ooRexx Documentation 5.0.0

Open Object Rexx

Application Programming Interfaces

ooRexx Documentation 5.0.0 Open Object Rexx Application Programming Interfaces
Edition 2022.12.22

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

Preface

This book describes how to interface applications to Open Object Rexx or extend the Rexx language by using Rexx C++ or classic application programming interfaces (APIs). As used here, the term application refers to programs written in languages other than Rexx, usually in the C or C++ language.

1. Document Conventions

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

1.1. Typographic Conventions

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` is used to highlight literal strings, class names, or inline code examples. For example:

> The **`Class`** class comparison methods return **`.true`** or **`.false`**, the result of performing the comparison operation.

> This method is exactly equivalent to **`subWord(`**_n_**`, 1)`**.

`Mono-spaced Normal` denotes method names or source code in program listings set off as separate examples.

> This method has no effect on the action of any `hasEntry`, `hasIndex`, `items`, `remove`, or `supplier` message sent to the collection.

```
-- reverse an array
a = .Array~of("one", "two", "three", "four", "five")

-- five, four, three, two, one
aReverse = .CircularQueue~new(a~size)~appendAll(a)~makeArray("lifo")
```

Proportional Italic is used for method and function variables and arguments.

> A supplier loop specifies one or two control variables, _index_, and _item_, which receive a different value on each repetition of the loop.

> Returns a string of length _length_ with _string_ centered in it and with _pad_ characters added as necessary to make up length.

1.2. 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.
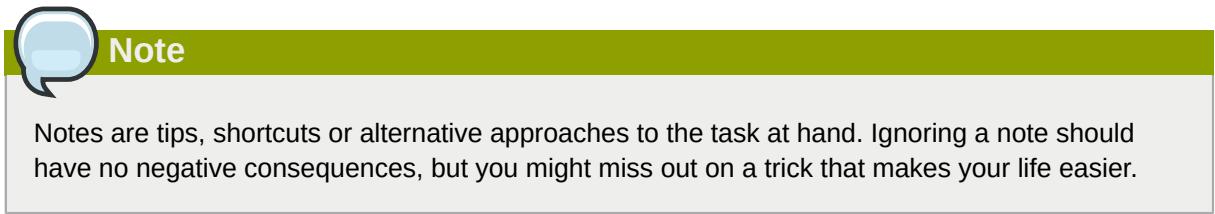
> **Important**
>
> Important boxes detail things that are easily missed, like mandatory initialization. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

> **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.

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

  ▶▶─(STATEMENT)─── *required_item* ───▶◀

- Optional items appear below the main path.

  ▶▶─(STATEMENT)──────────▶◀
  　　　　　　　└─ *optional_item* ─┘
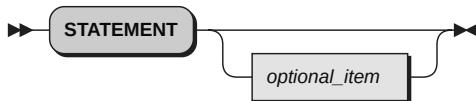
- If you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path.

  ▶▶─(STATEMENT)──┬─ *required_choice1* ─┬──▶◀
  　　　　　　　　　　└─ *required_choice2* ─┘
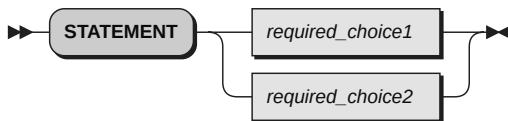
- If choosing one of the items is optional, the entire stack appears below the main path.

  ▶▶─(STATEMENT)──────────────────▶◀
  　　　　　　　　　├─ *optional_choice1* ─┤
  　　　　　　　　　└─ *optional_choice2* ─┘
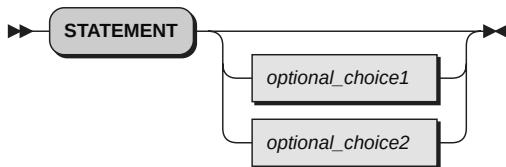
- If one of the items is the default, it is usually the topmost item of the stack of items below the main path.
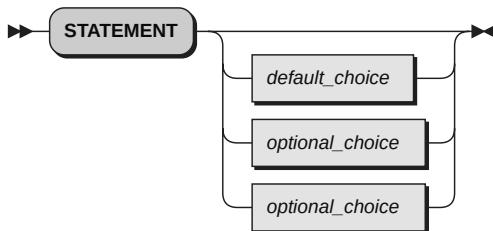
- A path returning to the left above the main line indicates an item that can be repeated.
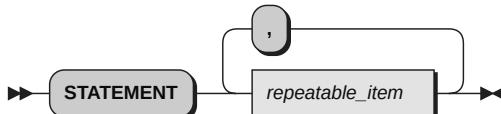
A repeat path above a stack indicates that you can repeat the items in the stack.

- A pointed rectangle 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.

- Keywords appear in uppercase (for example, **SIGNAL**). 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, *index*). 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:
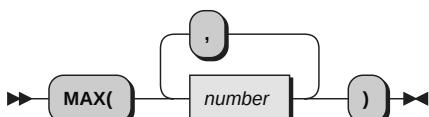
3. Getting Help and Submitting Feedback

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

Open Object Rexx utilizes SourceForge to house its source repositories, mailing lists and other project features at *https://sourceforge.net/projects/oorexx*. ooRexx uses the Developer and User mailing lists at *https://sourceforge.net/p/oorexx/mailman* for discussions concerning ooRexx. The ooRexx user is most likely to get timely replies from one of these mailing lists.

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

The Developer Mailing List
 Subscribe to the oorexx-devel mailing list at *https://lists.sourceforge.net/lists/listinfo/oorexx-devel* to discuss ooRexx project development activities and future interpreter enhancements. You can find its archive of past messages at *http://sourceforge.net/mailarchive/forum.php?forum_name=oorexx-devel*.

The Users Mailing List

Subscribe to the oorexx-users mailing list at *https://lists.sourceforge.net/lists/listinfo/oorexx-users* to discuss how to use ooRexx. It also supports a historical archive of past messages.

The Announcements Mailing List

Subscribe to the oorexx-announce mailing list at *https://lists.sourceforge.net/lists/listinfo/oorexx-announce* to receive announcments of significant ooRexx project events.

The Bug Mailing List

Subscribe to the oorexx-bugs mailing list at *https://lists.sourceforge.net/lists/listinfo/oorexx-bugs* to monitor changes in the ooRexx bug tracking system.

Bug Reports

You can view ooRexx bug reports at *https://sourceforge.net/p/oorexx/bugs*. To be able to create new bug reports, you will need to first register for a SourceForge userid at *https://sourceforge.net/user/registration*. When reporting a bug, please try to provide as much information as possible to help developers determine the cause of the issue. Sample program code 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 *https://sourceforge.net/p/oorexx/documentation*. 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 new suggest ooRexx features or enhancements at *https://sourceforge.net/p/oorexx/feature-requests*.

Patch Reports

If you create an enhancement patch for ooRexx please post the patch at *https://sourceforge.net/p/oorexx/patches*. 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 at *https://sourceforge.net/p/oorexx/bugs* 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 at *https://sourceforge.net/p/oorexx/discussion*. 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 maintains a forum at *http://www.rexxla.org/forum.html*.

3.3. comp.lang.rexx Newsgroup

The comp.lang.rexx newsgroup at *https://groups.google.com/forum/#!forum/comp.lang.rexx* is a good place to obtain help from many individuals within the Rexx community. You can obtain help on Open Object Rexx and other Rexx interpreters and tools.

4. Related Information

See also: *Open Object Rexx: Reference*

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.

1.1. 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 2.3.3, "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* API:

Example 1.1. 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 1.2. 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* 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.

1.1.1. RexxCreateInterpreter

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

Example 1.3. 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 1.1.2, "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.

1.1.2. 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 1.4. API — RexxOption INITIAL_ADDRESS_ENVIRONMENT

```
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* API. The application data pointer allows methods, functions, exits, and command handlers to recover access to globally defined application data.

Example 1.5. API — RexxOption APPLICATION_DATA

```
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 1.6. API — RexxOption EXTERNAL_CALL_PATH

```
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 1.7. API — RexxOption EXTERNAL_CALL_EXTENSIONS

```
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 ::REQUIRES LIBRARY. Multiple libraries can be loaded by specifying this option multiple times.

Example 1.8. API — RexxOption LOAD_REQUIRED_LIBRARY

```
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 registered 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 1.12, "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 ::REQUIRES LIBRARY. Multiple libraries can be registered by specifying this option multiple times.

Example 1.9. API — RexxOption REGISTER_LIBRARY

```
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 1.15, "Rexx Exits Interface"* for details.

Example 1.10. API — RexxOption DIRECT_EXITS

```
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 direct 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 entry point of NULL or a handler name of NULL. The direct environment handlers are called using the calling convention described in *Section 1.16, "Command Handler Interface"*.

Example 1.11. API — RexxOption DIRECT_ENVIRONMENTS

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


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

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

REDIRECTING_ENVIRONMENTS

Registers one or more redirecting subcommand handler environments with the interpreter instance. The handlers are a list of RexxRedirectingEnvironment structs. Each enabled handler is specified in a single RexxRedirectingEnvironment struct identifying the handler name and entry point. The list is terminated by an instance using a handler entry point of NULL or a handler name of NULL. The redirecting environment handlers are called using the calling convention described in *Section 1.16, "Command Handler Interface"*.

Example 1.12. API — RexxOption REDIRECTING_ENVIRONMENTS

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


environments[0].handler = redirectingCommandHandler;
environments[0].name = "SYSTEM";
environments[1].handler = NULL;
environments[1].name = NULL;

options[0].optionName = REDIRECTING_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 2.6, "Registered System Exit Interface"* for details.

Example 1.13. API — RexxOption REGISTERED_EXITS

```
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 2.4, "Subcommand Interface"*.

Example 1.14. API — RexxOption REGISTERED_ENVIRONMENTS

```
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;
```

1.2. 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.

1.2.1. *CHG* 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-dimensional 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.

RexxStringTableObject
    An instance of Rexx StringTable class. Like for directories, there are APIs provided for access and manipulating data stored in a StringTable.

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.

1.2.2. 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.
positive_wholenumber_t	very similar to above wholenumber_t, but with the added restriction that the value must be equal to or larger than one. For 32-bit versions, this gives a range of 999,999,999 to 1. On 64-bit systems, the range is 999,999,999,999,999,999 to 1.
nonnegative_wholenumber_t	very similar to above wholenumber_t, but with the added restriction that the value must be equal to or larger than zero. For 32-bit versions, this gives a range of 999,999,999 to 0. On 64-bit systems, the range is 999,999,999,999,999,999 to 0.
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.

1.3. 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 1.15. 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 to 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 1.16. 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 1.17. 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*, *RexxMethodContext*, and *RexxExitContext* 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.

1.4. Threading Considerations

When using *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* 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* 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.

1.5. 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 *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, *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* on an object reference you no longer require will remove the local lock, and thus limit the overhead associated with tracking the object references.

1.6. 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* and *Terminate*. AttachThread() allows additional externally identified threads to be included in the interpreter instance threadpool. AttachThread returns a *RexxThreadContext* 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.

1.7. Rexx Thread Context Interface

The RexxThreadContext interface vector provides a very wide range of functions to your application code. There are more than 170 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* 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* call that created the interpreter instance. The *AttachThread* method will create a RexxThreadContext instance for additional threads that you add to an interpreter instance. Additionally, the *RexxMethodContext*, *RexxCallContext*, and *RexxExitContext* objects embed a RexxThreadContext object the same way that a RexxThreadContext object embeds a RexxInstance object.

1.8. Rexx Method Context Interface

A RexxMethodContext object is included as an argument to any native C++ method (*Section 1.14, "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* 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 conditions to be raised. These calls are invoked as if the current context is operating with SIGNAL ON ANY 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* API, and pending conditions can be cancelled using *ClearCondition*.

1.9. Rexx Call Context Interface

A RexxCallContext object is included as an argument to any native C++ routine (*Section 1.13, "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* 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 conditions to be raised. These calls are invoked as if the current context is operating with SIGNAL ON ANY 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* API, and pending conditions can be cancelled using *ClearCondition*.

1.10. Rexx Exit Context Interface

A RexxExitContext object is included as an argument to any *system exit* or *command handler*. The exit context provides services that are specific to an 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* 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 conditions to be raised. These calls are invoked as if the current context is operating with SIGNAL ON ANY 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* API, and pending conditions can be cancelled using *ClearCondition*.

1.11. *NEW* Rexx I/O Redirector Context Interface

A RexxIORedirectorContext object is included as the last argument to any redirecting *command handler*. The I/O Redirector context provides services that are specific to the redirection of STDIN, STDOUT, and STDERR of external commands, including:

• Providing information whether redirection was requested, and for which standard stream

• Retrieving input data for STDIN redirection

• Returning output data from STDOUT and STDERR redirection

API calls made using the RexxIORedirectorContext APIs may cause Rexx syntax errors or other conditions to be raised. These calls are invoked as if the current context is operating with SIGNAL ON ANY 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* API, and pending conditions can be cancelled using *ClearCondition*.

1.12. 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 1.18. 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 1.13, "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 1.14, "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 1.19. 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);
```

1.13. 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 2.5, "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* 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 1.20. API — REXX_TYPED_ROUTINE_PROTOTYPE

```
// 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()
};
```

1.13.1. 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 1.21. 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 1.22. 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 1.13.2, "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.

1.13.2. Routine Argument Types

A routine argument or return value may be a numeric type (*Section 1.2.2, "Rexx Numeric Types"*) or an object type (*Section 1.2.1, "*CHG* 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-dimensional 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 *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 1.23. 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* API is more appropriate in this situation.

Example 1.24. 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.

1.14. 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 1.25. 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()
};
```

1.14.1. 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 1.26. 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 1.27. 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* 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 expressions, but may be used as stand-alone message instructions.

1.14.2. Method Argument Types

A method argument or return value may be a numeric type (*Section 1.2.2, "Rexx Numeric Types"*) or an object type (*Section 1.2.1, "*CHG* 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-dimensional 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 *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 1.28. 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 1.29. 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 1.30. 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:

```
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.

1.14.3. 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.

1.14.3.1. 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 is to store the buffer object in an object instance variable using *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 that's part of the RexxThreadContext interface. Once created, you can access the Buffer's data area using *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

```
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 1.33. 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 dynamically 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 1.34. 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;
}
```

1.14.3.2. 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 referenced 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 1.35. 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;
}
```

1.14.3.3. 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 1.36. 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 1.37. 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.

1.14.3.4. 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 1.38. 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 *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 hierarchy 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 *CHG* ObjectToCSelf* allows the CSELF information for an object other than the current active object to be retrieved.

1.15. 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.

1.15.1. Writing Context Exit Handlers

The following is a sample exit handler declaration:

Example 1.39. 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.

1.15.1.1. 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/0/1/2* 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".

1.15.1.2. 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.

1.15.1.3. 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.

1.15.2. 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

1.15.2.1. 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 1.40. 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 2.2, "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.

1.15.2.2. 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 1.41. 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.

1.15.2.3. 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 1.42. 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.

1.15.2.4. 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 1.43. 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* and *rxcmd_addressl* define the current ADDRESS setting. *rxcmd_dll* currently is always NULL and *rxcmd_dll_len* is always zero. *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.

1.15.2.5. 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 1.44. 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 1.45. 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 1.46. 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 1.47. API — Rexx_IO_Exit parameter list

```
typedef struct {
   RXSTRING     rxmsq_name;      /* RXSTRING containing       */
                                 /* queue name.               */
} RXMSQNAM_PARM;
```

rxmsq_name contains the new queue name.

1.15.2.6. RXSIO

Standard input and output.
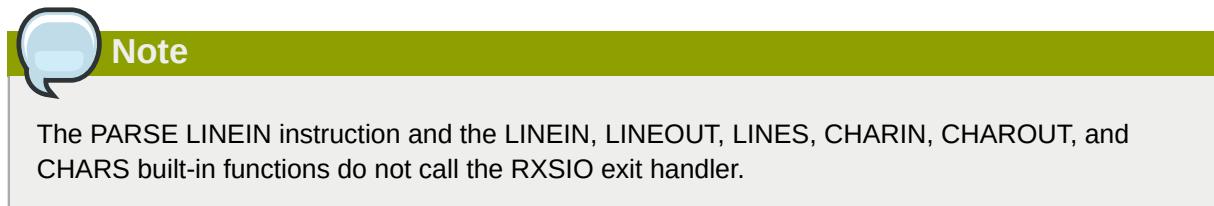
> **Note**
>
> The PARSE LINEIN instruction and the LINEIN, LINEOUT, LINES, CHARIN, CHAROUT, and
> CHARS built-in functions do not call the RXSIO exit handler.

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 1.48. 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 1.49. 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 1.50. 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 1.51. 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.

1.15.2.7. 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 1.52. 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;
> ```

1.15.2.8. 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 1.53. 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.

1.15.2.9. 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 1.54. 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.

1.15.2.10. 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* 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 1.55. 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.

1.15.2.11. 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* API to initialize application-specific variables.

     - Use *SetThreadTrace* API to switch on the interactive Rexx debug mode.

   - Parameters: None.

1.15.2.12. 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* to retrieve the Rexx variable values.

   - Parameters: None.

1.16. Command Handler Interface

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

There are two types of command handlers: in addition to standard "direct" command handlers, "redirecting" command handlers offer optional redirection of STDIN from Rexx objects and redirection of STDOUT and STDERR to Rexx objects. Redirection is requested by using the WITH subkeyword of the ADDRESS instruction.

Command handlers can be registered by using interpreter instance options *DIRECT_ENVIRONMENTS* or *REDIRECTING_ENVIRONMENTS* when the interpreter instance is created, or through the **NEW* AddCommandEnvironment* API.

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, returning a return code value back to the Rexx program, and, if requested, providing redirected input to the command and capturing command output.

Handlers are called using two different function signatures, the first for direct handlers, and the second signature for redirecting command handlers:

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

RexxObjectPtr RexxEntry RedirectingCommandHandler(RexxExitContext *context,
     RexxStringObject address, RexxStringObject command, RexxIORedirectorContext *ioContext)
```

Arguments

context A *RexxExitContext* 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.

ioContext (For redirecting command handlers only.) A *RexxIORedirectorContext* interface vector that provides access to redirection API methods **NEW* AreOutputAndErrorSameTarget*, **NEW* IsErrorRedirected*, **NEW* IsInputRedirected*, **NEW* IsOutputRedirected*, **NEW* IsRedirectionRequested*, **NEW* ReadInput*, **NEW* ReadInputBuffer*, **NEW* WriteError*, **NEW* WriteErrorBuffer*, **NEW* WriteOutput*, and **NEW* WriteOutputBuffer*.

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* API provided by the RexxExitContext. For example:

```
// 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);
```

1.17. Rexx Interface Methods Listing

This section describes each available method and its associated context.

ooRexx 5.0.0 has introduced the following new APIs.

NEW AddCommandEnvironment	*NEW* ReallocateObjectMemory
NEW AllocateObjectMemory	*NEW* SendMessageScoped
NEW AreOutputAndErrorSameTarget	*NEW* SetGuardOffWhenUpdated
NEW FreeObjectMemory	*NEW* SetGuardOnWhenUpdated
NEW GetContextVariableReference	*NEW* SetVariableReferenceValue
NEW GetInterpreterInstance	*NEW* StringTableAt
NEW GetObjectVariableReference	*NEW* StringTablePut
NEW IsErrorRedirected	*NEW* StringTableRemove
NEW IsInputRedirected	*NEW* ThrowCondition
NEW IsOutputRedirected	*NEW* ThrowException/0/1/2
NEW IsRedirectionRequested	*NEW* VariableReferenceName
NEW IsStringTable	*NEW* VariableReferenceValue
NEW IsVariableReference	*NEW* WriteError
NEW NewStringTable	*NEW* WriteErrorBuffer
NEW ReadInput	*NEW* WriteOutput
NEW ReadInputBuffer	*NEW* WriteOutputBuffer

1.17.1. *NEW* AddCommandEnvironment

This API is available in contexts *Instance*, *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
CSTRING name;
REXXPFN handler;
size_t type;


// Method Syntax Form(s)


context->AddCommandEnvironment(name, handler, type);
```

Adds a *command handler* to the Rexx interpreter instance. If a command handler with the specified *name* already exists, it is overwritten.

Arguments

name	The ASCII-Z name of the command handler's environment name.
handler	The address of the subcommand handler entry point within the application executable code. For a description of the required handler function signature see *Command Handler Interface*.
type	The type of command handler to add. DIRECT_COMMAND_ENVIRONMENT for a command handler with no support for redirection. REDIRECTING_COMMAND_ENVIRONMENT for a command handler that supports redirection.

Returns

Void.

1.17.2. *NEW* AllocateObjectMemory

This API is available in context *Method* since ooRexx 5.0.

```
size_t bytes;
POINTER ptr;

// Method Syntax Form(s)

ptr = context->AllocateObjectMemory(bytes);
```

Allocates object memory similar to malloc(), where the allocated memory is garbage-collected together with the object.

Arguments

bytes	The number of bytes of memory to allocate.

Returns

A POINTER to the allocated object memory.

See also methods *NEW* ReallocateObjectMemory and *NEW* FreeObjectMemory.

1.17.3. *NEW* AreOutputAndErrorSameTarget

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
logical_t flag;

// Method Syntax Form(s)

flag = context->AreOutputAndErrorSameTarget();
```

Tests whether for the current command the output object and the error object specified by the WITH subkeyword of an ADDRESS instruction are the same objects.

Arguments

None.

Returns

1 if the output and the error object are te same, **0** otherwise.

See also methods **NEW* IsErrorRedirected*, **NEW* IsInputRedirected*, **NEW* IsOutputRedirected*, **NEW* IsRedirectionRequested*, **NEW* ReadInput*, **NEW* ReadInputBuffer*, **NEW* WriteError*, **NEW* WriteErrorBuffer*, **NEW* WriteOutput*, and **NEW* WriteOutputBuffer*.

1.17.4. Array

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

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

// Method Syntax Form(s)

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 creates a new single-dimensional 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.

1.17.5. ArrayAppend

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

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

// Method Syntax Form(s)

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.

1.17.6. ArrayAppendString

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

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

// Method Syntax Form(s)

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.

1.17.7. ArrayAt

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

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

// Method Syntax Form(s)

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.

1.17.8. ArrayDimension

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxArrayObject arr;
size_t sz;

// Method Syntax Form(s)

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

Returns number of dimensions of an Array.

Arguments

 arr   The target Array object.

Returns

The number of Array dimensions.

1.17.9. ArrayItems

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxArrayObject arr;
size_t sz;

// Method Syntax Form(s)

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

Returns number of elements in an Array.

Arguments

 arr   The source Array object.

Returns

The number of Array elements.

1.17.10. ArrayOfFour

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

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

// Method Syntax Form(s)

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

Create a new single-dimensional 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.

1.17.11. ArrayOfOne

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxArrayObject arr;
RexxObjectPtr obj;

// Method Syntax Form(s)

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

Create a new single-dimensional Array with the specified object.

Arguments

obj   The object to be added.

Returns

The new Array object.

1.17.12. ArrayOfThree

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

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

// Method Syntax Form(s)

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

Create a new single-dimensional 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.

1.17.13. ArrayOfTwo

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxArrayObject arr;
RexxObjectPtr obj1, obj2;

// Method Syntax Form(s)

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

Create a new single-dimensional 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.

1.17.14. ArrayPut

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

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

// Method Syntax Form(s)

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.

1.17.15. ArraySize

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxArrayObject arr;
size_t sz;

// Method Syntax Form(s)
```

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

Returns the size of an Array.

Arguments

arr The source Array object.

Returns

The Array size.

1.17.16. AttachThread

This API is available in context *Instance*.

```
RexxThreadContext *tc;

// Method Syntax Form(s)

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.

1.17.17. BufferData

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxBufferObject obj;
POINTER str;

// Method Syntax Form(s)

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.

1.17.18. BufferLength

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxBufferObject obj;
size_t sz;

// Method Syntax Form(s)

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.

1.17.19. BufferStringData

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxBufferStringObject obj;
POINTER str;

// Method Syntax Form(s)

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* before it can be used in any other context.

1.17.20. BufferStringLength

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxBufferStringObject obj;
size_t sz;

// Method Syntax Form(s)

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

Return the length of a RexxBufferStringObject instance.

Arguments

obj   The source RexxBufferStringObject.

Returns

The length of the RexxBufferStringObject.

1.17.21. CallProgram

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

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

// Method Syntax Form(s)

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 return a NULLOBJECT result. The
CheckCondition can be used to check if any errors occurred during the call.

1.17.22. CallRoutine

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj, ret;
RexxArrayObject arr;

// Method Syntax Form(s)

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 return a NULLOBJECT result. The
CheckCondition can be used to check if any errors occurred during the call.

1.17.23. CheckCondition

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
logical_t flag;

// Method Syntax Form(s)

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.

1.17.24. ClearCondition

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
// Method Syntax Form(s)

context->ClearCondition();
```

Clears any pending condition status.

Arguments

None.

Returns

Void.

1.17.25. CString

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

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

// Method Syntax Form(s)

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..

1.17.26. DecodeConditionInfo

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dir;
RexxCondition cond;

// Method Syntax Form(s)

context->DecodeConditionInfo(dir, &cond);
```

Decodes the condition information into a RexxCondition structure, which is defined as follows:

```
typedef struct
{
  wholenumber_t code;                // full condition code
  wholenumber_t rc;                  // return code value
  size_t          position;          // line number position
  RexxStringObject conditionName;    // name of the condition
  RexxStringObject message;          // fully filled in message
  RexxStringObject errortext;        // major error text
  RexxStringObject program;          // program name
  RexxStringObject description;      // description text
  RexxArrayObject  additional;       // additional information
} RexxCondition;
```

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*.

1.17.27. DetachThread

This API is available in context *Thread*.

```
// Method Syntax Form(s)

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.

1.17.28. DirectoryAt

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dirobj;
RexxObjectPtr obj;
CSTRING str;

// Method Syntax Form(s)

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.

1.17.29. DirectoryPut

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dirobj;
RexxObjectPtr item;
CSTRING index;

// Method Syntax Form(s)

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.

1.17.30. DirectoryRemove

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dirobj;
RexxObjectPtr obj;
CSTRING str;

// Method Syntax Form(s)

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.

1.17.31. DisplayCondition

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
wholenumber_t rc;

// Method Syntax Form(s)

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.

1.17.32. Double

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
double n;
logical_t flag;

// Method Syntax Form(s)
```

```
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**.

1.17.33. DoubleToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
double n;

// Method Syntax Form(s)

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.

1.17.34. DoubleToObjectWithPrecision

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
size_t p;
double n;

// Method Syntax Form(s)

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.

1.17.35. DropContextVariable

This API is available in contexts *Call* and *Exit*.

```
CSTRING name;

// Method Syntax Form(s)

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.

1.17.36. DropObjectVariable

This API is available in context *Method*.

```
CSTRING str;

// Method Syntax Form(s)

context->DropObjectVariable(str);
```

Drops an instance variable in the current method's scope.

Arguments

str	The name of the object variable.

Returns

Void.

1.17.37. DropStemArrayElement

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStemObject sobj;
size_t n;
```

```
// Method Syntax Form(s)

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.

1.17.38. DropStemElement

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStemObject sobj;
CSTRING name;

// Method Syntax Form(s)

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.

1.17.39. False

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;

// Method Syntax Form(s)

obj = context->False();
```

This method returns the Rexx .false (0) object.

Arguments

None.

Returns

The Rexx .false object.

1.17.40. FindClass

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxClassObject class;
CSTRING name;

// Method Syntax Form(s)

class = context->FindClass(name);
```

Locates a Class object in either the current Thread or Exit context, or in the current Method or Routine Package context. The latter case is equivalent to calling *FindContextClass*.

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.

1.17.41. FindContextClass

This API is available in contexts *Method* and *Call*.

```
CSTRING name;
RexxClassObject obj;

// Method Syntax Form(s)

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.

1.17.42. FindPackageClass

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxPackageObject pkg;
RexxClassObject class;
CSTRING name;

// Method Syntax Form(s)

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.

1.17.43. FinishBufferString

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxBufferStringObject obj;
RexxStringObject str;
size_t len;

// Method Syntax Form(s)

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.

1.17.44. ForwardMessage

This API is available in context *Method*.

```
CSTRING str;
RexxObjectPtr obj, ret;
RexxClassObject sobj;
RexxArrayObject arr;

// Method Syntax Form(s)

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	An 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.

1.17.45. *NEW* FreeObjectMemory

This API is available in context *Method* since ooRexx 5.0.

```
POINTER ptr;

// Method Syntax Form(s)

context->FreeObjectMemory(ptr);
```

Frees object memory allocated with *NEW* AllocateObjectMemory. Object memory is also automatically freed at the time the object gets garbage-collected.

Arguments

ptr	A POINTER to object memory allocated with *NEW* AllocateObjectMemory or reallocated with *NEW* ReallocateObjectMemory.

Returns

Void.

See also methods *NEW* AllocateObjectMemory and *NEW* ReallocateObjectMemory.

1.17.46. GetAllContextVariables

This API is available in contexts *Call* and *Exit*.

```
RexxDirectoryObject obj;

// Method Syntax Form(s)

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.

1.17.47. GetAllStemElements

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStemObject sobj;
RexxDirectoryObject obj;

// Method Syntax Form(s)

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.

1.17.48. GetApplicationData

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
// Method Syntax Form(s)

ptr = context->GetApplicationData();
```

Returns the application data pointer that was set via the APPLICATION_DATA option when the interpreter instance was created.

Arguments

None.

Returns

The application instance data set when the interpreter instance was created.

1.17.49. GetArgument

This API is available in contexts *Method* and *Call*.

```
RexxObjectPtr obj;
size_t n;

// Method Syntax Form(s)

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.

1.17.50. GetArguments

This API is available in contexts *Method* and *Call*.

```
RexxArrayObject arr;

// Method Syntax Form(s)

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.

1.17.51. GetCallerContext

This API is available in contexts *Call* and *Exit*.

```
RexxObjectPtr obj;

// Method Syntax Form(s)

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.

1.17.52. GetConditionInfo

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dir;
```

```
// Method Syntax Form(s)

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.

1.17.53. GetContextDigits

This API is available in context *Call*.

```
stringsize_t sz;

// Method Syntax Form(s)

sz = context->GetContextDigits();
```

Get the routine caller's current NUMERIC DIGITS setting.

Arguments

None.

Returns

The current NUMERIC DIGITS setting.

1.17.54. GetContextForm

This API is available in context *Call*.

```
stringsize_t sz;

// Method Syntax Form(s)

sz = context->GetContextForm();
```

Get the routine caller's current NUMERIC FORM setting.

Arguments

None.

Returns

The current NUMERIC FORM setting.

1.17.55. GetContextFuzz

This API is available in context *Call*.

```
stringsize_t sz;

// Method Syntax Form(s)

sz = context->GetContextFuzz();
```

Get the routine caller's current NUMERIC FUZZ setting.

Arguments

None.

Returns

The current NUMERIC FUZZ setting.

1.17.56. GetContextVariable

This API is available in contexts *Call* and *Exit*.

```
RexxObjectPtr obj;
CSTRING name;

// Method Syntax Form(s)

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.

1.17.57. *NEW* GetContextVariableReference

This API is available in contexts *Call* and *Exit* since ooRexx 5.0.

```
CSTRING name;
RexxVariableReferenceObject obj;

// Method Syntax Form(s)

obj = context->GetContextVariableReference(name);
```

Creates a VariableReference instance from a context variable name.

Arguments

name   The name of a simple or a stem context variable for which a reference should be created. A compound variable name is not allowed.

Returns

A VariableReference object referencing *name*.

See also methods **NEW* GetObjectVariableReference*, **NEW* IsVariableReference*, **NEW* SetVariableReferenceValue*, **NEW* VariableReferenceName*, and **NEW* VariableReferenceValue*.

1.17.58. *CHG* GetCSelf

This API is available in context *Method*.

```
POINTER ptr;

// Method Syntax Form(s)

ptr = context->GetCSelf();
```

Returns a pointer to the CSELF value for the current object. CSELF is a special argument type used for classes to store native pointers or structures inside an object instance. GetCSelf() will search all of the object's variable scopes 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

None.

Returns

The CSELF value for the current object, or NULL if no CSELF value was found.

See also method **CHG* ObjectToCSelf*.

1.17.59. GetGlobalEnvironment

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dir;

// Method Syntax Form(s)

dir = context->GetGlobalEnvironment();
```

Returns a reference to the .environment Directory.

Arguments

None.

Returns

A RexxDirectoryObject pointer to the .environment Directory.

1.17.60. *NEW* GetInterpreterInstance

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxInstance *instance;

// Method Syntax Form(s)

instance = context->GetInterpreterInstance();
```

Returns the interpreter instance context the current context is running on.

Arguments

None.

Returns

A RexxInstance pointer to the interpreter instance context.

1.17.61. GetLocalEnvironment

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dir;

// Method Syntax Form(s)

dir = context->GetLocalEnvironment();
```

Returns a reference to the interpreter instance .local Directory.

Arguments

None.

Returns

A RexxDirectoryObject pointer to the .local Directory.

1.17.62. GetMessageName

This API is available in context *Method*.

```
CSTRING str;

// Method Syntax Form(s)

str = context->GetMessageName(obj);
```

Returns the message name used to invoke the current method.

Arguments

None.

Returns

The current method message name.

1.17.63. GetMethod

This API is available in context *Method*.

```
RexxMethodObject obj;

// Method Syntax Form(s)

obj = context->GetMethod();
```

Returns the Method object for the currently executing method.

Arguments

None.

Returns

The current Method object.

1.17.64. GetMethodPackage

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxMethodObject obj;
RexxPackageObject pkg;

// Method Syntax Form(s)

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.

1.17.65. GetObjectVariable

This API is available in context *Method*.

```
CSTRING str;
```

```
RexxObjectPtr obj;

// Method Syntax Form(s)

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.

1.17.66. *NEW* GetObjectVariableReference

This API is available in context *Method* since ooRexx 5.0.

```
CSTRING name;
RexxVariableReferenceObject obj;

// Method Syntax Form(s)

obj = context->GetObjectVariableReference(name);
```

Creates a VariableReference instance from an object variable name.

Arguments

name   The name of a simple or a stem object variable for which a reference should be
                    created. A compound variable name is not allowed.

Returns

A VariableReference object referencing *name*.

See also methods *NEW* GetContextVariableReference, *NEW* IsVariableReference, *NEW* SetVariableReferenceValue, *NEW* VariableReferenceName, and *NEW* VariableReferenceValue.

1.17.67. GetPackageClasses

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dir;
RexxPackageObject pkg;

// Method Syntax Form(s)

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.

1.17.68. GetPackageMethods

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dir;
RexxPackageObject pkg;

// Method Syntax Form(s)

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.

1.17.69. GetPackagePublicClasses

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dir;
RexxPackageObject pkg;

// Method Syntax Form(s)

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.

1.17.70. GetPackagePublicRoutines

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dir;
RexxPackageObject pkg;

// Method Syntax Form(s)

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.

1.17.71. GetPackageRoutines

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject dir;
RexxPackageObject pkg;

// Method Syntax Form(s)

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.

1.17.72. GetRoutine

This API is available in context *Call*.

```
RexxRoutineObject obj;

// Method Syntax Form(s)

obj = context->GetRoutine();
```

Returns current Routine object.

Arguments

None

Returns

The current Routine object.

1.17.73. GetRoutineName

This API is available in context *Call*.

```
CSTRING name;

// Method Syntax Form(s)

name = context->GetRoutineName();
```

Returns the name of the current routine.

Arguments

None

Returns

A pointer ASCII-Z routine name.

1.17.74. GetRoutinePackage

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxRoutineObject obj;
RexxPackageObject pkg;

// Method Syntax Form(s)

pkg = context->GetRoutinePackage(obj);
```

Returns Routine object's associated Package object.

Arguments

  obj   The routine object to query.

Returns

The Package object instance.

1.17.75. GetScope

This API is available in context *Method*.

```
RexxObjectPtr obj;

// Method Syntax Form(s)

obj = context->GetScope();
```

Return the current active method's scope.

Arguments

None.

Returns

The current Method's scope.

1.17.76. GetSelf

This API is available in context *Method*.

```
RexxObjectPtr obj;

// Method Syntax Form(s)

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.

1.17.77. GetStemArrayElement

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStemObject sobj;
RexxObjectPtr obj;
size_t n;

// Method Syntax Form(s)

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.

1.17.78. GetStemElement

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStemObject sobj;
RexxObjectPtr obj;
CSTRING name;

// Method Syntax Form(s)

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 object at the target index or NULLOBJECT if the target index has not been assigned a value.

1.17.79. GetStemValue

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStemObject sobj;
RexxObjectPtr obj;
CSTRING name;

// Method Syntax Form(s)

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.

1.17.80. GetSuper

This API is available in context *Method*.

```
RexxObjectPtr obj;

// Method Syntax Form(s)
```

```
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.

1.17.81. Halt

This API is available in context *Instance*.

```
// Method Syntax Form(s)

context->Halt();
```

Raise a HALT condition on all threads associated with the interpreter instance.

Arguments

None.

Returns

Void.

1.17.82. HaltThread

This API is available in context *Thread*.

```
// Method Syntax Form(s)

context->HaltThread();
```

Raises a HALT condition on the thread corresponding to the current *context* pointer.

Arguments

None

Returns

Void.

1.17.83. HasMethod

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
logical_t flag;
```

```
RexxObjectPtr obj;
CSTRING name;

// Method Syntax Form(s)

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.

1.17.84. Int32

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;
int32_t n;

// Method Syntax Form(s)

obj = context->Int32(n);

flag = 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*.

1.17.85. Int32ToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
int32_t n;

// Method Syntax Form(s)
```

```
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.

1.17.86. Int64

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;
int64_t n;

// Method Syntax Form(s)

obj = context->Int64(n);

flag = 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*.

1.17.87. Int64ToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
int64_t n;

// Method Syntax Form(s)

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.

1.17.88. InterpreterVersion

This API is available in contexts *Instance*, *Thread*, *Method*, *Call*, and *Exit*.

```
size_t version;

// Method Syntax Form(s)

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.

1.17.89. Intptr

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;
intptr_t n;

// Method Syntax Form(s)

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*.

1.17.90. IntptrToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
intptr_t n;

// Method Syntax Form(s)

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.

1.17.91. InvalidRoutine

This API is available in context *Call*.

```
RexxDirectoryObject obj;

// Method Syntax Form(s)

context->InvalidRoutine();
```

Raises the standard Error 40, "Incorrect call to routine" syntax error for the current routine. This error will be raised by the Rexx runtime once the routine returns.

Arguments

None.

Returns

Void.

1.17.92. IsArray

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)
```

```
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.

1.17.93. IsBuffer

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)

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.

1.17.94. IsDirectory

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)

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.

1.17.95. *NEW* IsErrorRedirected

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
logical_t flag;

// Method Syntax Form(s)

flag = context->IsErrorRedirected();
```

Tests whether for the current command error output redirection was requested using the WITH subkeyword of an ADDRESS instruction.

Arguments

None.

Returns

1 if error redirection was requested, **0** otherwise.

See also methods *NEW* AreOutputAndErrorSameTarget, *NEW* IsInputRedirected, *NEW* IsOutputRedirected, *NEW* IsRedirectionRequested, *NEW* ReadInput, *NEW* ReadInputBuffer, *NEW* WriteError, *NEW* WriteErrorBuffer, *NEW* WriteOutput, and *NEW* WriteOutputBuffer.

1.17.96. *NEW* IsInputRedirected

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
logical_t flag;

// Method Syntax Form(s)

flag = context->IsInputRedirected();
```

Tests whether for the current command input redirection was requested using the WITH subkeyword of an ADDRESS instruction.

Arguments

None.

Returns

1 if input redirection was requested, **0** otherwise.

See also methods *NEW* AreOutputAndErrorSameTarget, *NEW* IsErrorRedirected, *NEW* IsOutputRedirected, *NEW* IsRedirectionRequested, *NEW* ReadInput, *NEW* ReadInputBuffer, *NEW* WriteError, *NEW* WriteErrorBuffer, *NEW* WriteOutput, and *NEW* WriteOutputBuffer.

1.17.97. IsInstanceOf

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
```

```
RexxClassObj class;
logical_t flag;

// Method Syntax Form(s)

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.

1.17.98. IsMethod

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)

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.

1.17.99. IsMutableBuffer

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)

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.

1.17.100. IsOfType

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
CSTRING class;
logical_t flag;

// Method Syntax Form(s)

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.

1.17.101. *NEW* IsOutputRedirected

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
logical_t flag;

// Method Syntax Form(s)

flag = context->IsOutputRedirected();
```

Tests whether for the current command output redirection was requested using the WITH subkeyword of an ADDRESS instruction.

Arguments

None.

Returns

1 if output redirection was requested, **0** otherwise.

See also methods *NEW* AreOutputAndErrorSameTarget, *NEW* IsErrorRedirected, *NEW* IsInputRedirected, *NEW* IsRedirectionRequested, *NEW* ReadInput, *NEW* ReadInputBuffer, *NEW* WriteError, *NEW* WriteErrorBuffer, *NEW* WriteOutput, and *NEW* WriteOutputBuffer.

1.17.102. IsPointer

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)

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.

1.17.103. *NEW* IsRedirectionRequested

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
logical_t flag;

// Method Syntax Form(s)

flag = context->IsRedirectionRequested();
```

Tests whether for the current command any redirection was requested using the WITH subkeyword of an ADDRESS instruction.

Arguments

None.

Returns

1 if any redirection was requested, **0** otherwise.

See also methods *NEW* AreOutputAndErrorSameTarget, *NEW* IsErrorRedirected, *NEW* IsInputRedirected, *NEW* IsOutputRedirected, *NEW* ReadInput, *NEW* ReadInputBuffer, *NEW* WriteError, *NEW* WriteErrorBuffer, *NEW* WriteOutput, and *NEW* WriteOutputBuffer.

1.17.104. IsRoutine

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)
```

```
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.

1.17.105. IsStem

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)

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.

1.17.106. IsString

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)

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.

1.17.107. *NEW* IsStringTable

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)

flag = context->IsStringTable(obj);
```

Tests if an Object is a StringTable object. A true result indicates the RexxObjectPtr value may be safely cast to a RexxStringTableObject.

Arguments

obj   The object to be tested.

Returns

1 = is a StringTable object, 0 = not a StringTable object.

See also methods *NEW* NewStringTable*, *NEW* StringTableAt*, *NEW* StringTablePut*, and *NEW* StringTableRemove*.

1.17.108. *NEW* IsVariableReference

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxObjectPtr obj;
logical_t flag;

// Method Syntax Form(s)

flag = context->IsVariableReference(obj);
```

Tests if an Object is a VariableReference object. A true result indicates the RexxObjectPtr value may be safely cast to a RexxVariableReferenceObject.

Arguments

obj   The object to be tested.

Returns

1 = is a VariableReference object, 0 = not a VariableReference object.

See also methods *NEW* GetContextVariableReference*, *NEW* GetObjectVariableReference*, *NEW* SetVariableReferenceValue*, *NEW* VariableReferenceName*, and *NEW* VariableReferenceValue*.

1.17.109. LanguageLevel

This API is available in contexts *Instance*, *Thread*, *Method*, *Call*, and *Exit*.

```
size_t langlevel;

// Method Syntax Form(s)

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 0x00000605 for language level 6.05. The oorexxapi.h header file will have a define matching these values using a the naming convention REXX_LANGUAGE_6_05 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.

1.17.110. LoadLibrary

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
CSTRING name;
logical_t success;

// Method Syntax Form(s)

success = 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.

1.17.111. LoadPackage

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
CSTRING name;
RexxPackageObject pkg;

// Method Syntax Form(s)
```

```
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*.

1.17.112. LoadPackageFromData

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
CSTRING name, data;
size_t sz;
RexxPackageObject pkg;

// Method Syntax Form(s)

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*.

1.17.113. Logical

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag, n;

// Method Syntax Form(s)

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.

1.17.114. LogicalToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag, n;

// Method Syntax Form(s)

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.

1.17.115. MutableBufferCapacity

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxMutableBufferObject obj;
size_t sz;

// Method Syntax Form(s)

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.

1.17.116. MutableBufferData

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxMutableBufferObject obj;
POINTER str;

// Method Syntax Form(s)

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.

1.17.117. MutableBufferLength

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxMutableBufferObject obj;
size_t sz;

// Method Syntax Form(s)

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.

1.17.118. NewArray

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxArrayObject obj;
size_t len;

// Method Syntax Form(s)

obj = context->NewArray(d);
```

Create an Array object of the specified size.

Arguments

  d   The size of the Array.

Returns

The new Array object.

1.17.119. NewBuffer

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxBufferObject obj;
size_t len;

// Method Syntax Form(s)

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.

1.17.120. NewBufferString

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxBufferStringObject obj;
size_t len;

// Method Syntax Form(s)

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* to transform this into a completed String object.

Arguments

len   The maximum length of the final String object.

Returns

A new RexxBufferString value.

1.17.121. NewDirectory

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxDirectoryObject obj;

// Method Syntax Form(s)
```

```
obj = context->NewDirectory();
```

Create a Directory object.

Arguments

None

Returns

The new Directory object.

1.17.122. NewMethod

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxMethodObject obj;
CSTRING name, code;
size_t sz;

// Method Syntax Form(s)

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*.

1.17.123. NewMutableBuffer

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxMutableBufferObject obj;
size_t len;

// Method Syntax Form(s)

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.

1.17.124. NewPointer

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxPointerObject obj;
POINTER p;

// Method Syntax Form(s)

obj = context->NewPointer(p);
```

Create a new Pointer object from a C pointer.

Arguments

p   The source C pointer.

Returns

The created Pointer object.

1.17.125. NewRoutine

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxRoutineObject obj;
CSTRING name, code;
size_t sz;

// Method Syntax Form(s)

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 routine will return a NULLOBJECT value. Information about errors can be retrieved using *GetConditionInfo*.

1.17.126. NewStem

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStemObject obj;
CSTRING str;

// Method Syntax Form(s)

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.

1.17.127. NewString

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStringObject obj;
CSTRING str;
size_t len;

// Method Syntax Form(s)

obj = context->NewString(str, len);
```

Create a new String object from program data.

Arguments

str   A pointer to a data buffer containing the string data.

len   Length of the *str* data buffer.

Returns

The new String object.

1.17.128. NewStringFromAsciiz

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStringObject obj;
CSTRING str;

// Method Syntax Form(s)

obj = context->NewStringFromAsciiz(str);
```

Create a new String object from a C string.

Arguments

str   A pointer to a null-terminated ASCII-Z string.

Returns

The new String object.

1.17.129. *NEW* NewStringTable

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxStringTableObject obj;

// Method Syntax Form(s)

obj = context->NewStringTable();
```

Create a StringTable object.

Arguments

None

Returns

The new StringTable object.

See also methods **NEW* IsStringTable*, **NEW* StringTableAt*, **NEW* StringTablePut*, and **NEW* StringTableRemove*.

1.17.130. NewSupplier

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxSupplierObject obj;
RexxArrayObject arr1, arr2;

// Method Syntax Form(s)

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.

1.17.131. Nil

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
```

```
// Method Syntax Form(s)

obj = context->Nil();
```

Returns the Rexx Nil object.

Arguments

None.

Returns

The Rexx Nil object.

1.17.132. NullString

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStringObject obj;

// Method Syntax Form(s)

obj = context->NullString();
```

This method returns a string object of zero length.

Arguments

None.

Returns

A null String object.

1.17.133. *CHG* ObjectToCSelf

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj, scope;
POINTER ptr;

// Method Syntax Form(s)

ptr = context->ObjectToCSelf(obj);

ptr = context->ObjectToCSelf(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. Objects that rely on CSELF values typically set the variable CSELF inside an INIT method for the class.

This method has two forms. The first form searches all of the object's variable scopes for a variable named CSELF. The second form searches 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.

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.

See also method *CHG* GetCSelf.

1.17.134. ObjectToDouble

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
double n;
logical_t flag;

// Method Syntax Form(s)

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*.

1.17.135. ObjectToInt32

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
int32_t n;
logical_t flag;

// Method Syntax Form(s)

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*.

1.17.136. ObjectToInt64

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
int64_t n;
logical_t flag;

// Method Syntax Form(s)

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*.

1.17.137. ObjectToIntptr

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
intptr_t n;
logical_t flag;

// Method Syntax Form(s)

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*.

1.17.138. ObjectToLogical

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag, n;
```

```
// Method Syntax Form(s)

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*.

1.17.139. ObjectToString

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
RexxStringObject str;

// Method Syntax Form(s)

str = context->ObjectToString(obj);
```

Convert an Object to a String object.

Arguments

obj   The source object for the conversion.

Returns

The String object.

1.17.140. ObjectToStringSize

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
size_t n;
logical_t flag;

// Method Syntax Form(s)

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*.

1.17.141. ObjectToStringValue

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
CSTRING str;

// Method Syntax Form(s)

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.

1.17.142. ObjectToUintptr

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
uintptr_t n;
logical_t flag;

// Method Syntax Form(s)

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*.

1.17.143. ObjectToUnsignedInt32

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
```

```
uint32_t n;
logical_t flag;

// Method Syntax Form(s)

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*.

1.17.144. ObjectToUnsignedInt64

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
uint64_t n;
logical_t flag;

// Method Syntax Form(s)

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*.

1.17.145. ObjectToValue

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
ValueDescriptor desc;
logical_t flag;

// Method Syntax Form(s)

flag = context->ObjectToValue(obj, &desc);
```

Convert a Rexx object to another type. The target type is identified by the ValueDescriptor
structure, and can be 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.

1.17.146. ObjectToWholeNumber

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
wholenumber_t n;
logical_t flag;

// Method Syntax Form(s)

flag = context->ObjectToWholeNumber(obj, &n);
```

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*.

1.17.147. PointerValue

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxPointerObject obj;
POINTER p;

// Method Syntax Form(s)

p = context->PointerValue(obj);
```

Return the wrapped C pointer value from a RexxPointerObject.

Arguments

obj	The source RexxPointerObject.

Returns

The wrappered C pointer value.

1.17.148. RaiseCondition

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
CSTRING str;
RexxStringObject sobj;
RexxArrayObject arr;
RexxObjectPtr obj;

// Method Syntax Form(s)

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	An 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.

See also methods **NEW* ThrowCondition* and *RaiseException/0/1/2*.

1.17.149. RaiseException/0/1/2

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
size_t n;
RexxObjectPtr array, obj1, obj2;

// Method Syntax Form(s)

context->RaiseException(n, array);

context->RaiseException0(n);

context->RaiseException1(n, obj1);

context->RaiseException2(n, obj1, obj2);
```

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.
array	An Array of error message substitution values.
obj1	The first substitution value for the error message.
obj2	The second substitution value for the error message.

Returns

Void.

See also methods *NEW* ThrowException/0/1/2.

1.17.150. *NEW* ReadInput

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
CSTRING data;
size_t  length;

// Method Syntax Form(s)

context->ReadInput(&data, &length);
```

Returns the next item or line of data from an input redirection Rexx object that was specified using the WITH subkeyword of an ADDRESS instruction. Items are converted to strings and missing items are replaced by a null string.

Arguments

data	The returned string. If no more items or lines are available, or there is no input redirection, NULL is returned.
length	The returned length of *data*. If an item is missing, or no more items or lines are available, or there is no input redirection, **0** is returned.

Returns

Void. Arguments *data* and *length* are updated.

See also methods *NEW* AreOutputAndErrorSameTarget, *NEW* IsErrorRedirected, *NEW* IsInputRedirected, *NEW* IsOutputRedirected, *NEW* IsRedirectionRequested, *NEW* ReadInputBuffer, *NEW* WriteError, *NEW* WriteErrorBuffer, *NEW* WriteOutput, and *NEW* WriteOutputBuffer.

1.17.151. *NEW* ReadInputBuffer

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
CSTRING data;
size_t  length;

// Method Syntax Form(s)
```

```
context->ReadInputBuffer(&data, &length);
```

Returns a string of all items or lines from an input redirection Rexx object that was specified using the WITH subkeyword of an ADDRESS instruction. Items are converted to strings and separated by the platform-specific line-end characters. Missing items are replaced by a null string.

Arguments

data   The returned string. NULL if there is no input redirection.

length   The returned length of *data*. **0** if there is no input redirection.

Returns

Void. Arguments *data* and *length* are updated.

See also methods *NEW* AreOutputAndErrorSameTarget, *NEW* IsErrorRedirected, *NEW* IsInputRedirected, *NEW* IsOutputRedirected, *NEW* IsRedirectionRequested, *NEW* ReadInput, *NEW* WriteError, *NEW* WriteErrorBuffer, *NEW* WriteOutput, and *NEW* WriteOutputBuffer.

1.17.152. *NEW* ReallocateObjectMemory

This API is available in context *Method* since ooRexx 5.0.

```
POINTER ptr;
size_t bytes;

// Method Syntax Form(s)

ptr = context->ReallocateObjectMemory(ptr, bytes);
```

Reallocates object memory allocated with *NEW* AllocateObjectMemory. The current memory contents are copied to the new allocation and the current allocation will be automatically freed.

Arguments

ptr   A POINTER to object memory allocated with *NEW* AllocateObjectMemory or reallocated with *NEW* ReallocateObjectMemory.

bytes   The new memory size in bytes.

Returns

A POINTER to the reallocated object memory. If the new size is not larger than the currently allocated memory size, the current object memory pointer is returned.

See also methods *NEW* AllocateObjectMemory and *NEW* FreeObjectMemory.

1.17.153. RegisterLibrary

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
CSTRING name;
logical_t success;

// Method Syntax Form(s)
```

```
success = 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 1.12, "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.

1.17.154. ReleaseGlobalReference

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr ref;

// Method Syntax Form(s)

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.

1.17.155. ReleaseLocalReference

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr ref;

// Method Syntax Form(s)

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.

1.17.156. RequestGlobalReference

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr ref, obj;

// Method Syntax Form(s)

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*.

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.

1.17.157. ResolveStemVariable

This API is available in context *Call*.

```
RexxObjectPtr obj;
RexxStemObject stem;

// Method Syntax Form(s)

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.

1.17.158. SendMessage/0/1/2

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj, ret, arg1, arg2;
CSTRING msg;
RexxArrayObject arr;

// Method Syntax Form(s)

ret = context->SendMessage(obj, msg, arr);

ret = context->SendMessage0(obj, msg);

ret = context->SendMessage1(obj, msg, arg1);

ret = context->SendMessage2(obj, msg, arg1, arg2);
```

Send a message to an object. Message arguments can be specified as an array of objects, or for cases where no argument, one argument, or two arguments are needed, short cut methods `SendMessage0`, `SendMessage1`, and `SendMessage2` can be used.

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	An array of message arguments to the receiving method.
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. *CheckCondition* can be used to check if an error occurred during the call.

1.17.159. *NEW* SendMessageScoped

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxObjectPtr obj, ret;
CSTRING msg;
RexxClassObject class;
RexxArrayObject arr;

// Method Syntax Form(s)

ret = context->SendMessageScoped(obj, msg, class, arr);
```

Send a message to an object with a scope override.

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.
class	A class object where to start searching for *msg*.
arr	An 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. *CheckCondition* can be used to check if an error occurred during the call.

1.17.160. SetContextVariable

This API is available in contexts *Call* and *Exit*.

```
RexxObjectPtr obj;
CSTRING name;

// Method Syntax Form(s)

context->SetContextVariable(name, obj);
```

Sets the value of a Rexx variable in the current call 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* to set the associated compound variable.

Arguments

name	The name of the Rexx variable.
obj	The object to assign to the variable.

Returns

Void.

1.17.161. SetGuardOff

This API is available in context *Method*.

```
// Method Syntax Form(s)

context->SetGuardOff();
```

Release the guard lock for this method scope.

Arguments

None.

Returns

Void.

See also methods *SetGuardOn*, **NEW* SetGuardOnWhenUpdated*, and **NEW* SetGuardOffWhenUpdated*.

1.17.162. *NEW* SetGuardOffWhenUpdated

This API is available in context *Method* since ooRexx 5.0.

```
CSTRING name;
RexxObjectPtr ret;

// Method Syntax Form(s)

ret = context->SetGuardOffWhenUpdated(name);
```

Waits for an object variable to be updated and returns the new value. The guard state will be OFF on return.

Arguments

name   The name of the variable to wait for. Only simple variables or stem variables are allowed, no compound variables.

Returns

The new value of the variable. Returns a NULLOBJECT if an error occurs.

See also methods *SetGuardOn*, **NEW* SetGuardOnWhenUpdated*, and *SetGuardOff*.

1.17.163. SetGuardOn

This API is available in context *Method*.

```
// Method Syntax Form(s)

context->SetGuardOn();
```

Obtain the guard lock for this object scope.

Arguments

None.

Returns

Void.

See also methods **NEW* SetGuardOnWhenUpdated*, *SetGuardOff*, and **NEW* SetGuardOffWhenUpdated*.

1.17.164. *NEW* SetGuardOnWhenUpdated

This API is available in context *Method* since ooRexx 5.0.

```
CSTRING name;
RexxObjectPtr ret;

// Method Syntax Form(s)

ret = context->SetGuardOnWhenUpdated(name);
```

Waits for an object variable to be updated and returns the new value. The guard state will be ON on return.

Arguments

name   The name of the variable to wait for. Only simple variables or stem variables are allowed, no compound variables.

Returns

The new value of the variable. Returns a NULLOBJECT if an error occurs.

See also methods *SetGuardOn*, *SetGuardOff*, and **NEW* SetGuardOffWhenUpdated*.

1.17.165. SetMutableBufferCapacity

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxMutableBufferObject obj;
size_t len;
POINTER data;

// Method Syntax Form(s)

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.

1.17.166. SetMutableBufferLength

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxMutableBufferObject obj;
size_t len;
size_t newLen;

// Method Syntax Form(s)

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, use *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.

1.17.167. SetObjectVariable

This API is available in context *Method*.

```
CSTRING str;
RexxObjectPtr obj;

// Method Syntax Form(s)

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.

1.17.168. SetStemArrayElement

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStemObject sobj;
RexxObjectPtr obj;
size_t n;

// Method Syntax Form(s)

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.

1.17.169. SetStemElement

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxStemObject sobj;
RexxObjectPtr obj;
CSTRING name;

// Method Syntax Form(s)

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.

1.17.170. SetThreadTrace

This API is available in context *Thread*.

```
logical_t flag;

// Method Syntax Form(s)

context->SetThreadTrace(flag);
```

Sets the interactive trace state for the current thread.

Arguments

flag	New state for interactive trace.

Returns

Void.

1.17.171. SetTrace

This API is available in context *Instance*.

```
logical_t flag;

// Method Syntax Form(s)

context->SetTrace(flag);
```

Sets the interactive 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.

1.17.172. *NEW* SetVariableReferenceValue

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxVariableReferenceObject ref;
RexxObjectPtr value;

// Method Syntax Form(s)

context->SetVariableReferenceValue(ref, value);
```

Sets the value of VariableReference object *ref* to *value*.

Arguments

ref   The VariableReference object.

value   The object instance to be set as the value.

Returns

Void.

See also methods *NEW* GetContextVariableReference, *NEW* GetObjectVariableReference, *NEW* IsVariableReference, *NEW* VariableReferenceName, and *NEW* VariableReferenceValue.

1.17.173. String

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxRoutineObject obj;
CSTRING str;
size_t len;

// Method Syntax Form(s)

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.

1.17.174. StringData

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
CSTRING str;

// Method Syntax Form(s)

str = context->StringData(obj);
```

Returns a pointer to the String object's string data (for read-only).

Arguments

obj   The source String object for the data.

Returns

A pointer to the String object's string data.

The data pointed to does have a trailing **\0** character, but note that also the data itself (like any Rexx string) may contain embedded **\0** characters. The program **must not** modify the data the returned pointer points to.

1.17.175. StringGet

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
POINTER str;
size_t c, len1, len2;

// Method Syntax Form(s)

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.

1.17.176. StringLength

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
size_t sz;

// Method Syntax Form(s)

sz = context->StringLength(obj);
```

Return the length a String object.

Arguments

| *obj* | The source String object. |

Returns

The string length of the String object.

1.17.177. StringLower

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr srcobj, newobj;

// Method Syntax Form(s)

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.

1.17.178. StringSize

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;
stringsize_t n;

// Method Syntax Form(s)

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*.

1.17.179. StringSizeToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
stringsize_t sz;

// Method Syntax Form(s)

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.

1.17.180. *NEW* StringTableAt

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxStringTableObject strtab_obj;
RexxObjectPtr obj;
CSTRING str;
```

```
// Method Syntax Form(s)

obj = context->StringTableAt(strtab_obj, str);
```

Return the object at the specified index.

Arguments

strtab_obj	The source StringTable object.
str	The index into the StringTable object.

Returns

The object at the specified index. Returns NULLOBJECT if the given index does not exist.

See also methods *NEW* IsStringTable, *NEW* NewStringTable, *NEW* StringTablePut, and *NEW* StringTableRemove.

1.17.181. *NEW* StringTablePut

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxStringTableObject strtab_obj;
RexxObjectPtr item;
CSTRING index;

// Method Syntax Form(s)

context->StringTablePut(strtab_obj, item, index);
```

Replace/add an Object at the specified StringTable index.

Arguments

strtab_obj	The source StringTable object.
item	The object instance to be stored at the index.
index	The ASCII-Z string index into the StringTable object.

Returns

Void.

See also methods *NEW* IsStringTable, *NEW* NewStringTable, *NEW* StringTableAt, and *NEW* StringTableRemove.

1.17.182. *NEW* StringTableRemove

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxStringTableObject strtab_obj;
RexxObjectPtr obj;
CSTRING str;

// Method Syntax Form(s)
```

```
obj = context->StringTableRemove(strtab_obj, str);
```

Removes and returns the object at the specified StringTable index.

Arguments

strtab_obj   The source StringTable object.

str   The ASCII-Z index into the StringTable object.

Returns

The object removed at the specified index. Returns NULLOBJECT if the index did not exist in the target StringTable.

See also methods *NEW* IsStringTable, *NEW* NewStringTable, *NEW* StringTableAt, and *NEW* StringTablePut.

1.17.183. StringUpper

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr srcobj, newobj;

// Method Syntax Form(s)

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.

1.17.184. SupplierAvailable

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxSupplierObjectPtr sobj;
logical_t flag;

// Method Syntax Form(s)

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.

1.17.185. SupplierIndex

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxSupplierObjectPtr sobj;
RexxObjectPtr obj;

// Method Syntax Form(s)

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.

1.17.186. SupplierItem

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxSupplierObjectPtr sobj;
RexxObjectPtr obj;

// Method Syntax Form(s)

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.

1.17.187. SupplierNext

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxSupplierObjectPtr sobj;

// Method Syntax Form(s)

context->SupplierNext(sobj);
```

Advance a Supplier object to the next enumeration position.

Arguments

sobj   The source supplier object.

Returns

Void.

1.17.188. Terminate

This API is available in context *Instance*.

```
// Method Syntax Form(s)

context->Terminate();
```

Terminates the current Rexx interpreter instance. This call will wait for all threads to complete processing before returning.

Arguments

None.

Returns

Void.

1.17.189. *NEW* ThrowCondition

This API is available in contexts *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
CSTRING str;
RexxStringObject sobj;
RexxArrayObject arr;
RexxObjectPtr obj;

// Method Syntax Form(s)

context->ThrowCondition(str, sobj, add, obj);
```

Throw a condition. The API call doesn't return and the current method, routine, or exit is exited immediately.

Arguments

str   The condition name.

sobj   The optional condition description as a String object.

add   An 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.

See also methods *RaiseCondition* and **NEW* ThrowException/0/1/2*.

1.17.190. *NEW* ThrowException/0/1/2

This API is available in contexts *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
size_t n;
RexxObjectPtr array, obj1, obj2;

// Method Syntax Form(s)

context->ThrowException(n, array);

context->ThrowException0(n);

context->ThrowException1(n, obj1);

context->ThrowException2(n, obj1, obj2);
```

Throw a SYNTAX condition. The API call doesn't return and the current method, routine, or exit is exited immediately.

Arguments

n	The exception condition number. There are #defines for the recognized condition errors in the oorexxerrors.h include file.
array	An Array of error message substitution values.
obj1	The first substitution value for the error message.
obj2	The second substitution value for the error message.

Returns

Void.

See also methods *RaiseException/0/1/2*.

1.17.191. True

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;

// Method Syntax Form(s)

obj = context->True();
```

This method returns the Rexx .true object.

Arguments

None.

Returns

The Rexx .true object.

1.17.192. Uintptr

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;
uintptr_t n;

// Method Syntax Form(s)

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**.

1.17.193. UintptrToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
uintptr_t n;

// Method Syntax Form(s)

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.

1.17.194. UnsignedInt32

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;
uint32_t n;

// Method Syntax Form(s)

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*.

1.17.195. UnsignedInt32ToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
uint32_t n;

// Method Syntax Form(s)

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.

1.17.196. UnsignedInt64

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
logical_t flag;
uint64_t n;
```

```
// Method Syntax Form(s)

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*.

1.17.197. UnsignedInt64ToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
uint64_t n;

// Method Syntax Form(s)

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.

1.17.198. ValuesToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxArrayObj obj;
ValueDescriptor desc[3];

// Method Syntax Form(s)

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.

1.17.199. ValueToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
ValueDescriptor desc;;

// Method Syntax Form(s)

obj = context->ValueToObject(&desc);
```

Convert a type to an Object representation. The source type is identified by the ValueDescriptor structure, and can be 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.

1.17.200. *NEW* VariableReferenceName

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxVariableReferenceObject ref;
RexxStringObject name;

// Method Syntax Form(s)

name = context->VariableReferenceName(ref);
```

Returns the name of VariableReference object *ref*.

Arguments

ref | The VariableReference object instance.

Returns

The name of the VariableReference as a String.

See also methods *NEW* GetContextVariableReference, *NEW* GetObjectVariableReference, *NEW* IsVariableReference, *NEW* SetVariableReferenceValue, and *NEW* VariableReferenceValue.

1.17.201. *NEW* VariableReferenceValue

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit* since ooRexx 5.0.

```
RexxVariableReferenceObject ref;
RexxObjectPtr value;

// Method Syntax Form(s)

value = context->VariableReferenceValue(ref);
```

Returns the name of VariableReference object *ref*.

Arguments

 ref   The VariableReference object instance.

Returns

The value of the VariableReference.

See also methods **NEW* GetContextVariableReference*, **NEW* GetObjectVariableReference*, **NEW* IsVariableReference*, **NEW* SetVariableReferenceValue*, and **NEW* VariableReferenceName*.

1.17.202. WholeNumber

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
wholenumber_t n;
logical_t flag;

// Method Syntax Form(s)

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*.

1.17.203. WholeNumberToObject

This API is available in contexts *Thread*, *Method*, *Call*, and *Exit*.

```
RexxObjectPtr obj;
wholenumber_t n;

// Method Syntax Form(s)

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.

1.17.204. *NEW* WriteError

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
CSTRING data;
size_t  length;

// Method Syntax Form(s)

context->WriteError(data, length);
```

Adds a string to an error output redirection Rexx object that was specified using the WITH subkeyword of an ADDRESS instruction.

This API is a no-op if there is no error redirection.

Arguments

data   The string to be written.
length   The length of *data*.

Returns

Void.

See also methods **NEW* AreOutputAndErrorSameTarget*, **NEW* IsErrorRedirected*, **NEW* IsInputRedirected*, **NEW* IsOutputRedirected*, **NEW* IsRedirectionRequested*, **NEW* ReadInput*, **NEW* ReadInputBuffer*, **NEW* WriteErrorBuffer*, **NEW* WriteOutput*, and **NEW* WriteOutputBuffer*.

1.17.205. *NEW* WriteErrorBuffer

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
CSTRING data;
size_t  length;
```

```
// Method Syntax Form(s)

context->WriteErrorBuffer(data, length);
```

Adds a string composed of strings separated by the platform-specific line-end characters as separate items or lines to an error output redirection Rexx object that was specified using the WITH subkeyword of an ADDRESS instruction.

This API is a no-op if there is no error redirection.

Arguments

data　　　　　　The string of line-end separated strings to be written.

length　　　　　The length of *data*. *data* string.

Returns

Void.

See also methods **NEW* AreOutputAndErrorSameTarget*, **NEW* IsErrorRedirected*, **NEW* IsInputRedirected*, **NEW* IsOutputRedirected*, **NEW* IsRedirectionRequested*, **NEW* ReadInput*, **NEW* ReadInputBuffer*, **NEW* WriteError*, **NEW* WriteOutput*, and **NEW* WriteOutputBuffer*.

1.17.206. *NEW* WriteOutput

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
CSTRING data;
size_t  length;

// Method Syntax Form(s)

context->WriteOutput(data, length);
```

Adds a string to an output redirection Rexx object that was specified using the WITH subkeyword of an ADDRESS instruction.

This API is a no-op if there is no output redirection.

Arguments

data　　　　　　The string to be written.

length　　　　　The length of *data*.

Returns

Void.

See also methods **NEW* AreOutputAndErrorSameTarget*, **NEW* IsErrorRedirected*, **NEW* IsInputRedirected*, **NEW* IsOutputRedirected*, **NEW* IsRedirectionRequested*, **NEW* ReadInput*, **NEW* ReadInputBuffer*, **NEW* WriteError*, **NEW* WriteErrorBuffer*, **NEW* WriteOutputBuffer*.

1.17.207. *NEW* WriteOutputBuffer

This API is available in context *I/O Redirector* since ooRexx 5.0.

```
CSTRING data;
size_t  length;

// Method Syntax Form(s)

context->WriteOutputBuffer(data, length);
```

Adds a string composed of strings separated by the platform-specific line-end characters as separate items or lines to an output redirection Rexx object that was specified using the WITH subkeyword of an ADDRESS instruction.

This API is a no-op if there is no output redirection.

Arguments

data   The string of line-end separated strings to be written.

length   The length of *data*. *data* string.

Returns

Void.

See also methods *NEW* AreOutputAndErrorSameTarget, *NEW* IsErrorRedirected, *NEW* IsInputRedirected, *NEW* IsOutputRedirected, *NEW* IsRedirectionRequested, *NEW* ReadInput, *NEW* ReadInputBuffer, *NEW* WriteError, and *NEW* WriteOutput.

Classic Rexx Application Programming Interfaces

This chapter 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 chapter 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 2.7, "Variable Pool Interface"*), and execute Rexx routines directly from memory (see *Section 2.11, "Macrospace Interface"*).

2.1. 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.

2.2. 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 2.1. 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 2.2. 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.

2.3. Calling the Rexx Interpreter

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

2.3.1. From the Operating System

You can run a Rexx program directly from the operating system command prompt using Rexx followed by the program name.

2.3.2. 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.

2.3.3. The RexxStart Function

RexxStart calls the Rexx interpreter to run a Rexx procedure.

```
retc = RexxStart(ArgCount, ArgList, ProgramName, Instore, EnvName,
                 CallType, Exits, ReturnCode, Result);
```

2.3.3.1. 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 2.11, "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 2.11, "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 2.3. 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 2.6, "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*.

2.3.3.2. 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 2.11, "Macrospace Interface"*) is not loaded in the macrospace, the return code is -3 ("Program is unreadable").

2.3.3.3. Example

Example 2.4. 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.

2.3.4. 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.

2.3.5. The RexxDidRexxTerminate Function (Deprecated)

RexxDidRexxTerminate always returns 1 for 4.0. This is maintained for binary compatibility with early releases.

```
retc = RexxDidRexxTerminate();
```

2.4. 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.

2.4.1. 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.

2.4.1.1. Creating Subcommand Handlers

The following example is a sample subcommand handler definition.

Example 2.5. 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.

2.4.1.1.1. Example

Example 2.6. 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        */
}
```

2.4.2. Subcommand Interface Functions

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

2.4.2.1. RexxRegisterSubcomDll

RexxRegisterSubcomDll registers a subcommand handler that resides in a dynamic-link library routine.

```
retc = RexxRegisterSubcomDll(EnvName, ModuleName, EntryPoint,
                             UserArea, DropAuth);
```

2.4.2.1.1. 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.

2.4.2.1.2. 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.

2.4.2.2. RexxRegisterSubcomExe

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

```
retc = RexxRegisterSubcomExe(EnvName, EntryPoint, UserArea);
```

2.4.2.2.1. 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.

2.4.2.2.2. 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.

2.4.2.2.3. 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 2.7. 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.

2.4.2.2.4. Example

Example 2.8. 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        */
```

2.4.2.3. RexxDeregisterSubcom

RexxDeregisterSubcom deregisters a subcommand handler.

```
retc = RexxDeregisterSubcom(EnvName, ModuleName);
```

2.4.2.3.1. 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.

2.4.2.3.2. 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."

2.4.2.3.3. Remarks

The handler is removed from the active subcommand handler list.

2.4.2.4. RexxQuerySubcom

RexxQuerySubcom queries a subcommand handler and retrieves saved user information.

```
retc = RexxQuerySubcom(EnvName, ModuleName, Flag, UserWord);
```

2.4.2.4.1. 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.

2.4.2.4.2. 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).

2.4.2.4.3. Example

```
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     */
```

2.5. 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 Open 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 1.12, "Building an External Native Library"*.

2.5.1. 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.

2.5.1.1. Creating External Functions

The following is a sample external function definition:

Example 2.10. 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.

2.5.2. 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.

2.5.2.1. Example

Example 2.11. 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     */
}
```

2.5.3. External Function Interface Functions

The following sections explain the functions for registering and using external functions.

2.5.3.1. RexxRegisterFunctionDll

RexxRegisterFunctionDll registers an external function that resides in a dynamic-link library routine.

```
retc = RexxRegisterFunctionDll(FuncName, ModuleName, EntryPoint);
```

2.5.3.1.1. 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*.

2.5.3.1.2. Return Codes

RXFUNC_OK	0	The call to the function completed successfully.

RXFUNC_NOEMEM	1002	Memory allocation failure, or related.

2.5.3.1.3. Remarks

Starting with ooRexx 5.0.0, on Windows, External functions that reside in a dynamic-link library routine no longer require a module-definition (.DEF) file that lists the external functions in the EXPORT section.

A Rexx procedure can register dynamic-link library-external functions with the RXFUNCADD built-in function. For example:

Example 2.12. 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.

2.5.3.1.4. Example

Example 2.13. 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      */
}
```

2.5.3.2. RexxRegisterFunctionExe

RexxRegisterFunctionExe registers an external function that resides within the application code.

```
retc = RexxRegisterFunctionExe(FuncName, EntryPoint);
```

2.5.3.2.1. 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.

2.5.3.2.2. 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.

2.5.3.3. RexxDeregisterFunction

RexxDeregisterFunction deregisters an external function.

```
retc = RexxDeregisterFunction(FuncName);
```

2.5.3.3.1. Parameters

FuncName (const char *) - input
> is the address of an external function name to be deregistered.

2.5.3.3.2. Return Codes

RXFUNC_DEFINED	10	The requested function is already registered.
RXFUNC_NOTREG	30	The requested function is not registered.

2.5.3.4. RexxQueryFunction

RexxQueryFunction queries the existence of a registered external function.

```
retc = RexxQueryFunction(FuncName);
```

2.5.3.4.1. Parameters

FuncName (const char *) - input
> is the address of an external function name to be queried.

2.5.3.4.2. Return Codes

RXFUNC_OK	0	The call to the function completed successfully.
RXFUNC_NOTREG	30	The requested function is not registered.

2.5.3.4.3. 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.

2.6. 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.

2.6.1. Writing System Exit Handlers

The following is a sample exit handler definition:

Example 2.14. 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.

2.6.1.1. 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".

2.6.1.2. 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.

2.6.1.3. 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 2.15. 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.

2.6.1.3.1. Example

Example 2.16. 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       */
}
```

2.6.2. 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

2.6.2.1. 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 2.17. 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.

2.6.2.2. 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 2.18. 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.

2.6.2.3. 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 2.19. 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 2.20. 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 2.21. 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 2.22. RXMSQ parameter list

```
typedef struct {
   CONSTRXSTRING     rxmsq_name;       /* RXSTRING containing          */
                                       /* queue name.                  */
} RXMSQNAM_PARM;
```

rxmsq_name contains the new queue name.

2.6.2.4. 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 2.23. 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 2.24. 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 2.25. 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 2.26. RXSIO parameter list

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

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

2.6.2.5. 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 2.27. 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.

2.6.2.6. 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 2.28. 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.

2.6.2.7. 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.

2.6.2.8. 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.

2.6.3. System Exit Interface Functions

The system exit functions are similar to the subcommand handler functions. The system exit functions are:

2.6.3.1. RexxRegisterExitDll

RexxRegisterExitDll registers an exit handler that resides in a dynamic-link library routine.

```
retc = RexxRegisterExitDll(ExitName, ModuleName, EntryPoint,
                           UserArea, DropAuth);
```

2.6.3.1.1. 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.

2.6.3.1.2. 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.

2.6.3.2. RexxRegisterExitExe

RexxRegisterExitExe registers an exit handler that resides within the application code.

```
retc = RexxRegisterExitExe(ExitName, EntryPoint, UserArea);
```

2.6.3.2.1. 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.

2.6.3.2.2. 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.

2.6.3.2.3. 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.

2.6.3.2.4. Example

Example 2.29. 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         */
```

2.6.3.3. RexxDeregisterExit

RexxDeregisterExit deregisters an exit handler.

```
retc = RexxDeregisterExit(ExitName, ModuleName);
```

2.6.3.3.1. 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.

2.6.3.3.2. 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."

2.6.3.3.3. Remarks
The handler is removed from the exit handler list.

2.6.3.4. RexxQueryExit

RexxQueryExit queries an exit handler and retrieves saved user information.

```
retc = RexxQueryExit(ExitName, ModuleName, Flag, UserWord);
```

2.6.3.4.1. 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.

2.6.3.4.2. 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).

2.6.3.4.3. Example

Example 2.30. 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     */
...
}
```

2.7. Variable Pool Interface

Application programs can use the Rexx Variable Pool Interface to manipulate the variables of a
currently active Rexx procedure.

2.7.1. Interface Types

Three of the Variable Pool Interface functions (set, fetch, and drop) have dual interfaces.

2.7.1.1. 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.

2.7.1.2. 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.

2.7.2. 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.

2.7.3. RexxVariablePool Interface Function

Rexx procedure variables are accessed using the RexxVariablePool function.

2.7.3.1. RexxVariablePool

RexxVariablePool accesses variables of a currently active Rexx procedure.

```
retc = RexxVariablePool(RequestBlockList);
```

2.7.3.1.1. 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 2.31. 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.

2.7.3.1.2. 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.

2.7.3.1.3. Example

Example 2.32. 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           */
}
```

2.8. 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, Open Object Rexx comes with two API function calls called RexxAllocateMemory() and RexxFreeMemory(). These functions are wrappers for the corresponding compiler or operating system memory functions.

2.8.1. 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.

2.8.2. The RexxFreeMemory() Function

```
RexxReturnCode REXXENTRY RexxFreeMemory( void *MemoryBlock );
```

where:

MemoryBlock
> is a void pointer to the block of memory allocated by the ooRexx interpreter, or allocated by a previous call to RexxAllocateMemory().

Return Codes

RexxFreeMemory() always returns 0.

2.9. 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.

2.9.1. Queue Interface Functions

The following sections explain the functions for creating and using named queues.

2.9.1.1. RexxCreateQueue

RexxCreateQueue creates a new (empty) queue.

```
retc = RexxCreateQueue(Buffer, BuffLen, RequestedName, DupFlag);
```

2.9.1.1.1. 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.

2.9.1.1.2. 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.

2.9.1.1.3. Remarks

Queue names must conform to the same syntax rules as Rexx variable names. Lowercase characters in queue names are translated to uppercase.

2.9.1.2. 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);
```

2.9.1.2.1. 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.

CreatedFlag (size_t *) - output
    indicates whether RexxOpenQueue created the indicated queue. If zero on return, then the named queue already existed.

2.9.1.2.2. 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.

2.9.1.2.3. Remarks

Queue names must conform to the same syntax rules as Rexx variable names. Lowercase characters in queue names are translated to uppercase.

2.9.1.3. RexxDeleteQueue

RexxDeleteQueue deletes a queue.

```
retc = RexxDeleteQueue(QueueName);
```

2.9.1.3.1. Parameters

QueueName (const char *) - input
   is the address of the ASCII name of the queue to be deleted.

2.9.1.3.2. 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.

2.9.1.3.3. Remarks

If a queue is busy (for example, wait is active), it is not deleted.

2.9.1.4. RexxQueueExists

RexxQueueExists tests if name queue exists.

```
retc = RexxQueueExists(QueueName);
```

2.9.1.4.1. Parameters

QueueName (const char *) - input
   is the address of the ASCII name of the queue to be queried.

2.9.1.4.2. 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.

2.9.1.5. RexxQueryQueue

RexxQueryQueue returns the number of entries remaining in the named queue.

```
retc = RexxQueryQueue(QueueName, Count);
```

2.9.1.5.1. 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.

2.9.1.5.2. 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.

2.9.1.6. RexxAddQueue

RexxAddQueue adds an entry to a queue.

```
retc = RexxAddQueue(QueueName, EntryData, AddFlag);
```

2.9.1.6.1. 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).

2.9.1.6.2. 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.

2.9.1.7. RexxPullFromQueue

RexxPullFromQueue removes the top entry from the queue and returns it to the caller.

```
retc = RexxPullFromQueue(QueueName, DataBuf, DateTime, WaitFlag);
```

2.9.1.7.1. 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. The caller is responsible for releasing the
    RXSTRING storage with *RexxFreeMemory*(DataBuf->strptr).

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.

2.9.1.7.2. 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.

2.9.1.8. RexxClearQueue

RexxClearQueue clears all entries from a named queue.

```
retc = RexxClearQueue(QueueName);
```

2.9.1.8.1. Parameters

QueueName (const char *) - input
    is the address of the ASCII name of the queue to be cleared.

2.9.1.8.2. 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. |

2.9.1.9. 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*.

```
retc = RexxPullQueue(QueueName, DataBuf, DateTime, WaitFlag);
```

2.9.1.9.1. 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.

2.9.1.9.2. 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.

2.9.1.9.3. Remarks
The caller is responsible for freeing the returned memory that DataBuf points to.

2.10. 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.

2.10.1. Halt and Trace Interface Functions

The Halt and Trace functions are:

2.10.1.1. RexxSetHalt

RexxSetHalt raises a HALT condition in a running Rexx program.

```
retc = RexxSetHalt(ProcessId, ThreadId);
```

2.10.1.1.1. 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.

2.10.1.1.2. 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.

2.10.1.1.3. Remarks

This call is not processed if the target Rexx program is running with the RXHLT exit enabled.

2.10.1.2. RexxSetTrace

RexxSetTrace turns on the interactive debug mode for a Rexx procedure.

```
retc = RexxSetTrace(ProcessId, ThreadId);
```

2.10.1.2.1. 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.

2.10.1.2.2. 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.

2.10.1.2.3. Remarks

A RexxSetTrace call is not processed if the Rexx procedure is using the RXTRC exit.

2.10.1.3. RexxResetTrace

RexxResetTrace turns off the interactive debug mode for a Rexx procedure.

```
retc = RexxResetTrace(ProcessId,ThreadId);
```

2.10.1.3.1. 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.

2.10.1.3.2. 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.

2.10.1.3.3. 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.

2.11. 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.

2.11.1. 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.

2.11.2. 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.

2.11.3. Macrospace Interface Functions

The functions to manipulate macrospaces are:

2.11.3.1. RexxAddMacro

RexxAddMacro loads a Rexx procedure into the macrospace.

```
retc = RexxAddMacro(FuncName, SourceFile, Position);
```

2.11.3.1.1. 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.

2.11.3.1.2. 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.

2.11.3.2. RexxDropMacro

RexxDropMacro removes a Rexx procedure from the macrospace.

```
retc = RexxDropMacro(FuncName);
```

2.11.3.2.1. Parameter

FuncName (const char *) - input

> is the address of the ASCII function name.

2.11.3.2.2. 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.

2.11.3.3. RexxClearMacroSpace

RexxClearMacroSpace removes all loaded Rexx procedures from the macrospace.

```
retc = RexxClearMacroSpace();
```

2.11.3.3.1. 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.

2.11.3.3.2. Remarks

RexxClearMacroSpace must be used with care. This function removes all functions from the macrospace, including functions loaded by other processes.

2.11.3.4. RexxSaveMacroSpace

RexxSaveMacroSpace saves all or part of the macrospace Rexx procedures to a disk file.

```
retc = RexxSaveMacroSpace(FuncCount, FuncNames, MacroLibFile);
```

2.11.3.4.1. 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.

2.11.3.4.2. 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.

2.11.3.4.3. 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.

2.11.3.5. RexxLoadMacroSpace

RexxLoadMacroSpace loads all or part of the Rexx procedures from a saved macrospace file.

```
retc = RexxLoadMacroSpace(FuncCount, FuncNames, MacroLibFile);
```

2.11.3.5.1. 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.

2.11.3.5.2. 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.

2.11.3.5.3. 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.

2.11.3.6. RexxQueryMacro

RexxQueryMacro searches the macrospace for a specified function.

```
retc = RexxQueryMacro(FuncName, Position)
```

2.11.3.6.1. 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.

2.11.3.6.2. 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.

2.11.3.7. RexxReorderMacro

RexxReorderMacro changes the search order position of a loaded macrospace function.

```
retc = RexxReorderMacro(FuncName, Position)
```

2.11.3.7.1. 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.

2.11.3.7.2. 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.

Appendix A. 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.

A.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.

A.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 B, Common Public License Version 1.0*. The source code is available at *https://sourceforge.net/p/oorexx/code-0/HEAD/ tree/docs/*.

The source code for this document is maintained in DocBook SGML/XML format.

The railroad diagrams were generated with the help of "Railroad Diagram Generator" located at *http:// bottlecaps.de/rr/ui*. Special thanks to Gunther Rademacher for creating and maintaining this tool.

Appendix B. 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.

B.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.

B.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.

B.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.

B.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.

B.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.

B.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.

B.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 C. Revision History

Revision 0-0   Aug 2016

   Initial creation for 5.0

Index