11.3.5.2. Return Codes

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_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.

9.11.3.5.3. 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.

9.11.3.6. RexxQueryMacro

RexxQueryMacro searches the macrospace for a specified function.

```
retc = RexxQueryMacro(FuncName, Position)
```

9.11.3.6.1. Parameters

FuncName (const char *) - input
    is the address of an ASCII function name.

Position (unsigned short *) - 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.

9.11.3.6.2. Return Codes

RXMACRO_OK	0	The call to the function completed successfully.
RXMACRO_NOT_FOUND	2	The requested function was not found in the macrospace.

9.11.3.7. RexxReorderMacro

RexxReorderMacro changes the search order position of a loaded macrospace function.

```
retc = RexxReorderMacro(FuncName, Position)
```

9.11.3.7.1. Parameters

FuncName (const char *) - 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.

9.11.3.7.2. Return Codes

RXMACRO_OK	0	The call to the function completed successfully.
RXMACRO_NOT_FOUND	2	The requested function was not found in the macrospace.
RXMACRO_INVALID_POSITION	8	An invalid search-order position request flag was used.

9.12. Windows Scripting Host Interface

The purpose of this section is to describe any behaviors specific to Object Rexx that the designer of a Windows Scripting Host (WSH) should be aware of. It is assumed that the reader is already familiar with how to do that, or has the appropriate documentation at hand. For further information, see "Windows Scripting Host Engine", in *Open Object Rexx: Reference*.

9.12.1. Concurrency

Object Rexx is a multithreaded program. (See Concurrency, in *Object Rexx for Windows: Reference*.) The closest Windows model is the free-threaded model for dealing with multiple threads. The WSH controls are typically apartment-threaded. Therefore, since Object Rexx does not restrict its callers to any particular thread, and it passes on any exterior calls in the thread context in which it was received, then for all practical purposes it should be treated as an apartment-threaded program.

9.12.2. WSH Features

9.12.2.1. COM Interfaces

There are several interfaces that a WSH engine can use. Object Rexx does not support all of them; some are supported, but created dynamically. Since the dynamically-supported interfaces will not appear in an OLE viewer, they are listed here.

The following interfaces are fully supported:

- IUnknown

- IActiveScriptParse

- IActiveScriptError

- IActiveScriptParseProcedure

- IObjectSafety

While Object Rexx has code for all of the methods of an interface that it supports, all methods may not be implemented. The methods that are not implemented will return E_NOTIMPL.

The following interfaces are supported:

- IDispatch
    - GetIDsOfNames

    - Invoke

- IDispatchEx
    - GetDispID - but does not support dynamic creation of properties or methods.

    - InvokeEx - but does not support dynamic creation of properties or methods.

    - GetMemberName

    - GetNextDispID

- IActiveScript
    - SetScriptSite

    - GetScriptState

    - SetScriptState

    - Close

    - AddNamedItem

    - AddTypeLib

    - GetScriptDispatch

9.12.2.2. Script Debugging

Object Rexx does not support the WSH Script Debugging facilities. For the best techniques for debugging an Object Rexx script, refer to the section on the Trace keyword in "Keyword Instructions", in *Open Object Rexx: Reference*.

9.12.2.3. DCOM

Object Rexx does not support DCOM.

Appendix A. Distributing Programs without Source

Open Object Rexx comes with a utility called RexxC. You can use this utility to produce versions of your programs that do not include the original program source. You can use these programs to replace any Rexx program file that includes the source, with the following restrictions:

1. The SOURCELINE built-in function returns **0** for the number of lines in the program and raises an error for all attempts to retrieve a line.

2. A sourceless program may not be traced. The TRACE instruction runs without error, but no tracing of instruction lines, expression results, or intermediate expression values occurs.

The syntax of the REXXC utility is:

```
>>-RexxC--inputfile--+------------+--+------+----------------><
                     +-outputfile-+  +- -s -+
```

If you specify the *outputfile*, the language processor processes the *inputfile* and writes the executable version of the program to the *outputfile*. If the *outputfile* already exists, it is replaced.

If the language processor detects a syntax error while processing the program, it reports the error and stops processing without creating a new output file. If you omit the *outputfile*, the language processor performs a syntax check on the program without writing the executable version to a file.

You can use the *s* option (*/s* on Windows and *-s* on unixes) to suppress the display of the information about the interpreter used.

> **Note**
>
> You can use the in-storage capabilities of the RexxStart programming interface to process the file image of the output file.

Due to changes in the 4.0 interpreter it is not possible for Rexx programs, tokenized with any previous version of ooRexx, to run under ooRexx 4.0.0. The programs will need to be re-tokenized with the 4.0.0 interpreter. The changes are such that the old tokenized form can not be changed to the new form. Therefore, no utility such as the old **RxMigrate** can be provided. Part of the purpose of the changes is to make this sitaution less likely to occur in the future.

Appendix B. Sample Rexx Programs

Rexx supplies the following sample programs as .REX files.

CCREPLY.REX
> A concurrent programming example.
>
> This program demonstrates how to use reply to run two methods at the same time.

COMPLEX.REX
> 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.

DESKICON.REX
> A WindowsProgramManager class example.
>
> This sample uses the method AddDesktopIcon of the WindowsProgramManager class to create a shortcut to a program or an application on the Windows desktop.

DESKTOP.REX
> This program demonstrates how you could use the WindowsProgramManager class to manipulate program groups and program items.

DRIVES.REX
> A sample use of the Sys... functions.
>
> This program displays information about drives using the utility functions SysDriveMap, SysDriveInfo, SysFileSystemType, and SysBootDrive.

EVENTLOG.REX
> A sample use of the WindowsEventLog class.
>
> This sample demonstrates how to read from and write to the Windows event log using the methods of the WindowsEventLog class.

FACTOR.REX
> A factorial program.
>
> This program demonstrates a way to define a factorial class using the subclass method and the .methods environment symbol.

GREPLY.REX
> 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.REX
> 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.REX

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.REX

An example that displays the days of the month January 1994.

This version demonstrates the use of arrays to replace stems.

OLE\APPS\SAMP01.REX

Starts Internet Explorer and shows the RexxLA homepage. After 10 seconds the RexxLA news page is displayed.

OLE\APPS\SAMP02.REX

Shows some features of the Windows Scripting Host Shell Object:
• Query environment string

• List special folders

• Create a shortcut on the desktop

OLE\APPS\SAMP03.REX

Shows some features of the Windows Scripting Host Network object:
• Query computer name, user name

• List network connections for drives and printers

OLE\APPS\SAMP04.REX

Creates a mail message in Lotus Notes® and sends it to a number of recipients automatically.

OLE\APPS\SAMP05.REX

Creates a new document in WordPro 97, enters some text with different attributes, and finally saves and prints the document.

OLE\APPS\SAMP06.REX

Creates a new document in WordPro 97 with a provided Smartmaster. Fills in some "Click here" fields with data prompted by the program or queried from the system. Finally the document is saved to the directory in which this Rexx program is located and is sent to the printer.

OLE\APPS\SAMP07.REX

Creates a new spreadsheet in Lotus 1-2-3® and fills in a table with fictional revenue numbers. The table also contains a calculated field and different styles. A second sheet is added with a 3D chart displaying the revenue data.

OLE\APPS\SAMP08.REX

Creates a Microsoft Word document, enters some text, and saves it. The program then loads the document again and modifies it.

OLE\APPS\SAMP09.REX

Creates a Microsoft Excel sheet, enters some data, and saves it.

OLE\APPS\SAMP10.REX

   Uses the Windows Script Host FileSystemObject to obtain information about the drives of the
   system.

OLE\APPS\SAMP11.REX

   Gets the stock price from the RexxLA internet page with Microsoft Internet Explorer, and stores it
   in a Rexx variable.

OLE\APPS\SAMP12.REX

   Demonstrates the use of events with Microsoft Internet Explorer:
   • Navigate to the RexxLA homepage and disallow the changing of the URL to a page not in that
     "address space".

OLE\APPS\SAMP13.REX

   Demonstrates the use of events with Microsoft Internet Explorer:
   • Search for the string "Rexx" on the RexxLA Web page, and go randomly to one of the found
     sites.

OLE\OLEINFO\OLEINFO.REX

   This application is a "small" browser for OLE objects.

OLE\ADSI\ADSI1.REX

   Retrieves information about a computer with ADSI.

OLE\ADSI\ADSI2.REX

   Gets a user's full name and changes it.

OLE\ADSI\ADSI3.REX

   Shows the use of ADSI containers.

OLE\ADSI\ADSI4.REX

   Shows the use of filters with ADSI collections.

OLE\ADSI\ADSI5.REX

   Displays namespaces and domains.

OLE\ADSI\ADSI6.REX

   Enables you to inspect the properties of an object.

OLE\ADSI\ADSI7.REX

   Creates a group, and places several users in it.

OLE\ADSI\ADSI8.REX

   Removes the users and the group that were created in sample ADSI7.REX.

OLE\METHINFO\MAIN.REX

   This application demonstrates the use of the GetKnownMethods method.

OLE\WMI\ACCOUNTS.REX

   This is a demo application for displaying all the accounts of the windows system with WMI. It also
   shows how to display all the properties of a WMI object in general.

OLE\WMI\SERVICES.REX

   This application demonstrates how to list, start, stop, pause, or resume windows services with
   WMI.

OLE\WMI\PROCESS.REX

This application displays by means of WMI the processes of a windows system that are running.

OLE\WMI\OSINFO.REX

This sample script uses a Windows Management Instrumentation (WMI) object ("Win32_OperatingSystem") to obtain information about the installed operating system(s).

OLE\WMI\SYSINFO\SYSINFO.REX

This is a demo application for inspecting some system properties using WMI.

PHILFORK.REX

Sample for concurrency with command line output.

PIPE.REX

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.REX

An example that types or pushes today's date and moon phase, in English date format.

QTIME.REX

An example that lays or stacks time in English time format, and also chimes.

REGISTRY.REX

This program demonstrates how you could use the WindowsRegistry class to work with the Windows registry.

SEMCLS.REX

An Object Rexx semaphore class.

This file implements a semaphore class in Object Rexx.

STACK.REX

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.REX

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.REX

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

Any reference to a non-open source product, program, or service is not intended to state or imply that only non-open source product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Rexx Language Association (RexxLA) intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-open source product, program, or service.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-open source products was obtained from the suppliers of those products, their published announcements or other publicly available sources. RexxLA has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-RexxLA packages. Questions on the capabilities of non-RexxLA packages should be addressed to the suppliers of those products.

All statements regarding RexxLA's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

C.1. Trademarks

Open Object Rexx™ and ooRexx™ are trademarks of the Rexx Language Association.

The following terms are trademarks of the IBM Corporation in the United States, other countries, or both:

1-2-3
AIX
IBM
Lotus
OS/2
S/390
VisualAge

AMD is a trademark of Advance Micro Devices, Inc.

Intel, Intel Inside (logos), MMX and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the Unites States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

C.2. Source Code For This Document

The source code for this document is available under the terms of the Common Public License v1.0 which accompanies this distribution and is available in the appendix *Appendix D, Common Public License Version 1.0*. The source code itself is available at `http://sourceforge.net/project/showfiles.php?group_id=119701`.

The source code for this document is maintained in DocBook SGML/XML format.

Appendix D. Common Public License Version 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

D.1. Definitions

"Contribution" means:

1. in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and

2. in the case of each subsequent Contributor:
   a. changes to the Program, and

   b. additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

D.2. Grant of Rights

1. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

2. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

3. Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement

of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

4. Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

D.3. Requirements

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

1. it complies with the terms and conditions of this Agreement; and

2. its license agreement:

    a. effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

    b. effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

    c. states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

    d. states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

1. it must be made available under this Agreement; and

2. a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

D.4. Commercial Distribution

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified

Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

D.5. No Warranty

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

D.6. Disclaimer of Liability

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

D.7. General

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed. In addition, if Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable.

However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. IBM is the initial Agreement Steward. IBM may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

Appendix E. Revision History

Revision 0-0   Tue Aug 7 2012   David Ashley

   Initial creation of book by publican

Index

Symbols

A

B

Index