The Regina Rexx Interpreter

Anders Christensen

<Anders.Christensen@idi.ntnu.no>
Norwegian Institute of Technology
University of Trondheim

April 29, 1998

Originally converted to Word by Ataman

Additions and corrections by Mark Hessling <M.Hessling@qut.edu.au>

Trademarks

Unix is a registered trademark of UNIX System Laboratories, Inc.

MS-DOS, Windows NT and **Windows 95** are registered trademarks of Microsoft Corporation.

IBM and **VM/CMS** are registered trademarks of International Business Machines Corporation

Amiga was a registered trademark of Commodore-Amiga Inc.

Table of Contents

Introduction to Regina

This chapter provides an introduction to Regina, a freeware Rexx Interpreter distributed under the GNU General Library License.

1. Purpose of this document

The purpose of this document is to provide an overview of the Rexx language and the Regina implementation of the Rexx language. It is not intended as a definitive reference to Rexx; you should really have a copy of the Rexx "bible"; *The Rexx Language*, by *Mike Cowlishaw* [TRL2].

2. Implementation

The Regina Rexx Interpreter is implemented as a library suitable for linking into third-party applications. Access to Regina from third-party applications is via the Regina API, which is consistent with the IBM's REXX SAA API. This API is implemented on most other Rexx interpreters.

The library containing Regina is available either as a static library or as a dynamically loadable library. The only functional difference between the two libraries is that the ability to dynamically load Rexx external function packages via the built-in function; RxFuncAdd, is only available with the dynamically loadable library.

The Regina distribution also includes a front end to the Regina library, to enable the execution of Rexx programs directly from the command line. The *command line* referred to here relates to the a Unix shell, an OS/2 or DOS command window or a Windows NT/95 command prompt.

3. Ports of Regina

Regina has been ported to most Unix operating systems, DOS, OS/2 and Windows NT/95. The following table provides implementation details of each of the ports of Regina.

Operating System	Dynamic Library	Static Library	Dynamic Executable	Static Executable
HP-UX	libregina.sl	libregina.a	regina	rexx
AIX	libregina_dlo.a	libregina.a	regina	rexx
Other Unix	libregina.so	libregina.a	regina	rexx
32-bit DOS (DJGPP)	N/A	libregna.a	N/A	rexx.exe
(Uses DPMI memory manager)				
32-bit DOS (EMX)	N/A	regina.a	N/A	rexx.exe
(Uses VCPI memory manager)				
OS/2 (EMX)	N/A	regina.a	N/A	rexx.exe
Windows NT/95	regina.dll (regina.lib)	rexx.lib	regina.exe	rexx.exe

4. Executing Rexx programs with Regina

Rexx programs are generally executed by Regina for the *command line* in the following manner:

regina [*switches*] [program] [*program parameters*]

where:

regina	is the name of the Regina executable (see table above)
switches	are optional switches. See the section below for an explanation of the switches currently supported by Regina
program	the name of the Rexx program to be executed. See the section **External Rexx Programs**, below, for details on how Regina interprets this argument. If no program name is specified, Regina waits for Rexx commands to be typed in and will execute those commands when the appropriate end-of-file character (^D on Unix and ^Z on DOS, OS/2 and Windows NT/95) is typed.
program parameters	any optional parameters to be passed to the Rexx program.

Rexx programs to be executed by Regina can take advantage of a feature of Unix shell programs called *magic numbers*. By having the first line of a Rexx program consist of the special sequence of **#!** followed by the full file name of the Regina executable, you can invoke this program simply by typing the name of the Rexx program on the *command line* followed by any parameters you wish to pass to the Rexx program. The file name must also have the appropriate execute bit set for this to work. As an example suppose your Rexx program, **myprog**, contained:

```
#!/usr/local/bin/regina
Parse Version ver
Say ver
```

When executing this program from the *command line* by typing **myprog**, the Unix shell program would execute the program `/usr/local/bin/regina` and pass the remainder of the lines in the file to this program via *stdin*.

The special processing done by Regina to find the file name in **REGINA_MACROS** and the file extension searching is not able to be carried out when using the magic number method of invocation.

4.1 Switches

The following switches allow the user to control how Regina executes the supplied Rexx program. Switches are recognised by a leading hyphen character; '-', followed immediately by a single alphabetic character. Some switches allow for optional parameters. These, too must follow the switch without any intervening spaces. All switches and their optional parameters are case-sensitive.

-t[*trace parameter*]	Turn on the specified tracing level. The optional *trace parameter* indicates the tracing level to be used. See the TRACE command later in this document for an explanation of each trace level.

4.2 External Rexx programs

Regina searches for Rexx programs, using a combination of the **REGINA_MACROS** environment variable and the addition of filename extensions. This rule applies to both external function calls and the **program** specified on the *command line*.

Assume you have a call to an external function, and it is coded as follows:

```
Call myextfunc arg1, arg2
```

First, Regina looks for a file called **myextfunc** in the current directory. If it can't find that file, it looks in each directory specified in the **REGINA_MACROS** environment variable for a file called **myextfunc**. If the file is not found, Regina then attempts to find a file called **myextfunc.rexx** in the current directory, then in each directory in **REGINA_MACROS**. Regina continues, next by appending **.rex** to the supplied external function name, and lastly by appending **.cmd**.

Only if a file does not exist in either the current directory, or any directory in **REGINA_MACROS,** either with the supplied filename or with that filename appended with **.rexx, .rex** or **.cmd** does Regina complain that the external function is unknown.

Rexx Language Constructs

In this chapter, the concept and syntax of REXX clauses are explained. At the end of the chapter there is a section describing how Regina differs from standard REXX are described in the first part of the chapter.

1. Definitions

A program in the REXX language consists of clauses, which are divided into four groups: null clauses, commands, assignments, and instructions. The three latter groups (commands, assignments, and instructions) are collectively referred to as statements. This does not match the terminology in [TRL2], where "instruction" is equivalent to what is known here as "statement", and "keyword instruction" is equivalent to what is known here as "instruction ". However, I find the terminology used here simpler and less confusing.

Incidentally, the terminology used here matches [DANEY].

A clause is defined as all non-clause-delimiters (i.e. blanks and tokens) up to and including a clause delimiter. A token delimiter can be:

- An end-of-line, unless it lies within a comment. An end-of-line within a constant string is considered a syntax error {6}.
- A semicolon character that is not within a comment or constant string.
- A colon character, provided that the sequence of tokens leading up to it consists of a single symbol and whitespace. If a sequence of two symbol tokens is followed by a colon, then this implies SYNTAX condition {13}.

Some systems have the ability to store a text file having a last line unterminated by an end-of-line character sequence. In general, this applies to systems that use an explicit end-of-line character sequence to denote end-of-lines, e.g. Unix and MS-DOS systems. Under these systems, if the last line is unterminated, it will strictly speaking not be a clause, since a clause must include its terminating clause delimiter. However, some interpreters are likely to regard the end-of-file as a clause delimiter too. The functionality of INTERPRET gives some weight to this interpretation. But other systems may ignore that last, unterminated line, or maybe issue a syntax error. (However, there is no SYNTAX condition number adequately covering this situation.

Example: Binary transferring files

Suppose a REXX program is stored on an MS-DOS machine. Then, an end-of-line sequence is marked in the file as the two characters carriage return and newline. If this file is transferred to a Unix system, then only newline marks the end-of-line. For this to work, the file must be transferred as a text file. If it is (incorrectly) transferred as a binary file, the result is that on the Unix system, each line seems to contain a trailing carriage return character. In an editor, it might look like this:

```
say 'hello world'^M
say 'that"s it'^M
```

This will probably raise SYNTAX condition {13}.

2. Null clauses

Null clauses are clauses that consist of only whitespace, or comments, or both; in addition to the terminating clause delimiter. These clauses are ignored when interpreting the code, except for one situation: null clauses containing at least one comment is traced when appropriate. Null clauses not containing any comments are ignored in every respect.

Example: Tracing comments

The tracing of comments may be a major problem, depending on the context. There are basically two strategies for large comments: either box multiple lines as a single comment, or make the text on each line an independent comment, as shown below:

```
trace all

/*
  This is a single, large comment, which spans multiple lines.
  Such comments are often used at the start of a subroutine or
  similar, in order to describe both the interface to and the
  functionality of the function.
*/


/* This is also a large comment, but it is written as multiple   */
/* comments, each on its own line. Thus, this is several clauses */
/* while the comment above is a single comment.                  */
```

During tracing, the first of these will be displayed as one large comment, and during interactive tracing, it will only pause once. The second will be displayed as multiple lines, and will make several pauses during interactive tracing. An interpreter may solve this situation in several ways, the main objective must be to display the comments nicely the to programmer debugging the code. Preferably, the code is shown in a fashion that resembles how it is entered in the file.

If a label is multiple defined, the first definition is used and the rest are ignored. Multiple defined labels is not an `SYNTAX` condition.

A null clause is not a statement. In some situations, like after the `THEN` subclause, only a statement come. If a null clause is provided, then it is ignored, and the next statement is used instead.

Consider the following code:

```
parse pull foo

if foo=2 then
    say ‘foo is not 2’
else
    /* do nothing */
say ‘that”s it’
```

This will not work the way indentation indicates, since the comment in this example is not a statement. Thus, the `ELSE` reads beyond the comment, and connects to the `SAY` instruction which becomes the `ELSE` part. (That what probably not what the programmer intended.) This code will say `that’s it`, only when `foo` is different from `2`. A separate instruction, `NOP` has been provided in order to fill the need that was inadequately attempted filled by the comment in the code fragment above.

Example: Trailing comments

The effect that comments are not statements can be exploited when documenting the program, and simultaneously making the program faster. Consider the following two loops:

```
sum = 0
do i=1 to 10
/* sum 1 2 3 ... 8 9 10 */
      sum = sum + i
end

sum = 0
do i=1 to 10
      sum = sum + i   /* sum 1 2 3 ... 8 9 10 */
end
```

In the first loop, there are two clauses, while the second loop contains only one clause, because the comment is appended to an already existing clause. During execution, the interpreter has to spend time ignoring the null clause in the first loop, while the second loop avoids this problem (assuming tracing is unenabled). Thus, the second loop is faster; although only insignificantly faster for small loops. Of course, the comment could have been taken out of the loop, which would be equally fast to the second version above.

3. Commands

3.1 Assignments

Assignments are clauses where the first token is a symbol and the second token is the equal sign (=). This definition opens for some curious effects, consider the following clauses:

`a == b`
: This is not a command, but an assignment of the expression `= b` to the variable `a`. Of course, the expression is illegal (`=b`) and will trigger a `SYNTAX` condition for syntax error {35}. TRL2 defines the operator `==` as consisting of two tokens. Thus, in the first of these examples, the second token is `=`, the third token is also `=`, while the fourth token is `b`.

`3 = 5`
: This is an assignment of the value `5` to the symbol `3`, but since this is not a variable symbol, this is an illegal assignment, and will trigger the `SYNTAX` condition for syntax error {31}.

`"hello " = foo`
: This is not an invalid assignment, since the first token in the clause is not a symbol. Instead, this becomes a command.

`arg =(foo) bar`
: The fourth statement is a valid assignment, which will space-concatenate the two variable symbols `foo` and `bar`, and assign the result to the variable symbol `arg`. It is specifically not an `ARG` instruction, even though it might look like one. If you need an `ARG` instruction which template starts with an absolute indirect positional pattern, use the `PARSE UPPER ARG` instruction instead, or prepend a dot in front of the template.

An assignment can assign a value to a simple variable, a stem variable or a compound variable. When assigning to a stem variable, all possible variable symbols having that stem are assigned the value. Note specifically that this is not like setting a default, it is a one time multiple assignment.

Example: Multiple assignment

The difference between REXX's multiple assignment and a default value can be seen from the following code:

```
foo. = 'bar'
foo.1 = 'baz'
drop foo.1
say foo.1        /* says "FOO.1" */
```

Here, the SAY instruction writes out FOO.1, not bar. During the DROP instruction, the variable FOO.1 regains its original, uninitialized value FOO.1, not the value of its stem variable FOO., i.e. bar, because stem assignments does not set up a default.

Example: Emulating a default value

If you want to set the compound variable to the value of its stem variable, if the stem is initialized, then you may use the following code:

```
if (symbol('foo.')) then
      foo.1 = foo.
else
      drop foo.1
```

In this example, the FOO.1 variable is set to the value of its stem if the stem currently is assigned a value. Else, the FOO.1 variable is dropped.

However, this is probably not exactly the same, since the internal storage of the computer is likely to store variables like FOO.2 and FOO.3 only implicitly (after all, it can not explicitly store every compound having FOO. as stem). After the assignment of the value of FOO. to FOO.1, the FOO.1 compound variable is likely to be explicitly stored in the interpreter.

There is no way you can discover the difference, but the effects are often that more memory is used, and some functionality that dumps all variables may dump FOO.1 but not FOO.2 (which is inconsistent). See section RexxVariablePool.

Example: Space considerations

Even more strange are the effects of the following short example:

```
foo. = 'bar'
drop foo.1
```

Although apparently very simple, there is no way that an interpreter can release all memory referring to FOO.1. After all, FOO.1 has a different value than FOO.2, FOO.3, etc., so the interpreter must store information that tells it that FOO.1 has the uninitialized value.

These considerations may seem like nit-picking, but they will matter if you drop lots of compound variables for a stem which has previously received a value. Some programming idioms do this, so be aware. If you can do without assigning to the stem variable, then it is possible for the interpreter to regain all memory used for that stem's compound variables.

4. Instructions

In this section, all instructions in standard REXX are described.

Extensions are listed later in this chapter.

First some notes on the terminology. What is called an instruction in this document is equivalent to a "unit " of clauses. That is, each instruction can consist of one or more clauses. For instance, the SAY instruction is always a single

instruction, but the IF instruction is a multi-clause instruction. Consider the following script, where each clause has been boxed:

```
if a=b then
      say 'hello'
else
      say 'bye'
```

Further, the THEN or ELSE parts of this instruction might consist of a DO/END pair, in which case the IF instruction might consists of an virtually unlimited number of clauses.

Then, some notes on the syntax diagrams used in the following descriptions of the instructions. The rules applying to these diagrams can be listed as:

- Anything written in courier font in the syntax diagrams indicates that it should occur as-is in the REXX program. Whenever something is written in *italic* font, it means that the term should be substituted for another value, expression, or terms.

- Anything contained within matching pairs of square brackets ([...]) are optional, and may be left out.

- Whenever a pair or curly braces is used, it contains two or more subclauses that are separated by the vertical bar (|). It means that the curly braces will be substituted for one of the subclauses it contains.

- Whenever the ellipsis (...) is used, it indicates that the immediately following subclauses may be repeated zero or more times. The scope of the ellipsis is limited to the contents of a set of square brackets or curly braces, if it occurs there.

- Whenever the vertical bar | is used in any of the syntax diagrams, it means that either the term to the left, or the term to the right can be used, but not both, and at least one of the must be used. This "operator " is associative (can be used in sequence), and it has lower priority than the square brackets (the scope of the vertical bar located within a pair of square brackets or curly braces is limited to the text within those square brackets or curly braces.

- Whenever a semicolon (;) is used in the syntax diagram, it indicates that a clause separator must be present at the point. It may either be a semicolon character, or an end-of-line.

- Whenever the syntax diagram is spread out over more lines, it means that any of the lines can be used, but that the individual lines are mutually exclusive. Consider the syntax:

  ```
  SAY = symbol
        string
  ```

  This is equivalent to the syntax:

  ```
  SAY [symbol | string ]
  ```

  Because in the first of these two syntaxes, the SAY part may be continued at either line.

- Sometimes the syntax of an instruction is so complex that parts of the syntax has been extracted, and is shown below in its expanded state. The following is an example of how this looks:

  ```
  SAY something TO someone

  something : = HI
        HELLO
        BYE

  someone : = THE BOSS
        YOUR NEIGHBOR
  ```

  You can generally identify these situations by the fact that they comes a bit below the real syntax diagram, and that they contains a colon character after the name of the term to be expanded.

In the syntax diagrams, some generic names have been used for the various parts, in order to indicate common attributes for the term. For instance, whenever a term in the syntax diagrams is called *expr*, it means that any valid REXX expression may occur instead of that term. The most common such names are:

condition
: Indicates that the subclause can be any of the names of the conditions, e.g. SYNTAX, NOVALUE, HALT, etc.

expr
: Indicates that the subclause can be any valid REXX expression, and will in general be evaluated as normal during execution.

statement
: Indicates that extra clauses may be inserted into the instruction, and that exactly one of them must be a true statement.

string
: Indicates that the subclause is a constant string, i.e. either enclosed by single quotes ('...') or double quotes ("...").

symbol
: Indicates that the subclause is a single symbol. In general, whenever *symbol* is used as the name for a subclause, it means that the symbol will not automatically be expanded to the value of the symbol. But instead, some operation is performed on the name of the symbol.

template
: Indicates that the subclause is a parsing template. The exact syntax of this is explain in a chapter on tracing, to be written later.

In addition to this, variants may also exists. These variants will have an extra letter or number appended to the name of the subclause, and is used for differing between two or more subclauses having the same "type" in one syntax diagram. In the case of other names for the subclauses, these are explained in the description of the instruction.

4.1 The ADDRESS Instruction

```
ADDRESS [ environment  [ command ] ] ;
        [ [ VALUE ] expression ] ;
```

The ADDRESS instruction controls where commands to external environment are sent. If both *environment* and *command* are specified, the given command will be executed in the given environment. The effect is the same as issuing an expression to be executed as a command (see section Commands), except that the environment in which it is to be executed can be explicitly specified in the ADDRESS clause. In this case, the special variable RC will be set as usual, and the ERROR or FAILURE conditions might be raised, as for normal commands.

The *environment* term must be a symbol or a literal string. If it is a symbol, its "name" is used, i.e. it is not tail substituted or swapped for a variable value. The *command* and *expression* terms can be any REXX expression.

REXX maintains a list of environments, the size of this list is at least two. If you select a new environment, it will be put in the front of this list, possibly squeezing the backend environment out of the list. Note that if *command* is specified, the contents of the environment stack is not changed. If you omit *command*, *environment* will always be put in the front of the list of environments.

What happens if you specify an environment that is already in the list, is not completely defined. Strictly speaking, you should end up with both entries in the list pointing to the same environment, but some implementations will probably handle this by reordering the list, leaving the selected environment in the front.

If you do not specify any subkeywords or parameters toADDRESS, the effect is to swap the two first entries in the list of environments. Consequently, executing ADDRESS multiple times will toggle between two environments.

The second syntax form of ADDRESS is a special case of the first form with *command* omitted. If the first token after ADDRESS is VALUE, then the rest of the clause is taken to be an expression, naming the environment which is to be made the current environment. Using VALUE makes it possible to circumvent the restriction that the name of the new

environment must be a symbol or literal string. However, you can not combine both VALUE and *command* in a single clause.

Example: Examples of the ADDRESS instruction

You can omit the VALUE keyword if the expression following ADDRESS starts with a token which is neither a symbol or a literal string. Confused? Let's look at some examples:

```
ADDRESS COMMAND

ADDRESS SYSTEM 'copy' fromfile tofile

ADDRESS system

ADDRESS VALUE newenv

ADDRESS

ADDRESS (oldenv)
```

The first of these sets the environment COMMAND as the current environment. The second performs the command copy in the environment SYSTEM, using the values of the symbols fromfile and tofile as parameters. Note that this will not set SYSTEM as current environment. The third example sets SYSTEM as current environment (it will be automatically converted to upper case). The fourth example sets as the current environment the contents of the symbol newenv, pushing SYSTEM down one level in the stack. The fifth clause swap the two uppermost entries on the stack; and SYSTEM ends up at the top. The last example sets the current environment to whatever is the value of the symbol oldenv.

Example: The VALUE subkeyword

Let us look a bit closer at the last example. Note the differences between the two clauses:

```
ADDRESS OLDENV

ADDRESS (OLDENV)
```

The first of these sets the current default environment to OLDENV, while the second sets it to the value of the symbol OLDENV. Actually, in the latter, the subkeyword VALUE has been omitted, which is legal since the parameter starts with a special character.

If you are still confused, Don't Panic; the syntax of ADDRESS is somewhat bizarre, and you should not put too much effort into learning all aspects of it. Just make sure that you understand how to use it in simple situations. Chances are that you will not have use for its more complicated variants for quite some time.

Then, what names are legal as environments? Well, that is implementation-specific, but some names seems to be in common use. The name COMMAND is sometimes used to refer to an environment that sends the command to the operating system. Likewise, the name of the operating system is often used for this (CMS, UNIX, etc.). You have to consult the implementation specific documentation for more information about this. Actually, there is not really any restrictions on what constitutes a legal environment name (even the nullstring is legal). Some interpreters will allow you to select anything as current environment; and if it is an illegal name, the interpreter will complain only when the environment is actually tried used. Other implementations may not allow you to select an invalid environment name at all.

Nor does the definition of REXX say anything about which environment is preselected when you invoke the interpreter, although TRL defines that one environment is automatically preselected when starting up a REXX script. Note that there does not exist any NONE environment in standard REXX, i.e. an environment that ignores commands. But some interpreters implement the TRACE setting ??? which accomplish this.

The list of environments will be saved across subroutine calls; so the effect of any ADDRESS clauses in the subroutine will cease upon return from the subroutine.

4.2 The ARG Instruction

```
ARG [ template ] ;
```

The ARG instruction will parse the argument strings at the current procedural level into the template. Parsing will be performed in upper case mode. This clause is equivalent to:

```
PARSE UPPER ARG [ template ] ;
```

For more information, see the PARSE instruction. Note that this is the only situation where a multistring template is relevant.

Example: Beware assignments

The similarity between ARG and PARSE UPPER ARG has one exception. Suppose the PARSE UPPER ARG has an absolute positional pattern as the first element in the template, like:

```
parse upper arg =(foo) bar
```

This is not equivalent to an ARG instruction, because ARG instruction would become an assignment. A simple trick to avoid this problem is just to prepend a placeholder period (.) to the pattern, thus the equal sign (=) is no longer the second token in the new ARG instruction. Also, unless the absolute positional pattern is indirect, the equal sign can be removed without changing the meaning of the statement.

4.3 The CALL Instruction

```
CALL = routine [ parameter ]
     [, [ parameter ] ... ] ;
     { ON | OFF } condition [ NAME label ] ;
```

The CALL instruction invokes a subroutine, named by *routine*, which can be internal, built-in, or external; and the three repositories of functions are searched in that order. are searched for *routine* in that order. The token *routine* must be either a literal string or a symbol (which is taken literally). However, if *routine* is a literal string, the pool of internal subroutines is not searched. Note that some interpreters may have additional repositories of labels to search.

In a CALL instruction, each *parameter* is evaluated, strictly in order from left to right, and passed as an argument to the subroutine. A *parameter* might be left out (i.e. an empty argument), which is not the same as passing the nullstring as argument.

Users often confuse a parameter which is the nullstring with leaving out the parameter. However, this is two very different situations. Consider the following calls to the built-in function TRANSLATE( ):

```
say translate('abcDEF'   )   /* says ABCDEF */

say translate('abcDEF',"")   /* says abcDEF */

say translate('abcDEF',,"")  /* says '      ' */
```

The TRANSLATE( ) function is able to differ between receiving the nullstring (i.e. a defined string having zero length), from the situation where a parameter was not specified (i.e. the undefined string). Since TRANSLATE( ) is one of the few functions where the parameters' default values are very different from the nullstring, the distinction becomes very visible.

For the CALL instruction, watch out for interference with line continuation. If there are trailing commas, it might be interpreted as line continuation. If a CALL instruction use line continuation between two parameters, two commas are needed: one to separate the parameters, and one to denote line continuation.

A number of settings are stored across internal subroutine calls. An internal subroutine will inherit the values in effect when the call is made, and the settings are restored on exit from the subroutine. These settings are:

- Conditions traps, see chapter Conditions.
- Current trapped condition, see section CTS.
- NUMERIC settings, see section Numeric.
- ADDRESS environments, see section Address.
- TRACE mode, see section Trace and chapter [not yet written].
- The elapse time clock, see section Time.

Also, the OPTIONS settings may or may not be restored, depending on the implementation. Further, a number of other things may be saved across internal subroutines. The effect on variables are controlled by the PROCEDURE instruction in the subroutine itself. The state of all DO-loops will be preserved during subroutine calls.

Example: Subroutines and trace settings

Subroutines can not be used to set various settings like trace settings, NUMERIC settings, etc. Thus, the following code will not work as intended:

```
say digits() /* says 9, maybe */
call inc_digits
say digits() /* still says 9 */
exit

inc_digits:
    numeric digits digits() + 1
    return
```

The programmer probably wanted to call a routine which incremented the precision of arithmetic operations. However, since the setting of NUMERIC DIGITS is saved across subroutine calls, the new value set in inc_digits is lost at return from that routine. Thus, in order to work correctly, the NUMERIC instruction must be located in the main routine itself.

Built-in subroutines will have no effect on the settings, except for explicitly defined side effects. Nor will external subroutines change the settings. For all practical purposes, an external subroutine is conceptually equivalent to reinvoking the interpreter in a totally separated process.

If the name of the subroutine is specified by a literal string, then the name will be used as-is; it will not be converted to upper case. This is important because a routine which contains lower case letters can only be invoked by using a literal string as the routine name in the CALL instruction.

Example: Labels are literals

Labels are literal, which means that they are neither tail-substituted nor substituted for the value of the variable. Further, this also means that the setting of NUMERIC DIGITS has no influence on the section of labels, even when the labels are numeric symbols. Consider the following code:

```
call 654.32
exit

654.321:
    say here
    return

654.32:
    say there
    return
```

In this example, the second of the two subroutines are always chosen, independent of the setting of NUMERIC DIGITS. Assuming that NUMERIC DIGITS are set to 5, then the number 654.321 is converted to 654.32, but that does not affect labels. Nor would a statement CALL 6.5432E2 call the second label, even though the numeric value of that symbol is equal to that of one of the labels.

The called subroutines may or may not return data to the caller. In the calling routine, the special variable RESULT will be set to the return value or dropped, depending on whether any data was returned or not. Thus, the CALL instruction is equivalent to calling the routine as a function, and assigning the return value to RESULT, except when the *routine* does not return data.

In REXX, recursive routines are allowed. A minimum number of 100 nested internal and external subroutine invocations, and support for a minimum of 10 parameters for each call are required by REXX. See chapter Limits for more information concerning implementation limits.

When the token following CALL is either ON or OFF, the CALL instruction is not used for calling a subroutine, but for setting up condition traps. In this case, the third token of the clause must be the name of a condition, which setup is to be changed.

If the second token was ON, then there can be either three or five tokens. If the five token version is used, then the fourth token must be NAME and the fifth token is taken to be the symbolic name of a label, which is the condition handler. This name can be either a constant string, or a symbol, which is taken literally. When OFF is used, the named condition trap is turned off.

Note that the ON and OFF forms of the CALL instruction were introduced in TRL2. Thus, they are not likely to be present on older interpreters. More information about conditions and condition traps are given in a chapter Conditions.

4.4 The DO/END Instruction

```
DO [ repetitor ] [ conditional ] ;
     [ clauses ]
     END [ symbol ] ;

repetitor := symbol = expri [ TO exprt ]
     [ BY exprb ] [ FOR exprf ]
     exprr
     FOREVER

conditional := WHILE exprw
     UNTIL expru
```

The DO/END instruction is the instruction used for looping and grouping several statements into one block. This is a multi-clause instruction.

The most simple case is when there is no *repetitor* or *conditional*, in which case it works like BEGIN/END in Pascal or {...} in C. I.e. it groups zero or more REXX clauses into one conceptual statement.

The *repetitor* subclause controls the control variable of the loop, or the number of repetitions. The *exprr* subclause may specify a certain number of repetitions, or you may use FOREVER to go on looping forever.

If you specify the control variable *symbol*, it must be a variable symbol, and it will get the initial value *expri* at the start of the loop. At the start of each iteration, including the first, it will be checked whether it has reached the value specified by *exprt*. At the end of each iteration the value *exprb* is added to the control variable. The loop will terminate after at most *exprf* iterations. Note that all these expressions are evaluated only once, before the loop is entered for the first iteration.

You may also specify UNTIL or WHILE, which take a boolean expression. WHILE is checked before each iteration, immediately after the maximum number of iteration has been performed. UNTIL is checked after each iteration, immediately before the control variable is incremented. It is not possible to specify both UNTIL and WHILE in the same DO instruction.

The FOREVER keyword is only needed when there is no *conditional*, and the *repetitor* would also be empty if FOREVER was not specified. Actually, you could rewrite this as DO WHILE 1. The two forms are equivalent, except for tracing output.

The subclauses TO, BY, and FOR may come in any order, and their expressions are evaluated in the order in which they occur. However, the initial assignment must always come first. Their order may affect your program if these expressions have any side effects. However, this is seldom a problem, since it is quite intuitive. Note that the counting of iterations, if the FOR subclause has been specified, is never affected by the setting of NUMERIC DIGITS.

Example: Evaluation order

What may prove a real trap, is that although the value to which the control variable is set is evaluated before any other expressions in the *repetitor*, it is assigned to the control variable after all expressions in the *repetitor* have been evaluated.

The following code illustrates this problem:

```
ctrl = 1
do ctrl=f(2) by f(3) to f(5)
      call func(6)
end
call func(7)
exit

f:
    say 'ctrl='ctrl 'arg='arg(1)
    return arg(1)
```

This code produces the following output:

```
ctrl=1 arg=2
ctrl=1 arg=3
ctrl=1 arg=5
ctrl=2 arg=6
ctrl=5 arg=6
ctrl=8 arg=7
```

Make sure you understand why the program produces this output. Failure to understand this may give you a surprise later, when you happen to write a complex DO-instruction, and do not get the expected result.

If the TO expression is omitted, there is no checking for an upper bound of the expression. If the BY subclause is omitted, then the default increment of 1 is used. If the FOR subclause is omitted, then there is no checking for a maximum number of iterations.

Example: Loop convergence For the reasons just explained, the instruction:

```
do ctrl=1
      nop  /* and other statements */
end
```

will start with CTRL being 1, and then iterate through 2, 3, 4, ..., and never terminate except by LEAVE, RETURN, SIGNAL, or EXIT.

Although similar constructs in other languages typically provokes an overflow at some point, something "strange" happens in REXX. Whenever the value of ctrl becomes too large, the incrementation of that variable produces a result that is identical to the old value of ctrl. For NUMERIC DIGITS set to 9, this happens when ctrl becomes 1.00000000E+9. When adding 1 to this number, the result is still 1.00000000E+9. Thus, the loop "converges" at that value.

If the value of NUMERIC DIGITS is 1, then it will "converge" at 10, or 1E+1 which is the "correct" way of writing that number under NUMERIC DIGITS 1. You can in general disregard loop "convergence", because it will only occur in very rare situations.

Example: Difference between UNTIL and WHILE

One frequent misunderstanding is that the WHILE and UNTIL subclauses of the DO/END instruction are equivalent, except that WHILE is checked before the first iteration, while UNTIL is first checked before the second iteration.

This may be so in other languages, but in REXX. Because of the order in which the parts of the loop are performed, there are other differences. Consider the following code:

```
count = 1
do i=1 while count  \= 5
      count = count + 1
end
say i count

count = 1
do i=1 until count=5
      count = count + 1
end
say i count
```

After the first loop, the numbers 6 and 5, while in the second loop, the numbers 5 and 5 are written out. The reason is that a WHILE clause is checked after the control variable of the loop has been incremented, but an UNTIL expression is checked before the incrementation.

A loop can be terminated in several ways. A RETURN or EXIT instruction terminates all active loops in the procedure levels terminated. Further, a SIGNAL instruction transferring control (i.e. neither a SIGNAL ON nor SIGNAL OFF) terminates all loops at the current procedural level. This applies even to "implicit" SIGNAL instructions, i.e. when triggering a condition handler by the method of SIGNAL. A LEAVE instruction terminates one or more loops. Last but not least, a loop can terminate itself, when it has reached its specified stop conditions.

Note that the SIGNAL instruction terminates also non-repetitive loops (or rather: DO/END pairs), thus after an SIGNAL instruction, you must not execute an END instruction without having executed its corresponding DO first (and after the SIGNAL instruction). However, as long as you stay away from the ENDs, it is all right according to TRL to execute code within a loop without having properly activated the loop itself.

Note that on exit from a loop, the value of the control variable has been incremented once after the last iteration of the loop, if the loop was terminated by the WHILE expression, by exceeding the number of max iterations, or if the control variable exceeded the stop value. However, the control variable has the value of the last iteration if the loop was terminated by the UNTIL expression, or by an instruction inside the loop (e.g. LEAVE, SIGNAL, etc.).

The following algorithm in REXX code shows the execution of a DO instruction, assuming that *expri*, *exprt*, *exprb*, *exprf*, *exprw*, *expru*, and *symbol* have been taken from the syntax diagram of DO.

```
@expri = expri
@exprt = exprt
@exprb = exprb
@exprf = exprf
@iters = 0

symbol = @expri

start_of_loop:
      if symbol > @extrt then signal after_loop
      if @iters > @exprf then signal after_loop
      if exprw then signal after_loop
            instructions
end_of_loop:
```

```
      if \expru then signal after_loop
      symbol = symbol + @exprb
      signal start_of_loop

after_loop:
```

Some notes are in order for this algorithm. First, it uses the SIGNAL instruction, which is defined to terminate all active loops. This aspect of the SIGNAL instruction has been ignored for the purpose of illustrating the DO, and consequently, the code shown above is not suitable for nested loops. Further, the order of the first four statements should be identical to the order in the corresponding subclauses in the *repetitor*. The code has also ignored that the WHILE and the UNTIL subclauses can not be used in the same DO instruction. And in addition, all variables starting with the at sign (@), are assumed to be internal variables, private to this particular loop. Within *instructions*, a LEAVE instruction is equivalent to `signal after_loop`, while a ITERATE instruction is equivalent to `signal end_of_loop`.

4.5 The DROP Instruction

```
DROP symbol [ symbol ... ] ;
```

The DROP instruction makes the named *variable*s uninitialized, i.e. the same state that they had at the startup of the program. The list of variable names are processed strictly from left to right and dropped in that order. Consequently, if one of the variables to be dropped is used in a tail of another, then the order might be significant. E.g. the following two DROP instructions are not equivalent:

```
bar = 'a'
drop bar foo.bar  /* drops 'BAR' and 'FOO.BAR' */
bar = 'a'
drop foo.bar bar  /* drops 'FOO.a' and 'BAR'
```

The *variable* terms can be either a variable symbol or a symbol enclosed in parentheses. The former form is first tail-substituted, and then taken as the literal name of the symbol to be dropped. The result names the variable to drop. In the latter form, the value of the variable symbol inside the parentheses is retrieved and taken as a space separated list of symbols. Each of these symbols is tail-substituted (if relevant); and the result is taken as the literal name of a variable to be dropped. However, this process is not recursive, so that the list of names referred to indirectly can not itself contain parentheses. Note that the second form was introduced in TRL2, mainly in order to make INTERPRET unnecessary.

In general, things contained in parentheses can be any valid REXX expression, but this does not apply to the DROP, PARSE, and PROCEDURE instructions.

Example: Dropping compound variables

Note a potential problem for compound variables: when a stem variable is set, it will not set a default value, rather it will assign "all possible variables" in that stem collection at once. So dropping a compound variable in a stem collection for which the stem variable has been set, will set that compound variable to the original uninitialized value; not the value of the stem variable. See section Assign for further notes on assignments. To illustrate consider the code:

```
foo. = 'default'
drop baz bar foo.bar
say foo.bar foo.baz /* says 'FOO.BAR default' */
```

In this example, the SAY instruction writes out the value of the two compound variables FOO.BAR and FOO.BAZ. When performing tail-substitution for these, the interpreter finds that both BAR and BAZ are uninitialized. Further, FOO.BAR has also been made uninitialized, while FOO.BAZ has the value assigned to it in the assignment to the stem variable.

Example: Tail-substitution in DROP

For instance, suppose that the variable FOO has the value bar. After being dropped, FOO will have its uninitialized value, which is the same as its name: FOO. If the variable to be dropped is a stem variable, then both the stem variable and all compound variables of that stem become uninitialized.

```
bar = 123
drop foo.bar   /* drops 'FOO.123' */
```

Technically, it should be noted that some operations involving dropping of compound variables can be very space consuming. Even though the standard does not operate with the term "default value" for the value assigned to a stem variable, that is the way in which it is most likely to be implemented. When a stem is assigned a value, and some of its compound variables are dropped afterwards, then the interpreter must use memory to store references to the variables dropped. This might seem counterintuitive at first, since dropping ought to release memory, not allocate more.

There is a parallel between DROP and PROCEDURE EXPOSE. However, there is one important difference, although PROCEDURE EXPOSE will expose the name of a variable enclosed in parentheses before starting to expose the symbols that variable refers to, this is not so for DROP. If DROP had mimicked the behavior of PROCEDURE EXPOSE in this matter, then the whole purpose of indirect specifying of variables in DROP would have been defeated.

Dropping a variable which does not have a value is not an error. There is no upper limit on the number of variables that can be dropped in one DROP clause, other than restrictions on the clause length. If an exposed variable is dropped, the variable in the caller is dropped, but the variable remains exposed. If it reassigned a value, the value is assigned to a variable in the caller routine.

4.6 The EXIT Instruction

```
EXIT [ expr ] ;
```

Terminates the REXX program, and optionally returns the expression *expr* to the caller. If specified, *expr* can be any string. In some systems, there are restrictions on the range of valid values for the *expr*. Often the return expression must be an integer, or even a non-negative integer. This is not really a restriction on the REXX language itself, but a restriction in the environment in which the interpreter operates, check the system dependent documentation for more information.

If *expr* is omitted, nothing will be returned to the caller. Under some circumstances that is not legal, and might be handled as an error or a default value might be used. The EXIT instruction behaves differently in a "program" than in an external subroutine. In a "program", it returns control to the caller e.g. the operating system command interpreter. While for an external routine, it returns control to the calling REXX script, independent of the level of nesting inside the external routine being terminated.

	RETURN	**EXIT**
At the main level of the program	Exits program	Exits program
At an internal subroutine level of the program	Exits subroutine, and returns to caller	Exits program
At the main level of an external subroutine	Exits the external subroutine	Exits the external subroutine
At a subroutine level within an external subroutine	Exits the subroutine, returning to calling routine within external subroutine script	Exits the external subroutine

Actions of RETURN and EXIT Instructions

If terminating an external routine (i.e. returning to the calling REXX script) any legal REXX string value is allowed as a return value. Also, no return value can be returned, and in both cases, this information is successfully transmitted back to the calling routine. In the case of a function call (as opposed to a subroutine call), returning no value will raise SYNTAX condition {44}. The table above describes the actions taken by the EXIT and RETURN instruction in various situations.

4.7 The IF/THEN/ELSE Instruction

```
IF expr [;] THEN [;] statement
                        [ ELSE [;] statement ]
```

This is a normal if-construct. First the boolean expression *expr* is evaluated, and its value must be either `0` or `1` (everything else is a syntax error which raises SYNTAX condition number {34}). Then, the statement following either `THEN` or `ELSE` is executed, depending on whether *expr* was `1` or `0`, respectively.

Note that there must come a statement after `THEN` and `ELSE`. It is not allowed to put just a null-clause (i.e. a comment or a label) there. If you want the `THEN` or `ELSE` part to be empty, use the `NOP` instruction. Also note that you can not directly put more than one statement after `THEN` or `ELSE`; you have to package them in a `DO-END` pair to make them a single, conceptual statement.

After `THEN`, after `ELSE`, and before `THEN`, you might put one or more clause delimiters (newlines or semicolons), but these are not required. Also, the `ELSE` part is not required either, in which case no code is executed if *expr* is false (evaluates to `0`). Note that there must also be a statement separator before `ELSE`, since the that statement must be terminated. This also applies to the statement after `ELSE`. However, since *statement* includes a trailing clause delimiter itself, this is not explicitly shown in the syntax diagram.

Example: Dangling ELSE

Note the case of the "dangling" `ELSE`. If an `ELSE` part can correctly be thought of as belonging to more than one `IF`/`THEN` instruction pair, it will be parsed as belonging to the closest (i.e. innermost) `IF` instruction:

```
parse pull foo bar
if foo then
      if bar then
            say 'foo and bar are true'
      else
            say 'one or both are false'
```

In this code, the `ELSE` instruction is nested to the innermost `IF`, i.e. to `IF BAR THEN`.

4.8 The INTERPRET Instruction

```
INTERPRET expr ;
```

The `INTERPRET` instruction is used to dynamically build and execute REXX instructions during run-time. First, it evaluates the expression *expr*, and then parses and interprets the result as a (possibly empty) list of REXX instructions to be executed. For instance:

```
foo = 'hello, world'
interpret 'say "''foo''!"''
```

executes the statement `SAY "hello, world!"` after having evaluated the expression following `INTERPRET`. This example shows several important aspects of `INTERPRET`. Firstly, it's very easy to get confused by the levels of quotes, and a bit of caution should be taken to nest the quotes correctly. Secondly, the use of `INTERPRET` does not exactly improve readability.

Also, `INTERPRET` will probably increase execution time considerably if put inside loops, since the interpreter may be forced to reparse the source code for each iteration. Many optimizing REXX interpreters (and in particular REXX compilers) has little or no support for `INTERPRET`. Since virtually anything can happen inside it, it is hard to optimize, and it often invalidates assumptions in other parts of the script, forcing it to ignore other possible optimizations. Thus, you should avoid `INTERPRET` when speed is at a premium.

There are some restrictions on which statements can be inside an `INTERPRET` statement. Firstly, labels cannot occur there. TRL states that they are not allowed, but you may find that in some implementations labels occurring there will not affect the label symbol table of the program being run. Consider the statement:

```
interpret 'signal there; there: say hallo'
there:
```

This statement transfers control to the label THERE in the program, never to the THERE label inside the expression of the INTERPRET instruction. Equivalently, any SIGNAL to a label THERE elsewhere in the program never transfers control to the label inside the INTERPRET instruction. However, labels are strictly speaking not allowed inside INTERPRET strings.

Example: Self-modifying Program

There is an idea for a self-modifying program in REXX which is basically like this:

```
string = ''
do i=1 to sourceline()
      string = string ';' sourceline(i)
end

string = transform( string )
interpret string
exit

transform: procedure
      parse arg string
      /* do some transformation on the argument */
      return string
```

Unfortunately, there are several reasons why this program will not work in REXX, and it may be instructive to investigate why. Firstly, it uses the label TRANSFORM, which is not allowed in the argument to INTERPRET. The interpret will thus refer to the TRANSFORM routine of the "outermost" invocation, not the one "in" the INTERPRET string.

Secondly, the program does not take line continuations into mind. Worse, the SOURCELINE() built-in function refers to the data of the main program, even inside the code executed by the INTERPRET instruction. Thirdly, the program will never end, as it will nest itself up till an implementation-dependent limit for the maximum number of nested INTERPRET instructions.

In order to make this idea work better, temporary files should be used.

On the other hand, loops and other multi-clause instructions, like IF and SELECT occur inside an INTERPRET expression, but only if the whole instruction is there; you can not start a structured instruction inside an INTERPRET instruction and end it outside, or vice-versa. However, the instruction SIGNAL is allowed even if the label is not in the interpreted string. Also, the instructions ITERATE and LEAVE are allowed in an INTERPRET, even when they refer to a loop that is external to the interpreted string.

Most of the time, INTERPRET is not needed, although it can yield compact and interesting code. If you do not strictly need INTERPRET, you should consider not using it, for reasons of compatibility, speed, and readability. Many of the traditional uses of INTERPRET have been replaced by other mechanisms in order to decrease the necessity of INTERPRET; e.g. indirect specification of variables in EXPOSE and DROP, the improved VALUE() built-in function, and indirect specification of patterns in templates.

Only semicolon (;) is allowed as a clause delimiter in the string interpreted by an INTERPRET instruction. The colon of labels can not be used, since labels are not allowed. Nor does specific end-of-line character sequences have any defined meaning there. However, most interpreters probably allow the end-of-line character sequence of the host operating system as alternative clause delimiters. It is interesting to note that in the context of the INTERPRET instruction, an implicit, trailing clause delimiter is always appended to the string to be interpreted.

4.9 The ITERATE Instruction

```
ITERATE [ symbol ] ;
```

The ITERATE instruction will iterate the innermost, active loop in which the ITERATE instruction is located. If *symbol* is specified, it will iterate the innermost, active loop having *symbol* as control variable. The simple DO/END statement without a *repetitor* and *conditional* is not affected by ITERATE. All active multiclause structures (DO, SELECT, and IF) within the loop being iterated are terminated.

The effect of an ITERATE is to immediately transfer control to the END statement of the affected loop, so that the next (if any) iteration of the loop can be started. It only affects loops on the current procedural level. All actions normally associated with the end of an iteration is performed.

Note that *symbol* must be specified literally; i.e. tail substitution is not performed for compound variables. So if the control variable in the DO instruction is FOO.BAR, then *symbol* must use FOO.BAR if it is to refer to the control variable, no matter the value of the BAR variable.

Also note that ITERATE (and LEAVE) are means of transferring control in the program, and therefore they are related to SIGNAL, but they do have the effect of automatically terminating all active loops on the current procedural level, which SIGNAL has.

Two types of errors can occur. Either *symbol* does not refer to any loop active at the current procedural level; or (if *symbol* is not specified) there does not exist any active loops at the current procedural level. Both errors are reported as SYNTAX condition {28}.

4.10 The LEAVE Instruction

```
LEAVE [ symbol ] ;
```

This statement terminates the innermost, active loop. If *symbol* is specified, it terminates the innermost, active loop having *symbol* as control variable. As for scope, syntax, errors, and functionality, it is identical to ITERATE, except that LEAVE terminates the loop, while ITERATE lets the loop start on the next iteration normal iteration. No actions normally associated with the normal end of an iteration of a loop is performed for a LEAVE instruction.

Example: Iterating a simple DO/END

In order to circumvent this, a simple DO/END can be rewritten as this:

```
if foo then do until 1
      say 'This is a simple DO/END group'
      say 'but it can be terminated by'
      leave
      say 'iterate or leave'
end
```

This shows how ITERATE has been used to terminate what for all practical purposes is a simple DO/END group. Either ITERATE or LEAVE can be used for this purpose, although LEAVE is perhaps marginally faster.

4.11 The NOP Instruction

```
NOP ;
```

The NOP instruction is the "no operation" statement; it does nothing. Actually, that is not totally true, since the NOP instruction is a "real" statement (and a placeholder), as opposed to null clauses. I've only seen this used in two circumstances.

- After any THEN or ELSE keyword, where a statement is required, when the programmer wants an empty THEN or ELSE part. By the way, this is the intended use of NOP. Note that you can not use a null clause there (label, comment, or empty lines), since these are not parsed as "independent" statements.

- I have seen it used as “trace-bait”. That is, when you start interactive trace, the statement immediately after the TRACE instruction will be executed before you receive interactive control. If you don’t want that to happen (or maybe the TRACE instruction was the last in the program), you need to add an extra dummy statement. However, in this context, labels and comments can be used, too.

4.12 The NUMERIC Instruction

```
NUMERIC = DIGITS [ expr ] ;
      FORM [ SCIENTIFIC | ENGINEERING | [ VALUE ] expr ] ;
      FUZZ [ expr ] ;
```

REXX has an unusual form of arithmetic. Most programming languages use integer and floating point arithmetic, where numbers are coded as bits in the computers native memory words. However, REXX uses floating point arithmetic of arbitrary precision, that operates on strings representing the numbers. Although much slower, this approach gives lots of interesting functionality. Unless number-crunching is your task, the extra time spent by the interpreter is generally quite acceptable and often almost unnoticeable.

The NUMERIC statement is used to control most aspects of arithmetic operations. It has three distinct forms: DIGITS, FORM and FUZZ; which to choose is given by the second token in the instruction:

DIGITS

Is used to set the number of significant digits in arithmetic operations. The initial value is 9, which is also the default value if *expr* is not specified. Large values for DIGITS tend to slow down some arithmetic operations considerably. If specified, *expr* must be a positive integer.

FUZZ

Is used in numeric comparisons, and its initial and default value is 0. Normally, two numbers must have identical numeric values for a number of their most significant digits in order to be considered equal. How many digit are considered is determined by DIGITS. If DIGITS is 4, then 12345 and 12346 are equal, but not 12345 and 12356. However, when FUZZ is non-zero, then only the DIGITS minus FUZZ most significant digits are checked. E.g. if DIGITS is 4 and FUZZ are 2, then 1234 and 1245 are equal, but not 1234 and 1345.

The value for FUZZ must be a non-negative integer, and less than the value of DIGITS. FUZZ is seldom used, but is useful when you want to make comparisons less influenced by inaccuracies. Note that using with values of FUZZ that is close to DIGITS may give highly surprising results.

FORM

Is used to set the form in which exponential numbers are written. It can be set to either SCIENTIFIC or ENGINEERING. The former uses a mantissa in the range 1.000... to 9.999..., and an exponent which can be any integer; while the latter uses a mantissa in the range 1.000... to 999.999..., and an exponent which is dividable by 3. The initial and default setting is SCIENTIFIC. Following the subkeyword FORM may be the subkeywords SCIENTIFIC and ENGINEERING, or the subkeyword VALUE. In the latter case, the rest of the statement is considered an expression, which will evaluate to either SCIENTIFIC or ENGINEERING. However, if the first token of the expression following VALUE is neither a symbol nor literal string, then the VALUE subkeyword can be omitted.

The setting of FORM never affects the decision about whether to choose exponential form or normal floating point form; it only affects the appearance of the exponential form once that form has been selected.

Many things can be said about the usefulness of FUZZ. My impression is that it is seldom used in REXX programs. One problem is that it only addresses relative inaccuracy: i.e. that the smaller value must be within a certain range, that is determined by a percentage of the larger value. Often one needs absolute inaccuracy, e.g. two measurements are equal if their difference are less than a certain absolute threshold.

Example: Simulating relative accuracy with absolute accuracy

As explained above, REXX arithmetic has only relative accuracy, in order to obtain absolute accuracy, one can use the following trick:

```
numeric fuzz 3
if a=b then
      say 'relative accuracy'
if abs(a-b)<=500 then
      say 'absolute accuracy'
```

In the first IF instruction, if A is 100,000, then the range of values for B which makes the expression true is 99,500—100,499, i.e. an inaccuracy of about +-500. If A has the value 10,000,000, then B must be within the range 9,950,000—10,049,999; i.e. an inaccuracy of about +-50,000.

However, in the second IF instruction, assuming A is 100,000, the expression becomes true for values of B in the range 99,500—100,500. Assuming that A is 10,000,000, the expression becomes true for values of B in the range 9,999,500—10,000,500.

The effect is largely to force an absolute accuracy for the second example, no matter what the values of A and B are. This transformation has taken place since an arithmetic subtraction is not affected by the NUMERIC FUZZ, only numeric comparison operations. Thus, the effect of NUMERIC FUZZ on the implicit subtraction in the operation = in the first IF has been removed by making the subtraction explicit.

Note that there are some minor differences in how numbers are rounded, but this can be fixed by transforming the expression into something more complex.

To retrieve the values set for NUMERIC, you can use the built-in functions DIGITS(), FORM(), and FUZZ(). These values are saved across subroutine calls and restored upon return.

4.13 The OPTIONS Instruction

```
OPTIONS expr ;
```

The OPTIONS instruction is used to set various interpreter-specific options. Its typical uses are to select certain REXX dialects, enable optimizations (e.g. time versus memory considerations), etc. No standard dictates what may follow the OPTIONS keyword, except that it should be a valid REXX expression, which is evaluated. Currently, no specific options are required by any standard.

The contents of *expr* is supposed to be word based, and it is the intention that more than one option can be specified in one OPTIONS instruction. REXX interpreters are specifically instructed to ignore OPTIONS words which they do not recognize. That way, a program can use run-time options for one interpreter, without making other interpreters trip when they see those options. An example of OPTION may be:

```
OPTIONS 4.00 NATIVE_FLOAT
```

The instruction might instruct the interpreter to start enforcing language level 4.00, and to use native floating point numbers in stead of the REXX arbitrary precision arithmetic. On the other hand, it might also be completely ignored by the interpreter.

It is uncertain whether modes selected by OPTIONS will be saved across subroutine calls. Refer to implementation-specific documentation for information about this.

Example: Drawback of OPTIONS

Unfortunately, the processing of the OPTIONS instruction has a drawback. Since an interpreter is instructed to ignore option-settings that it does not understand, it may ignore options which are essential for further processing of the program. Continuing might cause a fatal error later, although the behavior that would most precisely point out the problem is a complaint about the non-supported OPTION setting. Consider:

```
options 'cms_bifs'
pos = find( haystack, needle )
```

If this code fragment is run on an interpreter that does not support the `cms_bifs` option setting, then the `OPTIONS` instruction may still seem to have been executed correctly. However, the second clause will generally crash, since the `FIND()` function is still not available. Even though the real problem is in the first line, the error message is reported for the second line.

4.14 The PARSE Instruction

```
PARSE [ UPPER ] type [ template ] ;
      type = { ARG | LINEIN | PULL | SOURCE | VERSION }
      VALUE [ expr ] WITH
      VAR symbol
```

The PARSE instruction takes one or more source strings, and then parses them using the *template* for directions. The process of parsing is one where parts of a source string are extracted and stored in variables. Exactly which parts, is determined by the patterns. A complete description of parsing is given in chapter [not yet written].

Which strings are to be the source of the parsing is defined by the *type* subclause, which can be any of:

ARG.
The data to use as the source during the parsing is the argument strings given at the invocation of this procedure level. Note that this is the only case where the source may consist of multiple strings.

LINEIN.
Makes the PARSE instruction read a line from the standard input stream, as if the `LINEIN()` built-in function had been called. It uses the contents of that line (after stripping off end-of-line characters, if necessary) as the source for the parsing. This may raise the `NOTREADY` condition if problems occurred during the read.

PULL.
Retrieves as the source string for the parsing the topmost line from the stack. If the stack is empty, the default action for reading an empty stack is taken. That is, it will read a whole line from the standard input stream, strip off any end-of-line characters (if necessary), and use that string as the source.

SOURCE.
The source string for the parsing is a string containing information about how this invocation of the REXX interpreter was started. This information will not change during the execution of a REXX script. The format of the string is:

```
system invocation filename
```

Here, the first space-separated word (*system*) is a single word describing the platform on which the system is running. Often, this is the name of the operating system. The second word describes how the script was invoked. TRL2 suggests that *invocation* could be `COMMAND`, `FUNCTION`, or `SUBROUTINE`, but notes that this may be specific to VM/CMS.

Everything after the second word is implementation-dependent. It is indicated that it should refer to the name of the REXX script, but the format is not specified. In practice, the format will differ because the format of file names differs between various operating systems. Also, the part after the second word might contain other types of information. Refer to the implementation-specific notes for exact information.

VALUE expr WITH.
This form will evaluate *expr* and use the result of that evaluation as the source string to be parsed. The token `WITH` may not occur inside *expr*, since it is a reserved subkeyword in this context.

VAR symbol.
This form uses the current value of the named variable *symbol* (after tail-substitution) as the source string to be parsed. The variable may be any variable symbol. If the variable is uninitialized, then a `NOTREADY` condition will be raised.

VERSION.

This format resembles SOURCE, but it contains information about the version of REXX that the interpreter supports. The string contains five words, and has the following format:

```
language level date month year
```

Where *language* is the name of the language supported by the REXX interpreter. This may seem like overkill, since the language is REXX, but there may be various different dialects of REXX. The word can be just about anything, except for two restrictions, the first four letters should be REXX (in upper case), and the word should not contain any periods. [TRL2] indicates that the remainder of the word (after the fourth character) can be used to identify the implementation.

The second word is the REXX language level supported by the interpreter. Note that this is not the same as the version of the interpreter, although several implementations makes this mistake. Strictly speaking, neither [TRL1] nor [TRL2] define the format of this word, but a numeric format is strongly suggested.

The last three words (*date*, *month*, and *year*) makes up the date part of the string. This is the release date of the interpreter, in the default format of the DATE() built-in function.

Much confusion seems to be related to the second word of PARSE VERSION. It describes the language level, which is not the same as the version number of the interpreter. In fact, most interpreters have a version numbering which is independent of the REXX language level. Unfortunately, several interpreters makes the mistake of using this field as for their own version number. This is very unfortunate for two reasons; first, it is incorrect, and second, it makes it difficult to determine which REXX language level the interpreter is supposed to support.

Chances are that you can find the interpreter version number in PARSE SOURCE or the first word of PARSE VERSION.

The format of the REXX language level is not rigidly defined, but TRL1 corresponds to the language level 3.50, while TRL2 corresponds to the language level 4.00. Both implicitly indicate the that language level description is a number, and states that an implementation less than a certain number "may be assumed to indicate a subset" of that language level. However, this must not be taken to literally, since language level 3.50 has at least two features which are missing in language level 4.00 (the Scan trace setting, and the PROCEDURE instruction that is not forced to be the first instruction in a subroutine). [TRH:PRICE] gives a very good overview over the varying functionality of different language levels of REXX up to level 4.00.

With the release of the ANSI REXX Standard [ANSI] in 1996, the REXX language IS now rigidly defined. The language level of ANSI REXX is 5.00. Regina is attempting to keep pace with the ANSI Standard. It includes some features of language level 5.00 such as date and time conversions in the DATE() and TIME() BIFs plus the new BIFs COUNTSTR() and CHANGESTR(). Regina does not supply multiple-level error messages as defined in the ANSI Standard, so does not comply to language level 5.00, but currently is a hybrid between 4.00 and 5.00. Thus PARSE VERSION will return 4.50 :-)

Note that even though the information of the PARSE SOURCE is constant throughout the execution of a REXX script, this is not necessarily correct for the PARSE VERSION. If your interpreter supports multiple language levels (e.g. through the OPTIONS instruction), then it will have to change the contents of the PARSE VERSION string in order to comply with different language levels. To some extent, this may also apply to PARSE SOURCE, since it may have to comply with several implementation-specific standards.

After the source string has been selected by the *type* subclause in the PARSE instruction, this string is parsed into the template. The functionality of templates is common for the PARSE, ARG and PULL instructions, and is further explained in chapter [not yet written].

4.15 The PROCEDURE Instruction

```
PROCEDURE [ EXPOSE [ varref [ varref ... ] ] ] ;
      varref = { symbol | ( symbol ) }
```

The PROCEDURE instruction is used by REXX subroutines in order to control how variables are shared among routines. The simplest use is without any parameters; then all future references to variables in that subroutine refer to

local variables. If there is no PROCEDURE instruction in a subroutine, then all variable references in that subroutine refer to variables in the calling routine's name space.

If the EXPOSE subkeyword is specified too, then any references to the variables in the list following EXPOSE refer to local variables, but to variables in the name space of the calling routine.

Example: Dynamic execution of PROCEDURE

The definition opens for some strange effects, consider the following code:

```
call testing

testing:
      say foo
      procedure expose bar
      say foo
```

Here, the first reference to FOO is to the variable FOO in the caller routine's name space, while the second reference to FOO is to a local variable in the called routine's name space. This is difficult to parse statically, since the names to expose (and even when to expose them) is determined dynamically during run-time. Note that this use of PROCEDURE is allowed in [TRL1], but not in [TRL2].

Several restrictions have been imposed on PROCEDURE in [TRL2] in order to simplify the execution of PROCEDURE (and in particular, to ease the implementation of optimizing interpreters and compilers).

- The first restriction, to which all REXX interpreters adhere as far as I know, is that each invocation of a subroutine (i.e. not the main program) may execute PROCEDURE at most once. Both TRL1 and TRL2 contain this restriction. However, more than one PROCEDURE instruction may exist "in" each routine, as long as at most one is executed at each invocation of the subroutine.

- The second restriction is that the PROCEDURE instruction must be the first statement in the subroutine. This restriction was introduced between REXX language level 3.50 and 4.00, but several level 4.00 interpreters may not enforce it, since there is no breakage when allowing it.

There are several important consequences of this second restriction:

(1) it implicitly includes the first restriction listed above, since only one instruction can be the first; (2) it prohibits selecting one of several possible PROCEDURE instructions; (3) it prohibits using the same variable name twice; first as an exposed and then as a local variable, as indicated in the example above; (4) it prohibits the customary use of PROCEDURE and INTERPRET, where the latter is used to create a level of indirectness for the PROCEDURE instruction. This particular use can be exemplified by:

```
testing:
    interpret 'procedure expose' bar
```

where BAR holds a list of variable names which are to be exposed. However, in order to make this functionality available without having to resort to INTERPRET, which is generally considered "bad" programming style, new functionality has been added to PROCEDURE between language levels 3.50 and 4.00. If one of the variables in the list of variables is enclosed in parentheses, that means indirection. Then, the variables exposed are: (1) the variable enclosed in parentheses; (2) the value of that variable is read, and its contents is taken to be a space-separated list of variable names; and (3) all there variable names are exposed strictly in order from left to right.

Example: Indirect exposing

Consider the following example:

```
testing:
      procedure expose foo (bar) baz
```

Assuming that the variable BAR holds the value one two, then variables exposed are the following: FOO, BAR, ONE, TWO, BAZ, in that order. In particular, note that the variable FOO is exposed immediately before the variables which it names are exposed.

Example: Order of exposing

Then there is another fine point about exposing, the variables are hidden immediately after the EXPOSE subkeyword, so they are not initially available when the variable list is processed. Consider the following code:

```
testing:
     procedure expose bar foo.bar foo.baz baz
```

which exposes variables in the order specified. If the variable BAR holds the value 123, then FOO.123 is exposed as the second item, since BAR is visible after having already been exposed as the first item. On the other hand, the third item will always expose the variable FOO.BAZ, no matter what the value of BAZ is in the caller, since the BAZ variable is visible only after it has been used in the third item. Therefore, the order in which variables are exposed is important. So, if a compound variable is used inside parentheses in an PROCEDURE instruction, then any simple symbols needed for tail substitution must previously to have been explicitly exposed. Compare this to the DROP instruction.

What exactly is exposing? Well, the best description is to say that it makes all future uses (within that procedural level) to a particular variable name refer to the variable in the calling routine rather than in the local subroutine. The implication of this is that even if it is dropped or it has never been set, an exposed variable will still refer to the variable in the calling routine. Another important thing is that it is the tail-substituted variable name that is exposed. So if you expose FOO.BAR, and BAR has the value 123, then only FOO.123 is exposed, and continues to be so, even if BAR later changes its value to e.g. 234.

Example: Global variables

A problem lurking on new REXX users, is the fact that exposing a variable only exposes it to the calling routine. Therefore, it is incorrect to speak of global variables, since the variable might be local to the calling routine. To illustrate, consider the following code:

```
foo = 'bar'
call sub1
call sub2
exit

sub1: procedure expose foo
      say foo  /* first says 'bar', then 'FOO' */
      return

sub2: procedure
      say foo  /* says 'FOO' */
      call sub1
      return
```

Here, the first subroutine call in the "main" program writes out bar, since the variable FOO in SUB1 refers to the FOO variable in the main program's (i.e. its caller routine's) name space. During the second call from the main program, SUB2 writes out FOO, since the variable is not exposed. However, SUB2 calls SUB1, which exposes FOO, but that subroutine also writes out FOO. The reason for this is that EXPOSE works on the run-time nesting of routines, not on the typographical structure of the code. So the PROCEDURE in SUB1 (on its second invocation) exposes FOO to SUB2, not to the main program as typography might falsely indicate.

The often confusing consequence of the run-time binding of variable names is that an exposed variable of SUB1 can be bound to different global variables, depending on from where it was called. This differs from most compiled languages, which bind their variables independently of from where a subroutine is called. In turn, the consequence of this is that REXX has severe problems storing a persistent, static variable which is needed by one subroutine only. A subroutine needing such a variable (e.g. a count variable which is incremented each time the subroutine is called),

must either use an operating system command, or all subroutines calling that subroutine (and their calling routines, etc.) must expose the variable. The first of these solution is very inelegant and non-standard, while the second is at best troublesome and at worst seriously limits the maximum practical size of a REXX program. There are hopes that the VALUE( ) built-in function will fix this in future standards of REXX.

Another important drawback with PROCEDURE is that it only works for internal subroutines; for external subroutines it either do not work, or PROCEDURE may not even be allowed on the main level of the external subroutine. However, in internal subroutines inside the external subroutines, PROCEDURE is allowed, and works like usual.

4.16 The PULL Instruction

```
PULL [ template ] ;
```

This statement takes a line from the top of the stack and parse it into the variables in the *template*. It will also translate the contents of the line to uppercase.

This statement is equivalent to PARSE UPPER PULL [*template*] with the same exception as explained for the ARG instruction. See chapter [not yet written] for a description of parsing and chapter Stack for a discussion of the stack.

4.17 The PUSH Instruction

```
PUSH [ expr ] ;
```

The PUSH instruction will add a string to the stack. The string added will either be the result of the *expr*, or the nullstring if *expr* is not specified.

The string will be added to the top of the stack (LIFO), i.e. it will be the first line normally extracted from the stack. For a thorough discussion of the stack and the methods of manipulating it, see chapter Stack for a discussion of the stack.

4.18 The QUEUE Instruction

```
QUEUE [ expr ] ;
```

The QUEUE instruction is identical to the PUSH instruction, except for the position in the stack where the new line is inserted. While the PUSH puts the line on the “top” of the stack, the QUEUE instruction inserts it at the bottom of the stack (FIFO), or in the bottom of the topmost buffer, if buffers are used.

For further information, refer to documentation for the PUSH instruction, and see chapter Stack for general information about the stack.

4.19 The RETURN Instruction

```
RETURN [ expr ] ;
```

The RETURN instruction is used to terminate the current procedure level, and return control to a level above. When RETURN is executed inside one or more nesting construct, i.e. DO, IF, WHEN, or OTHERWISE, then the nesting constructs (in the procedural levels being terminated) are terminated too.

Optionally, an expression can be specified as an argument to the RETURN instruction, and the string resulting from evaluating this expression will be the return value from the procedure level terminated to the caller procedure level. Only a single value can be returned. When RETURN is executed with no argument, no return value is returned to the caller, and then a SYNTAX condition {44} is raised if the subroutine was invoked as a function.

Example: Multiple entry points

A routine can have multiple exit points, i.e. a procedure can be terminated by any of several RETURN instructions. A routine can also have multiple entry points, i.e. several routine entry points can be terminated by the same RETURN instruction. However, this is rarer than having multiple exit points, because it is generally perceived that it creates less structured and readable code. Consider the following code:

```
call foo
call bar
call baz
exit

foo:
      if datatype(name, 'w') then
            drop name
      signal baz
bar:
      name = 'foo'
baz:
      if symbol('name')== 'VAR' then
            say 'NAME currently has the value' name
      else
            say 'NAME is currently an unset variable'
      return
```

Although this is hardly a very practical example, it shows how the main bulk of a routine can be used together with three different entry points. The main part of the routine is the IF statement having two SAY statements. It can be invoked by calling FOO, BAR, or BAZ.

There are several restrictions to this approach. For instance, the PROCEDURE statement becomes cumbersome, but not impossible, to use.

Also note that when a routine has multiple exit points, it may choose to return a return value only at some of those exit points.

When a routine is located at the very end of a source file, there is an implicit RETURN instruction after the last explicit clause. However, according to good programming practice, you should avoid taking advantage of this feature, because it can create problems later if you append new routines to the source file and forget to change the implied RETURN to an explicit one.

If the current procedure level is the main level of either the program or an external subroutine, then a RETURN instruction is equivalent to an EXIT instruction, i.e. it will terminate the REXX program or the external routine. The table in the Exit section shows the actions of both the RETURN and EXIT instructions depending on the context in which they occur.

The SAY Instruction

```
SAY [ expr ] ;
```

Evaluates the expression *expr*, and prints the resulting string on the standard output stream. If *expr* is not specified, the nullstring is used instead. After the string has been written, an implementation-specific action is taken in order to produce an end-of-line.

The SAY instruction is roughly equivalent to

```
call lineout , expr
```

The differences are that there is no way of determining whether the printing was successfully completed if SAY is used, and the special variable RESULT is never set when executing a SAY instruction. Besides, the effect of omitting *expr* is different. In SAA API, the RXSIOSAY subfunction of the RXSIO exit handler is able to trap a SAY instruction, but not a call to the LINEOUT() built-in function. Further, the NOTREADY condition is never raised for a SAY instruction.

4.20 The SELECT/WHEN/OTHERWISE Instruction

```
SELECT ; whenpart [ whenpart ... ] [ OTHERWISE [;]
      [ statement ... ] ] END ;

      whenpart : WHEN expr [;] THEN [;] statement
```

This instruction is used for general purpose, nested IF structures. Although it has certain similarities with CASE in Pascal and switch in C, it is in some respects very different from these. An example of the general use of the SELECT instruction is:

```
select
      when expr1 then statement1
      when expr2 then do
            statement2a
            statement2b
      end
      when expr3 then statement3
      otherwise
            ostatement1
            ostatement2
end
```

When the SELECT instruction is executed, the next statement after the SELECT statement must be a WHEN statement. The expression immediately following the WHEN token is evaluated, and must result in a valid boolean value. If it is true (i.e. 1), the statement following the THEN token matching the WHEN is executed, and afterwards, control is transferred to the instruction following the END token matching the SELECT instruction. This is not completely true, since an instruction may transfer control elsewhere, and thus implicitly terminate the SELECT instruction; e.g. LEAVE, EXIT, ITERATE, SIGNAL, or RETURN or a condition trapped by method SIGNAL.

If the expression of the first WHEN is not true (i.e. '0), then the next statement must be either another WHEN or an OTHERWISE statement. In the former case, the process explained above is iterated. In the latter case, the clauses following the OTHERWISE up to the END statement are interpreted.

It is considered a SYNTAX condition, {7} if no OTHERWISE statement when none of the WHEN-expressions evaluates to true. In general this can only be detected during runtime. However, if one of the WHENs is selected, the absence of an OTHERWISE is not considered an error.

By the nature of the SELECT instruction, the WHENs are tested in the sequence they occur in the source. If more than one WHEN have an expression that evaluates to true, the first one encountered is selected.

If the programmer wants to associate more than one statement with a WHEN statement, a DO/END pair must be used to enclose the statements, to make them one statement conceptually. However, zero, one, or more statements may be put after the OTHERWISE without having to enclose them in a DO/END pair. The clause delimiter is optional after OTHERWISE, and before and after THEN.

Example: Writing SWITCH as IF

Although CASE in Pascal and switch in C are in general table-driven (they check an integer constant and jumps directly to the correct case, based on the value of the constant), SELECT in REXX is not so. It is a just a shorthand notation for nested IF instructions. Thus a SWITCH instruction can always be written as set of nested IF statements; but for very large SWITCH statements, the corresponding nested IF structure may be too deeply nested for the interpreter to handle.

The following code shows how the SWITCH statement shown above can be written as a nested IF structure:

```
if expr1 then statement1
else if expr2 then do
      statement2a
      statement2b
```

```
end else if expr3 then statement3
else
      ostatement1
      ostatement2
end
```

4.21 The SIGNAL Instruction

```
SIGNAL = { string | symbol } ;
      [ VALUE ] expr ;
      { ON | OFF } condition [ NAME
      { string | symbol } ] ;
```

The SIGNAL instruction is used for two purposes: (a) to transfer control to a named label in the program, and (b) to set up a named condition trap.

The first form in the syntax definition transfers control to the named label, which must exist somewhere in the program; if it does not exist, a SYNTAX condition {16} is raised. If the label is multiple defined, the first definition is used. The parameter can be either a symbol (which is taken literally) or a string. If it is a string, then be sure that the case of the string matches the case of the label where it is defined. In practice, labels are in upper case, so the string should contain only uppercase letters too, and no space characters.

The second form of the syntax is used if the second token of the instruction is VALUE. Then, the rest of the instruction is taken as a general REXX expression, which result after evaluation is taken to be the name of the label to transfer control to. This form is really just a special case of the first form, where the programmer is allowed to specify the label as an expression. Note that if the start of expr is such that it can not be misinterpreted as the first form (i.e. the first token of *expr* is neither a string nor a symbol), then the VALUE subkeyword can be omitted.

Example: Transferring control to inside a loop

When the control of execution is transferred by a SIGNAL instruction, all active loops at the current procedural level are terminated, i.e. they can not continued later, although they can of course be reentered from the normal start. The consequence of this is that the following code is illegal:

```
do forever
      signal there
there:
nop
end
```

The fact that the jump is altogether within the loop does not prevent the loop from being terminated. Thus, after the jump to the loop, the END instruction is attempted executed, which will result in a SYNTAX condition {10}. However, if control is transferred out of the loop after the label, but before the END, then it would be legal, i.e. the following is legal:

```
do forever
      signal there
there:
nop
signal after
end

after:
```

This is legal, simply because the END instruction is never seen during this script. Although both TRL1 and TRL2 allow this construct, it will probably be disallowed in ANSI.

Just as loops are terminated by a SIGNAL instruction, SELECT and IF instructions are also terminated. Thus, it is illegal to jump to a location within a block of statements contained in a WHEN, OTHERWISE, or IF instruction, unless the control is transferred out of the block before the execution reaches the end of the block.

Whenever execution is transferred during a SIGNAL instruction, the special variable SIGL is set to the line number of the line containing the SIGNAL instruction, before the control is transferred. If this instruction extends over several lines, it refers to the first of this. Note that even blanks are part of a clause, so if the instruction starts with a line continuation, the real line of the instruction is different from that line where the instruction keyword is located.

The third form of syntax is used when the second token in the instruction is either ON or OFF. In both cases must the third token in the instruction be then name of a condition (as a constant string or a symbol, which is taken literally), and the setup of that condition trap is changed. If the second token is OFF, then the trap of the named condition is disabled.

If the second token is ON, then the trap of the named condition is enabled. Further, in this situation two more tokens may be allowed in the instruction: the first must be NAME and the second must be the name of a label (either as a constant string or a symbol, which is taken literally). If the five token form is used, then the label of the condition handler is set to the named label, else the name of the condition handler is set to the default, which is identical to the name of the condition itself.

Note that the NAME subclause of the SIGNAL instruction was a new construct in TRL2, and is not a part of TRL1. Thus, older interpreters may not support it.

Example: Naming condition traps

Note that the default value for the condition handler (if the NAME subclause is not specified) is the name of the condition, not the condition handler from the previous time the condition was enabled. Thus, after the following code, the name of the condition handler for the condition SYNTAX is SYNTAX, not FOOBAR:

```
signal on syntax name foobar
signal on syntax
```

Example: Named condition traps in TRL1

A common problem when trying to port REXX code from a TRL2 interpreter to a TRL1 interpreter, is that explicitly named condition traps are not supported. There exist ways to circumvent this, like:

```
syntax_name = 'SYNTAX_HANDLER'
signal on syntax
if 1 + 2 then  /* will generate SYNTAX condition */
      nop
syntax:
oldsigl = sigl
signal value translate(syntax_name)

syntax_handler:
say 'condition at line' oldsigl 'is being handled...'
exit
```

Here, a "global" variable is used to store the name of the real condition handler, in the absence of a field for this in the interpreter. This works fine, but there are some problems: the variable SYNTAX_NAME must be exposed to everywhere, in order to be available at all times. It would be far better if this value could be stored somewhere from which it could be retrieved from any part of the script, no matter the current state of the call-stack. This can be fixed with programs like GLOBALV under VM/CMS and putenv under Unix.

Another problem is that this destroys the possibility of setting up the condition handler with the default handler name. However, to circumvent this, add a new DEFAULT_SYNTAX_HANDLER label which becomes the new name for the old SYNTAX label.

Further information about conditions and condition traps are given in chapter Conditions.

4.22 The TRACE Instruction

```
TRACE [ number | setting | [ VALUE ] expr ] ;
      setting = A | S | C | E | F | I | L | N | O | R | S
```

The TRACE instruction is used to set a tracing mode. Depending on the current mode, various levels of debugging information is displayed for the programmer. Also interactive tracing is allowed, where the user can re-execute clauses, change values of variables, or in general, execute REXX code interactively between the statements of the REXX script.

If *setting* is not specified, then the default value N is assumed. If the second token after TRACE is VALUE, then the remaining parts of the clause is interpreted as an expression, which value is used as the trace setting. Else, if the second token is either a string of a symbol, then it is taken as the trace setting; and a symbol is taken literally. In all other circumstances, whatever follows the token TRACE is taken to be an expression, which value is the trace setting.

If a parameter is given to the TRACE instruction, and the second token in the instruction is not VALUE, then there must only be one token after TRACE, and it must be either a constant string or a symbol (which is taken literally). The value of this token can be either a whole number or a trace setting.

If is it a whole number and the number is positive, then the number specifies how many of interactive pauses to skip. This assumes interactive tracing; if interactive tracing is not enabled, this TRACE instruction is ignored. If the parameter is a whole, negative number, then tracing is turned off temporarily for a number of clauses determined by the absolute value of *number*.

If the second token is a symbol of string, but not a whole number, then it is taken to be one of the settings below. It may optionally be preceded by one or more question mark (?) characters. Of the rest of the token, only the first letter matter; this letter is translated to upper case, and must be one of the following:

[A]

(All) Traces all clauses before execution.

[C]

(Commands) Traces all command clauses before execution.

[E]

(Errors) Traces any command that would raise the ERROR condition (whether enabled or not) after execution. Both the command clause and the return value is traced.

[F]

(Failures) Trances any command that would raise the FAILURE condition (whether enabled or not) after execution. Both the command clause and the return value is traced.

[I]

(Intermediate) Traces not only all clauses, but also traces all evaluation of expressions; even intermediate results. This is the most detailed level of tracing.

[L]

(Labels) Traces the name of any label clause executed; whether the label was jumped to or not.

[N]

(Normal or Negative) This is the same as the Failure setting.

[O]

(Off) Turns off all tracing.

[R]

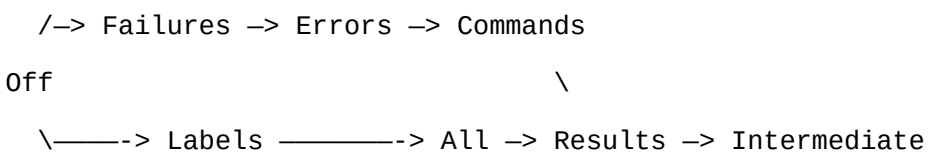
(Results) Traces all clauses and the results of evaluating expressions. However, intermediate expressions are not traced.

[S]

(Scan) Traces all clauses from the current position in the script, until the end of the file, in sequence. However, it does not execute any of the clauses. When the end of the program is reached, the interpreter exits.

The Errors and Failures settings are not influenced by whether the ERROR or FAILURE conditions are enabled or not. These TRACE settings will trace the command and return value after the command have been executed, but before the respective condition is raised.

The levels of tracing might be set up graphically, as in the figure below. An arrow indicates that the setting pointed to is a superset of the setting pointed from.



Hierarchy of TRACE settings

According to this figure, Intermediate is a superset of Result, which is a superset of All. Further, All is a superset of both Commands and Labels. Commands is a superset of Errors, which is a superset of Failures. Both Failure and Labels are supersets of Off. Actually, Command is strictly speaking not a superset of Errors, since Errors traces after the command, while Command traces before the command.

Scan is not part of this diagram, since it provides a completely different tracing functionality. Note that Scan is part of TRL1, but was removed in TRL2. It is not likely to be part of newer REXX interpreters.

5. Operators

An operator represents an operation to be carried out between two terms, such as division. There are 5 types of operators in the Rexx Language: *Arithmetic*, *Assignment*, *Comparative*, *Concatenation*, and *Logical* Operators. Each is described in further details below.

5.1 Arithmetic Operators

Arithmetic operators can be applied to numeric constants and Rexx variables that evaluate to valid Rexx numbers. The following operators are listed in descreasing order of precedence:

-	Unary prefix. Same as **0 - number**.
+	Unary prefix. Same as **0 + number**.
******	Power
*****	Multiply
/	Divide
%	Integer divide. Divide and return the integer part of the division.
//	Remainder divide. Divide and return the remainder of the division.
+	Add
-	Subtract.

5.2 Assignment Operators

Assignment operators are a means to change the value of a variable. Rexx only has one assignment operator.

=	Assign the value on the right side of the "=" to the variable on the left.

5.3 Comparative Operators

The Rexx comparative operators compare two terms and return the logical value **1** if the result of the comparison is true, or **0** if the result of the comparison is false. The non-strict comparative operators will ignore leading or trailing blanks for string comparisons, and leading zeros for numeric comparisons. Numeric comparisons are made if both

terms to be compared are valid Rexx numbers, otherwise string comparison is done. String comparisons are case sesitive, and the shorter of the two strings is padded with blanks.

The following lists the non-strict comparative operators.

=	Equal
\=, ^=	Not equal
>	Greater than.
<	Less than.
>=	Greater than or equal.
<=	Less than or equal
<>, ><	Greater than or less than. Same as Not equal.

The following lists the strict comparative operators. For two strings to be considered equal when using the strict equal comparative operator, both strings must be the same length.

==	Strictly equal
\==, ^==	Strictly not equal.
>>	Strictly greater than.
<<	Strictly less than.
>>=	Striclty greater than or equal.
<<=	Strictly less than or equal.

5.4 Concatenation Operators

The concatenation operators combine two strings to form one, by appending the seond string to the right side of the first. The Rexx concatenation operators are:

(**blank**)	Concatenation of strings with one space between them.
(**abuttal**)	Concatenation of strings with no intervening space.
||	Concatenation of strings with no intervening soace.

Examples:

```
a = abc;b = 'def'
Say a b                  -> results in 'abc def'
Say a || b          -> results in 'abcdef'
Say a'xyz'          -> results in 'abcxyz'
```

5.5 Logical Operators

Logical operators work with the Rexx strings 1 and 0, usually as a result of a comparative operator. These operators also only result in logical TRUE; 1 or logical FALSE; 0.

&	And	Returns 1 if both terms are 1.
|	Inclusive or	Returns 1 if either term is 1.
&&	Exclusive or	Returns 1 if either term is 1 but NOT both terms.
\	Logical not	Reverses the result; 0 becomes 1 and 1 becomes 0.

6. Implementation-Specific Information

6.1 Environments in Regina 0.05h

External environments name stack
External environments names
OPTIONS settings

Are saved across subroutines, just like other pieces of information, like conditions settings, NUMERIC settings, etc. See chapter Options for more information about OPTIONS settings.

Return value

To the program that called Regina is limited to being an integer, when this is required by the operating systems. All current implementations are for operating systems that require this.

Default return value

From a REXX program is `0` under most systems, specifically Unix, OS/2, MS-DOS. Here, VMS deviates, since it uses `1` as the default return value. Using `0` under VMS tends to make VMS issue a warning saying that no error occurred.

Transferring control into a loop

Works fine in Regina, as long as no `END`, `THEN`, `ELSE`, `WHEN`, or `OTHERWISE` instructions are executed afterwards; unless the normal entrypoint for the construct has been executed after the transfer of control.

PARSE SOURCE information
PARSE VERSION **information**
Last line of source code

Is implicitly taken to be terminated by an end-of-line sequence in Regina, even if such a sequence is not present in the source code of the REXX script. This applies only to source code. Also, the end-of-string in `INTERPRET` strings is taken to be implicitly terminated by an end-of-line character sequence.

Moving code MS-DOS to Unix

Is simplified by Regina, since it will accept the MS-DOS type end of line sequences as valid. I.e. any Ctrl-M in front of a Ctrl-J in the source file is ignored on Unix systems by Regina. This applies only to source code.

Labels in INTERPRET

Is handles by Regina in the following way: A label can occur inside an `INTERPRET` string, but it is ignored, and can never be jumped to in a `SIGNAL` or `CALL` instruction.

6.2 List of All Environment Names in Use

Regina supports the following environments:

ENVIRONMENT
OS2ENVIRONEMNT
SYSTEM
PATH
COMMAND

REXX Built-in Functions

This chapter describes the REXX library of built-in functions. It is divided into three parts:

- *First a general introduction to built-in functions, pointing out concepts, pitfalls, parameter conventions, peculiarities, and possible system dependencies.*
- *Then there is the reference section, which describes in detail each function in the built-in library.*
- *At the end, there is documentation that describes where and how Regina differs from standard REXX, as described in the two other sections. It also lists Regina's extensions to the built-in library.*

It is recommended that you read the first part on first on first reading of this documentation, and that you use the second part as reference. The third part is only relevant if you are going to use Regina.

1. General Information

This section is an introduction to the built-in functions. It describes common behavior, parameter conventions, concepts and list possible system-dependent parts.

1.1 The Syntax Format

In the description of the built-in functions, the syntax of each one is listed. For each of the syntax diagrams, the parts written in *italic* font names the parameters. Terms enclosed in [square brackets] denote optional elements. And the `courier` font is used to denote that something should be written as is, and it is also used to mark output from the computer.

Note that in standard REXX it is not really allowed to let the last possible parameter be empty if all commas are included, although some implementations allow it. In the following calls:

```
say D2X( 61 )
say D2X( 61, 1 )
say D2X( 61, )
```

The two first return the string consisting of a single character A, while the last should return error. If the last argument of a function call is omitted, you can not safely include the immediately preceding comma.

1.2 Precision and Normalization

The built-in library uses its own internal precision for whole numbers, which may be the range from -999999999 to +999999999. That is probably far more than you will ever need in the built-in functions. For most functions, neither parameters nor return values will be effected by any setting of `NUMERIC`. In the few cases where this does not hold, it is explicitly stated in the description of the function.

In general, only parameters that are required to be whole numbers are used in the internal precision, while numbers not required to be whole numbers are normalized according to the setting of `NUMERIC` before use. But of course, if a parameter is a numeric expression, that expression will be calculated and normalized under the settings of `NUMERIC` before it is given to the function as a parameter.

1.3 Standard Parameter Names

In the descriptions of the built-in functions, several generic names are used for parameters, to indicate something about the type and use of that parameter, e.g. valid range. To avoid repeating the same information for the majority of the functions, some common "rules" for the standard parameter names are stated here. These rules implicitly apply for the rest of this chapter.

Note that the following list does not try to classify any general REXX "datatypes", but provides a binding between the sub-datatypes of strings and the methodology used when naming parameters.

- *Length* is a non-negative whole number within the internal precision of the built-in functions. Whether it denotes a length in characters or in words, depends on the context.

- *String* can be any normal character string, including the nullstring. There are no further requirements for this parameter. Sometimes a string is called a "packed string" to explicitly show that it usually contains more than the normal printable characters.

- *Option* is used in some of the functions to choose a particular action, e.g. in `DATE()` to set the format in which the date is returned. Everything except the first character will be ignored, and case does not matter. note that the string should consequently not have any leading space.

- *Start* is a positive whole number, and denotes a start position in e.g. a string. Whether it refers to characters or words depends on the context. The first position is always numbered `1`, unless explicitly stated otherwise in the documentation. Note that when return values denotes positions, the number `0` is generally used to denote a nonexistent position.

- *Padchar* must be a string, exactly one character long. That character is used for padding.

- *Streamid* is a string that identifies a REXX stream. The actual contents and format of such a string is implementation dependent.

- *Number* is any valid REXX number, and will be normalized according to the settings of `NUMERIC` before it is used by the function.

If you see one of these names having a number appended, that is only to separate several parameters of the same type, e.g. *string1, string2* etc. They still follow the rules listed above. There are several parameters in the built-in functions that do not easily fall into the categories above. These are given other names, and their type and functionality will be described together with the functions in which they occur.

1.4 Error Messages

There are several errors that might occur in the built-in functions. Just one error message is only relevant for all the built-in functions, that is number 40 (*Incorrect call to routine*). In fact, an implementation of REXX can choose to use that for any problem it encounters in the built-in functions.

Depending on the implementation, other error messages might be used as well. Error message number 26 (*Invalid whole number*) might be used for any case where a parameter should have been a whole number, or where a whole number is out of range. It is implied that this error message can be used in these situations, and it is not explicitly mentioned in the description of the functions.

Other general error messages that might be used in the built-in functions are error number 41 (*Bad arithmetic conversion*) for any parameter that should have been a valid REXX number. The error message 15 (*Invalid binary or hexadecimal string*) might occur in any of the conversion routines that converts from binary or hexadecimal format (`B2X()`, `X2B()`, `X2C()`, `X2D()`). And of course the more general error messages like error message 5 (*Machine resources exhausted*) can occur.

Generally, it is taken as granted that these error messages might occur for any relevant built-in function, and this will not be restated for each function. When other error messages than these are relevant, it will be mentioned in the text.

In REXX, it is in general not an error to specify a start position that is larger than the length of the string, or a length that refers to parts of a string that is beyond the end of that string. The meaning of such instances will depend on the context, and are described for each function.

1.5 Possible System Dependencies

Some of the functions in the built-in library are more or less system or implementation dependent. The functionality of these may vary, so you should use defensive programming and be prepared for any side-effects that they might have. These functions include:

- ADDRESS() is dependent on your operating system and the implementation of REXX, since there is not standard for naming environments.

- ARG() at the main level (not in subroutines and functions) is dependent on how your implementation handles and parses the parameters it got from the operating system.

- BITAND(), BITOR() and BITXOR() are dependent on the character set of your machine. Seemingly identical parameters will in general return very different results on ASCII and EBCDIC machines. Results will be identical if the parameter was given to these functions as a binary or hexadecimal literal.

- C2X(), C2D(), D2C() and X2C() will be effected by the character set of your computer since they convert to or from characters. Note that if C2X() and C2D() get their first parameter as a binary or hexadecimal literal, the result will be unaffected by the machine type. Also note that the functions B2X(), X2B(), X2D() and D2X() are not effected by the character set, since they do not use character representation.

- CHARIN(), CHAROUT(), CHARS(), LINEIN(), LINEOUT(), LINES() and STREAM() are the interface to the file system. They might have system dependent peculiarities in several ways. Firstly, the naming of streams is very dependent on the operating system. Secondly, the operation of stream is very dependent on both the operating system and the implementation. You can safely assume very little about how streams behave, so carefully read the documentation for your particular implementation.

- CONDITION() is dependent on the condition system, which in turn depends on such implementation dependent things as file I/O and execution of commands. Although the general operation of this function will be fairly equal among systems, the details may differ.

- DATATYPE() and TRANSLATE() know how to recognize upper and lower case letters, and how to transform letters to upper case. If your REXX implementation supports national character sets, the operation of these two functions will depend on the language chosen.

- DATE() has the options Month, Weekday and Normal, which produce the name of the day or month in text. Depending on how your implementation handles national character sets, the result from these functions might use the correct spelling of the currently chosen language.

- DELWORD(), SUBWORD(), WORD(), WORDINDEX(), WORDLENGTH(), WORDPOS() and WORDS() requires the concept of a “word”, which is defined as a non-blank characters separated by blanks. However, the interpretation of what is a blank character depends upon the implementation.

- ERRORTEXT() might have slightly different wordings, depending on the implementation, but the meaning and numbering should be the same. However, note that some implementations may have additional error messages, and some might not follow the standard numbering.

- QUEUED() refers to the system specific concept of a “stack”, which is external to REXX. The result of this function may therefore be dependent on how the stack is implemented on your system.

- RANDOM() will differ from machine to machine, since the algorithm is implementation dependent. If you set the seed, you can safely assume that the same interpreter under the same operating system and on the same hardware platform will return a reproducible sequence. But if you change to another interpreter, another machine or even just another version of the operating system, the same seed might not give the same pseudo-random sequence.

- SOURCELINE() has been changed between REXX language level 3.50 and 4.00. In 4.00 it can return 0 if the REXX implementation finds it necessary, and any request for a particular line may get a nullstring as result. Before assuming that this function will return anything useful, consult the documentation.

- TIME() will differ somewhat on different machines, since it is dependent on the underlying operating system to produce the timing information. In particular, the granularity and accuracy of this information may vary.

- VALUE() will be dependent on implementation and operating system if it is called with its third parameter specified. Consult the implementation specific documentation for more information about how each implementation handles this situation.

- XRANGE() will return a string, which contents will be dependent on the character set used by your computer. You can safely make very few assumptions about the visual representation, the length, or the character order of the string returned by this function.

As you can see, even REXX interpreters that are within the standard can differ quite a lot in the built-in library. Although the points listed above seldom are any problem, you should never assume anything about them before you have read the implementation specific documentation. Failure to do so will give you surprises sooner or later.

And, by the way, many implementations (probably the majority) do not follow the standard completely. So, in fact, you should never assume anything at all. Sorry ...

1.6 Blanks vs. Spaces

Note that the description differs between “blanks” and the <space> character. A blank is any character that might be used as “whitespace” to separate text into groups of characters. The <space> character is only one of several possible blanks. When this text says “blank” it means any one from a set of characters that are used to separate visual characters into words. When this text says <space>, it means one particular blank, that which is generally bound to the spacebar on a normal computer keyboard.

All implementation can be trusted to treat the <space> character as blank. Additional characters that might be interpreted as blanks are <tab> (horizontal tabulator), <ff> (formfeed), <vt> (vertical tabulator), <nl> (newline) and <cr> (carriage return). The interpretation of what is blank will vary between machines, operating systems and interpreters. If you are using support for national character sets, it will even depend on the language selected. So be sure to check the documentation before you assume anything about blank characters.

Some implementations use only one blank character, and perceives the set of blank characters as equivalent to the <space> character. This will depend on the implementation, the character set, the customs of the operating system and various other reasons.

2. REXX Standard Built-in Functions

Below follows an in depth description of all the functions in the library of built-in functions. Note that only the standard REXX functions is included. The extended functions available in some implementations are not described here.

ABBREV(*long*,*short*[,*length*])

Returns 1 if the string *short* is strictly equal to the initial first part of the string *long*, and returns 0 otherwise. The minimum length which *short* must have, can be specified as *length*. If *length* is unspecified, no minimum restrictions for the length of *short* applies, and thus the nullstring is an abbreviation of any string.

Note that this function is case sensitive, and that leading and trailing spaces are not stripped off before the two strings are compared.

```
ABBREV( ‘Foobar’, ‘Foo’ )     ->    1
```

```
ABBREV( 'Foobar', 'Foo', 4)   ->    0  /*Too short */
ABBREV( 'Foobar', 'foo' )     ->    0  /*Different case */
```

ABS(*number*)

Returns the absolute value of the *number*, which can be any valid REXX number. Note that the result will be normalized according to the current setting of NUMERIC.

```
ABS( -42 )  –>    42
ABS( 100 )  –>    100
```

ADDRESS()

Returns the current default environment to which commands are sent. The value is set with the ADDRESS clause, for more information, see documentation on that clause.

```
ADDRESS()   –>    UNIX  /* Maybe */
```

ARG([*argno*[,*option*]])

Returns information about the arguments of the current procedure level. For subroutines and functions it will refer to the arguments with which they were called. For the "main" program it will refer to the arguments used when the REXX interpreter was called.

Note that under some operating systems, REXX scripts are run by starting the REXX interpreter as a program, giving it the name of the script to be executed as parameter. Then the REXX interpreter might process the command line and "eat" some or all of the arguments and options. Therefore, the result of this function at the main level is implementation dependent. The parts of the command line which are not available to the REXX script might for instance be the options and arguments meaningful only to the interpreter itself.

Also note that how the interpreter on the main level divides the parameter line into individual arguments, is implementation dependent. The standard seems to define that the main procedure level can only get one parameter string, but don't count on it.

For more information on how the interpreter processes arguments when called from the operating system, see the documentation on how to run a REXX script.

When called without any parameters, ARG() will return the number of comma-delimited arguments. Unspecified (omitted) arguments at the end of the call are not counted. Note the difference between using comma and using space to separate strings. Only comma-separated arguments will be interpreted by REXX as different arguments. Space-separated strings are interpreted as different parts of the same argument.

Argno must be a positive whole number. If only *argno* is specified, the argument specified will be returned. The first argument is numbered 1. If *argno* refers to an unspecified argument (either omitted or *argno* is greater than the number of arguments), a nullstring is returned.

If *option* is also specified, the return value will be 1 or 0, depending on the value of *option* and on whether the numbered parameter was specified or not. Option can be:

[O]
: (Omitted) Returns 1 if the numbered argument was omitted or unspecified. Otherwise, 0 is returned.

[E]
: (Existing) Returns 1 if the numbered argument was specified, and 0 otherwise.

If called as:

```
CALL FUNCTION 'This' 'is', 'a',, 'test',,
```

```
ARG()      –>    4  /*Last parameter omitted */
ARG(1)     –>    'This is'
ARG(2)     –>    'a'
ARG(3)     –>    ''
ARG(9)     –>    ''  /*Ninth parameter nonexisting */
ARG(2,'E') –>    1
ARG(2,'O') –>    0
ARG(3,'E') –>    0  /*Third parameter omitted */
ARG(9,'O') –>    1
```

B2X(*binstring*)

Takes a parameter which is interpreted as a binary string, and returns a hexadecimal string which represent the same information. *Binstring* can only contain the binary digits `0` and `1`. To increase readability, blanks may be included in *binstring* to group the digits into groups. Each such group must have a multiple of four binary digits, except from the first group. If the number of binary digits in the first group is not a multiple of four, that group is padded at the left with up to three leading zeros, to make it a multiple of four. Blanks can only occur between binary digits, not as leading or trailing characters.

Each group of four binary digits is translated into on hexadecimal digit in the output string. There will be no extra blanks in the result, and the upper six hexadecimal digits are in upper case.

```
B2X('0010 01011100 0011')     –>    '26C3'
B2X('10 0101 11111111')       –>    '26FF'
B2X('0100100 0011')           –>    '243'
```

BITAND(*string1*[,[*string2*][,*padchar*]])

Returns the result from bytewise applying the operator AND to the characters in the two strings *string1* and *string2*. Note that this is not the logical AND operation, but the bitwise AND operation. *String2* defaults to a nullstring. The two strings are left-justified; the first characters in both strings will be AND'ed, then the second characters and so forth.

The behavior of this function when the two strings do not have equal length is defined by the *padchar* character. If it is undefined, the remaining part of the longer string is appended to the result after all characters in the shorter string have been processed. If *padchar* is defined, each char in the remaining part of the longer string is logically AND'ed with the *padchar* (or rather, the shorter string is padded on the right length, using *padchar*).

When using this function on character strings, e.g. to uppercase or lowercase a string, the result will be dependent on the character set used. To lowercase a string in EBCDIC, use `BITAND()` with a *padchar* value of `'bf'x`. To do the same in ASCII, use `BITOR()` with a *padchar* value of `'20'x`.

```
BITAND('123456'x, '3456'x)          –>    '101456'x
BITAND('foobar',, 'df'x)            –>    'FOOBAR' /*For ASCII*/
BITAND('123456'x, '3456'x, 'f0'x)   –>    '101450'x
```

BITOR(*string1*[,[*string2*][,*padchar*]])

Works like `BITAND()`, except that the logical function OR is used instead of AND. For more information see `BITAND()`.

```
BITOR('123456'x, '3456'x)     –>    '367656'x
BITOR('FOOBAR',, '20'x) –>    'foobar' /*For ASCII */
BITOR('123456'x, '3456'x, 'f0'x)    –>    '3676F6'x
```

BITXOR(*string1*[,[*string2*][,*padchar*]])

Works like BITAND(), except that the logical function XOR (exclusive OR) is used instead of AND. For more information see BITAND().

```
BITXOR('123456'x, '3456'x)          ->    '266256'x
BITXOR('FooBar',, '20'x)            ->    'fOObAR' /*For ASCII */
BITXOR('123456'x, '3456'x, 'f0'x)   ->    '2662A6'x
```

C2D(*string*[,*length*])

Returns an whole number, which is the decimal representation of the packed string *string*, interpreted as a binary number. If *length* (which must be a non-negative whole number) is specified, it denotes the number of characters in *string* to be converted, and *string* is interpreted as a two's complement representation of a binary number, consisting of the length rightmost characters in *string*. If *length* is not specified, *string* is interpreted as an unsigned number.

If *length* is larger than the length of *string*, *string* is sign-extended on the left. I.e. if the most significant bit of the leftmost char of *string* is set, *string* is padded with 'ff'x chars at the left side. If the bit is not set, '00'x chars are used for padding.

If *length* is too short, only the *length* rightmost characters in *string* are considered. Note that this will not only in general change the value of the number, but it might even change the sign.

Note that this function is very dependent on the character set that your computer is using.

If it is not possible to express the final result as a whole number under the current settings of NUMERIC DIGITS, an error is reported. The number to be returned will not be stored in the internal representation of the built-in library, so size restrictions on whole numbers that generally applies for built-in functions, do not apply in this case.

```
C2D('foo')        ->    '6713199'   /*For ASCII machines */
C2D('103'x)       ->    '259'
C2D('103'x,1)     ->    '3'
C2D('103'x,2)     ->    '259'
C2D('0103'x,3)    ->    '259'
C2D('ffff'x,2)    ->    '-1'
C2D('ffff'x)      ->    '65535'
C2D('ffff'x,3)    ->    '65535'
C2D('fff9'x,2)    ->    '-6'
C2D('ff80'x,2)    ->    '-128'
```

C2X(*string*)

Returns a string of hexadecimal digits that represents the character string *string*. Converting is done bytewise, the six highest hexadecimal digits are in uppercase, and there are no blank characters in the result Leading zeros are not stripped off in the result. Note that the behavior of this function is dependent on the character set that your computer is running (e.g. ASCII or EBCDIC).

```
C2X('ffff'x)            ->    'FFFF'
C2X('Abc')              ->    '416263'    /*For ASCII Machines */
C2X('1234'x)            ->    '1234'
C2X('011 0011 1101'b)   ->    '033D'
```

CENTER(*string*, *length* [, *padchar*])
CENTRE(*string*, *length* [, *padchar*])

This function has two names, to support both American and British spelling. It will center *string* in a string total of length *length* characters. If *length* (which must be a non-negative whole number) is greater than the length of *string*, *string* is padded with *padchar* or <space> if *padchar* is unspecified. If *length* is smaller than the length of *string* character will be removed.

If possible, both ends of *string* receives (or loses) the same number of characters. If an odd number of characters are to be added (or removed), one character more is added to (or removed from) the right end than the left end of *string*.

```
CENTER('Foobar',10)     ->    '  Foobar  '
CENTER('Foobar',11)     ->    '  Foobar   '
CENTRE('Foobar',3)      ->    'oob'
CENTER('Foobar',4)      ->    'ooba'
CENTER('Foobar',10,'*') ->    '**Foobar**'
```

CHANGESTR(*needle*, *haystack*, *newneedle*)

This function was introduced with the REXX ANSI Standard. Its purpose is to replace all occurrences of *needle* in the string *haystack* with *newneedle*. The function returns the changed string.

If *haystack* does not contain *needle*, then the original *haystack* is returned.

```
CHANGESTR('a','fred','c')                -> 'fred'
CHANGESTR('','','x')                     -> ''
CHANGESTR('a','abcdef','x')              -> 'xbcdef'
CHANGESTR('0','0','1')                   -> '1'
CHANGESTR('a','def','xyz')               -> 'def'
CHANGESTR('a','','x')                    -> ''
CHANGESTR('','def','xyz')                -> 'def'
CHANGESTR('abc','abcdef','xyz')          -> 'xyzdef'
CHANGESTR('abcdefg','abcdef','xyz')      -> 'abcdef'
CHANGESTR('abc','abcdefabccdabcd','z')   -> 'zdefzcdzd'
```

CHARIN([*streamid*][,[*start*][,*length*]])

This function will in general read characters from a stream, and return a string containing the characters read. The *streamid* parameter names a particular stream to read from. If it is unspecified, the default input stream is used.

The *start* parameter specifies a character in the stream, on which to start reading. Before anything is read, the current read position is set to that character, and it will be the first character read. If *start* is unspecified, no repositioning will be done. Independent of any conventions of the operating system, the first character in a stream is always numbered 1. Note that transient streams do not allow repositioning, and an error is reported if the *start* parameter is specified for a transient stream.

The *length* parameter specifies the number of characters to read. If the reading did work, the return string will be of length *length*. There are no other ways to how many characters were read than checking the length of the return value. After the read, the current read position is moved forward as many characters as was read. If *length* is unspecified, it defaults to `1`. If *length* is `0`, nothing is read, but the file might still be repositioned if *start* was specified.

Note that this function read the stream raw. Some operating systems use special characters to differ between separate lines in text files. On these systems these special characters will be returned as well. Therefore, never assume that this function will behave identical for text streams on different systems.

What happens when an error occurs or the End-Of-File (EOF) is seen during reading, is implementation dependent. The implementation may choose to set the NOTREADY condition (does not exist in REXX language level 3.50). For more information, see chapter on **Stream Input and Output**.

(Assuming that the file "`/tmp/file`" contains the first line: "`This is the first line`"):

```
CHARIN()                ->    'F'  /*Maybe*/
CHARIN(,,6)             ->    'Foobar'  /*Maybe*/
CHARIN('/tmp/file',,6)  ->    'This i'
CHARIN('/tmp/file',4,6) ->    's is t'
```

CHAROUT([*streamid*][,[*string*][,*start*]])

In general this function will write *string* to a *streamid*. If *streamid* is not specified the default output stream will be used.

If *start* is specified, the current write position will be set to the *start*th character in *streamid*, before any writing is done. Note that the current write position ca not be set for transient streams, and attempts to do so will report an error. Independent of any conventions that the operating system might have, the first character in the stream is numbered `1`. If *start* is not specified, the current write position will not be changed before writing.

If *string* is omitted, nothing is written, and the effect is to set the current write position if *start* is specified. If neither *string* nor *start* is specified, the implementation can really do whatever it likes, and many implementations use this operation to close the file, or flush any changes. Check implementation specific documentation for more information.

The return value is the number of characters in *string* that was not successfully written, so `0` denotes a successful write. Note that in many REXX implementations there is no need to open a stream; it will be implicitly opened when it is first used in a read or write operation.

(Assuming the file referred to by `outdata` was empty, it will contain the string `FoobWow` afterwards. Note that there might will not be an End-Of-Line marker after this string, it depends on the implementation.)

```
CHAROUT(, 'Foobar')           ->    '0'
CHAROUT(outdata, 'Foobar')    ->    '0'
CHAROUT(outdata, 'Wow', 5)    ->    '0'
```

CHARS([*streamid*])

Returns the number of characters left in the named *streamid*, or the default input stream if *streamid* is unspecified. For transient streams this will always be either `1` if more characters are available, or `0` if the End-Of-File condition has been met. For persistent streams the number of remaining bytes in the file will be possible to calculate and the true number of remaining bytes will be returned.

However, on some systems, it is difficult to calculate the number of characters left in a persistent stream; the requirements to `CHARS()` has therefore been relaxed, so it can return `1` instead of any number other than `0`. If it returns `1`, you can therefore not assume anything more than that there is at least one more character left in the input stream.

```
CHARS()           ->    '1'   /* more data on def. input stream */
CHARS()           ->    '0'   /* EOF for def. input stream */
CHARS('outdata')  ->    '94'  /* maybe */
```

COMPARE(*string1*,*string2*[,*padchar*])

This function will compare *string1* to *string2*, and return a whole number which will be 0 if they are equal, otherwise the position of the first character at which the two strings differ is returned. The comparison is case-sensitive, and leading and trailing space do matter.

If the strings are of unequal length, the shorter string will be padded at the right hand end with the *padchar* character to the length of the longer string before the comparison. If a *padchar* is not specified, <space> is used.

```
COMPARE('FooBar', 'Foobar')         ->    '4'
COMPARE('Foobar', 'Foobar')         ->    '0'
COMPARE('Foobarrr', 'Fooba')        ->    '6'
COMPARE('Foobarrr', 'Fooba', 'r' )  ->    '0'
```

CONDITION([*option*])

Returns information about the current trapped condition. A condition becomes the current trapped condition when a condition handler is called (by `CALL` or `SIGNAL`) to handle the condition. The parameter *option* specifies what sort of information to return:

[C]

(Condition) The name of the current trapped condition is return, this will be one of the condition named legal to SIGNAL ON, like SYNTAX, HALT, NOVALUE, NOTREADY, ERROR or FAILURE.

[D]

(Description) A text describing the reason for the condition. What to put into this variable is implementation and system dependent.

[I]

(Instruction) Returns either CALL or SIGNAL, depending on which method was current when the condition was trapped.

[S]

(State) The current state of the current trapped condition. This can be one of ON, OFF or DELAY. Note that this option reflect the current state, which may change, not the state at the time when the condition was trapped.

For more information on conditions, consult the chapter Conditions. Note that condition may in several ways be dependent on the implementation and system, so read system and implementation dependent information too.

COPIES(*string*,*copies*)

Returns a string with *copies* concatenated copies of *string*. *Copies* must be a non-negative whole number. No extra space is added between the copies.

```
COPIES('Foo', 3)  ->    'FooFooFoo'
COPIES('*', 16)   ->    '****************'
COPIES('Bar ', 2) ->    'Bar Bar '
COPIES('', 10000) ->    ''
```

COUNTSTR(*needle*,*haystack*)

This function was introduced with the REXX ANSI Standard. It returns a count of the number of occurrences of *needle* in *haystack* that do not overlap.

```
COUNTSTR('', '')                    ->    0
COUNTSTR('a', 'abcdef')             ->    1
COUNTSTR(0, 0)                      ->    1
COUNTSTR('a', 'def')                ->    0
COUNTSTR('a', '')                   ->    0
COUNTSTR('', 'def')                 ->    0
COUNTSTR('abc', 'abcdef')           ->    1
COUNTSTR('abcdefg', 'abcdef'        ->    0
COUNTSTR('abc', 'abcdefabccdabcd')  ->    3
```

DATATYPE(*string*[,*option*])

With only one parameter, this function identifies the "datatype" of *string*. The value returned will be "NUM" if *string* is a valid REXX number. Otherwise, "CHAR" is returned. Note that the interpretation of whether *string* is a valid number will depend on the current setting of NUMERIC.

If *option* is specified too, it will check if *string* is of a particular datatype, and return either "1" or "0" depending on whether *string* is or is not, respectively, of the specified datatype. The possible values of *option* are:

[A]

(Alphanumeric) Consisting of only alphabetic characters (in upper, lower or mixed case) and decimal digits.

[B]

(Binary) Consisting of only the two binary digits `0` and `1`. Note that blanks are not allowed within *string*, as would have allowed been within a binary string.

[L]

(Lower) Consisting of only alphabetic characters in lower case.

[M]

(Mixed) Consisting of only alphabetic characters, but the case does not matter (i.e. upper, lower or mixed.)

[N]

(Numeric) If *string* is a valid REXX number, i.e. `DATATYPE(`*`string`*`)` would return `NUM`.

[S]

(Symbolic) Consists of characters that are legal in REXX symbols. Note that this test will pass several strings that are not legal symbols. The characters includes plus, minus and the decimal point.

[U]

(Upper) Consists of only upper case alphabetic characters.

[W]

(Whole) If *string* is a valid REXX whole number under the current setting of `NUMERIC`. Note that `13.0` is a whole number since the decimal part is zero, while `13E+1` is not a whole number, since it must be interpreted as 130 plus/minus 5.

[X]

(Hexadecimal) Consists of only hexadecimal digits, i.e. the decimal digits 0-9 and the alphabetic characters A-F in either case (or mixed.) Note that blanks are not allowed within *string*, as it would have been within a hexadecimal string.

If you want to check whether a string is suitable as a variable name, you should consider using the `SYMBOL()` function instead, since the `Symbolic` option only verifies which characters *string* contains, not the order. You should also take care to watch out for lower case alphabetic characters, which are allowed in the tail of a compound symbol, but not in a simple or stem symbol or in the head of compound symbol.

Also note that the behavior of the options `A`, `L`, `M` and `U` might depend on the setting of language, if you are using an interpreter that supports national character sets.

```
DATATYPE(' - 1.35E-5 ')            ->    'NUM'
DATATYPE('1E999999999')            ->    'CHAR'
DATATYPE('1E9999999999')           ->    'CHAR'
DATATYPE('!@#&#$(&*%`')            ->    'CHAR'
DATATYPE('FooBar', 'A')            ->    '1'
DATATYPE('Foo Bar', 'A')           ->    '0'
DATATYPE('010010111101', 'B')      ->    '1'
DATATYPE('0100 1011 1101', 'B')    ->    '0'
DATATYPE('foobar', 'L')            ->    '1'
DATATYPE('FooBar', 'M')            ->    '1'
DATATYPE(' -34E3 ', 'N')           ->    '1'
DATATYPE('A_SYMBOL!?!', 'S')       ->    '1'
DATATYPE('1.23.39E+4.5', 'S')      ->    '1'
DATATYPE('Foo bar', 'S')           ->    '0'
DATATYPE('FOOBAR', 'U')            ->    '1'
DATATYPE('123deadbeef', 'X')       ->    '1'
```

DATE([*option_out* [,*date* [,*option_in*]]])

This function returns information relating to the current date. If the *option_out* character is specified, it will set the format of the return string. The default value for *option_out* is "N".

Possible options are:

[B]

(Base) The number of complete days from January 1st 0001 until yesterday inclusive, as a whole number. This function uses the Gregorian calendar extended backwards. Therefore Date('B') // 7 will equal the day of the week where 0 corresponds to Monday and 6 Sunday.

[C]

(Century) The number of days in this century from January 1st –00 until today, inclusive. The return value will be a positive integer.

[D]

(Days) The number of days in this year from January 1st until today, inclusive. The return value will be a positive integer.

[E]

(European) The date in European format, i.e. " dd/mm/yy". If any of the numbers is single digit, it will have a leading zero.

[M]

(Month) The unabbreviated name of the current month, in English.

[N]

(Normal) Return the date with the name of the month abbreviated to three letters, with only the first letter in upper case. The format will be "dd Mmm yyyy", where Mmm is the month abbreviation (in English) and dd is the day of the month, without leading zeros.

[O]

(Ordered) Returns the date in the ordered format, which is " yy/mm/dd".

[S]

(Standard) Returns the date according the format specified by International Standards Organization Recommendation ISO/R 2014-1971 (E). The format will be "yyyymmdd", and each part is padded with leading zero where appropriate.

[U]

(USA) Returns the date in the format that is normally used in USA, i.e. " mm/dd/yy", and each part is padded with leading zero where appropriate.

[W]

(Weekday) Returns the English unabbreviated name of the current weekday for today. The first letter of the result is in upper case, the rest is in lower case.

Note that the "C" option is present in REXX language level 3.50, but was removed in level 4.00. The new "B" option should be used instead. When porting code that use the "C" option to an interpreter that only have the " B" option, you will can use the conversion that January 1st 1900 is day 693595 in the Gregorian calendar.

Note that none of the formats in which DATE() return its answer are effected by the settings of NUMERIC. Also note that if there are more than one call to DATE() (and TIME()) in a single clause of REXX code, all of them will use the same basis data for calculating the date (and time).

If the REXX interpreter contains national support, some of these options may return different output for the names of months and weekdays.

Assuming that today is January 6th 1992:

```
DATE('B')   ->    '727203'
DATE('C')   ->    '33609'
DATE('D')   ->    '6'
DATE('E')   ->    '06/01/92'
DATE('M')   ->    'January'
DATE('N')   ->    '6 Jan 1992'
DATE('O')   ->    '92/01/06'
```

```
DATE('S')   ->    '19920106'
DATE('U')   ->    '01/06/92'
DATE('W')   ->    'Monday'
```

If the *date* option is specified, the function provides for date conversions. The optional *option_in* specifies the format in which *date* is supplied. The possible values for *option_in* are: **BDEOUNS.**
The default value for *option_in* is **N**.

```
DATE('O', '13 Feb 1923')      ->    '23/02/13'
DATE('O', '06/01/50', 'U')    ->    '50/06/01'
```

If the *date* supplied does not include a century in its format, then the result is chosen to make the year within 50 years past or 49 years future of the current year.

The date conversion capability of the DATE BIF was introduced with the ANSI standard.

DELSTR(*string*,*start*[,*length*])

Returns *string*, after the substring of length *length* starting at position *start* has been removed. The default value for *length* is the rest of the string. *Start* must be a positive whole number, while *length* must be a non-negative whole number. It is not an error if *start* or *length* (or a combination of them) refers to more characters than *string* holds

```
DELSTR('Foobar', 3)     ->    'Foo'
DELSTR('Foobar', 3, 2)  ->    'Foor'
DELSTR('Foobar', 3, 4)  ->    'Foo'
DELSTR('Foobar', 7)     ->    'Foobar'
```

DELWORD(*string*,*start*[,*length*])

Removes *length* words and all blanks between them, from *string*, starting at word number *start*. The default value for *length* is the rest of the string. All consecutive spaces immediately after the last deleted word, but no spaces before the first deleted word is removed. Nothing is removed if *length* is zero.

The valid range of *start* is the positive whole numbers; the first word in *string* is numbered `1`. The valid range of *length* is the non-negative integers. It is not an error if *start* or *length* (or a combination of them) refers to more words than *string* holds.

```
DELWORD('This is a test', 3)        -> 'This is '
DELWORD('This is a test', 2, 1)     -> 'This a test'
DELWORD('This is a test', 2, 5)     -> 'This'
DELWORD('This is a test', 1, 3)     -> 'test' /*No leading space*/
```

DIGITS()

Returns the current precision of arithmetic operations. This value is set using the `NUMERIC` statement. For more information, refer to the documentation on `NUMERIC`.

```
DIGITS()    ->    '9'  /* Maybe */
```

D2C(*integer*[,*length*])

Returns a (packed) string, that is the character representation of *integer*, which must be a whole number, and is governed by the settings of `NUMERIC`, not of the internal precision of the built-in functions. If *length* is specified the string returned will be *length* bytes long, with sign extension. If *length* (which must be a non-negative whole number) is not large enough to hold the result, an error is reported.

If *length* is not specified, *integer* will be interpreted as an unsigned number, and the result will have no leading <nul> characters. If *integer* is negative, it will be interpreted as a two's complement, and *length* must be specified.

```
D2C(0)      ->    ''
D2C(127)    ->    '7F'x
D2C(128)    ->    '80'x
D2C(128, 3) ->    '000080'x
D2C(-128)   ->    '80'x
D2C(-10, 3) ->    'fffff5'x
```

D2X(*integer*[,*length*])

Returns a hexadecimal number that is the hexadecimal representation of *integer*. *Integer* must be a whole number under the current settings of NUMERIC, it is not effected by the precision of the built-in functions.

If *length* is not specified, then *integer* must be non-negative, and the result will be stripped of any leading zeros.

If *length* is specified, then the resulting string will have that length. If necessary, it will be sign-extended on the left side to make it the right length. If *length* is not large enough to hold *integer*, an error is reported.

```
D2X(0)      ->    '0'
D2X(127)    ->    '7F'
D2X(128)    ->    '80'
D2X(128, 5) ->    '00080'x
D2X(-128)   ->    '80'x
D2X(-10, 5) ->    'ffff5'x
```

ERRORTEXT(*errno*)

Returns the REXX error message associated with error number *errno*.

If the error message is not defined, a nullstring is returned.

The error messages in REXX might be slightly different between the various implementations. The standard says that *errno* must be in the range 0-99, but in some implementations it might be within a less restricted range which gives room for system specific messages. You should in general not assume that the wordings and ordering of the error messages are constant between implementations and systems.

```
ERRORTEXT(20)     ->    'Symbol expected'
ERRORTEXT(30)     ->    'Name or string too long'
ERRORTEXT(40)     ->    'Incorrect call to routine'
```

FORM()

Returns the current "form", in which numbers are presented when exponential form is used. This might be either SCIENTIFIC (the default) or ENGINEERING. This value is set through the NUMERIC FORM clause. For more information, see the documentation on NUMERIC.

```
FORM()      ->    'SCIENTIFIC'  /* Maybe */
```

FORMAT(*number*[,[*before*][,[*after*][,[*expp*][,[*expt*]]]]])

This function is used to control the format of numbers, and you may request the size and format in which the number is written. The parameter *number* is the number to be formatted, and it must be a valid REXX number. note that before any conversion or formatting is done, this number will be normalized according to the current setting of NUMERIC.

The *before* and *after* parameters determines how many characters that are used before and after the decimal point, respectively. Note that *before* does **not** specify the number of digits in the integer part, it specifies the size of the field

in which the integer part of the number is written. Remember to allocate space in this field for a minus too, if that is relevant. If the field is not long enough to hold the integer part (including a minus if relevant), an error is reported.

The *after* parameter will dictate the size of the field in which the fractional part of the number is written. The decimal point itself is not a part of that field, but the decimal point will be omitted if the field holding the fractional part is empty. If there are less digits in the number than the size of the field, it is padded with zeros at the right. If there is more digits then it is possible to fit into the field, the number will be rounded (not truncated) to fit the field.

Before must at least be large enough to hold the integer part of *number*. Therefore it can never be less than `1`, and never less than `2` for negative numbers. The integer field will have no leading zeros, except a single zero digit if the integer part of *number* is empty.

The parameter *expp* the size of the field in which the exponent is written. This is the size of the numeric part of the exponent, so the "`E`" and the sign comes in addition, i.e. the real length if the exponent is two more than *expp* specifies. If *expp* is zero, it signalizes that exponential form should not be used. *Expp* must be a non-negative whole number. If *expp* is positive, but not large enough to hold the exponent, an error is reported.

Expt is the trigger value that decides when to switch from simple to exponential form. Normally, the default precision (`NUMERIC DIGITS`) is used, but if *expt* is set, it will override that. Note that if *expt* is set to zero, exponential form will always be used. However, if *expt* tries to force exponential form, simple form will still be used if *expp* is zero. Negative values for *expt* will give an error. Exponential form is used if more digits than *expt* is needed in the integer part, or more than twice *expt* digits are needed in the fractional part.

Note that the *after* number will mean different things in exponential and simple form. If *after* is set to e.g. `3`, then in simple form it will force the precision to 0.001, no matter the magnitude of the number. If in exponential form, it will force the number to 4 digits precision.

```
FORMAT(12.34,3,4)       ->    ' 12.3400'
FORMAT(12.34,3,,3,0)    ->    '  1.234E+001'
FORMAT(12.34,3,1)       ->    ' 12.3400'
FORMAT(12.34,3,0)       ->    ' 12.3'
FORMAT(12.34,3,4)       ->    ' 12'
FORMAT(12.34,,,,0)      ->    '1.234E+1'
FORMAT(12.34,,,0)       ->    '12.34'
FORMAT(12.34,,,0,0)     ->    '12.34'
```

FUZZ()

Returns the current number of digits which are ignored when comparing numbers, during operations like `=` and `>`. The default value for this is `0`. This value is set using the `NUMERIC FUZZ` statement, for more information see that.

```
FUZZ()      ->    '0'  /* Maybe */
```

INSERT(*string1*,*string2*[,*position*[,*length*[,*padchar*]]])

Returns the result of inserting *string1* into a copy of *string2*. If *position* is specified, it marks the character in *string2* which *string1* it to be inserted after. *Position* must be a non-negative whole number, and it defaults to `0`, which means that *string2* is put in front of the first character in *string1*.

If *length* is specified, *string1* is truncated or padded on the right side to make it exactly *length* characters long before it is inserted. If padding occurs, then *padchar* is used, or <space> if *padchar* is undefined.

```
INSERT('first', 'SECOND')               ->    'SECONDfirst'
INSERT('first', 'SECOND', 3)            ->    'fiSECONDrst'
INSERT('first', 'SECOND', 3, 10)        ->    'fiSECOND    rst'
INSERT('first', 'SECOND', 3, 10, '*')   ->    'fiSECOND****rst'
INSERT('first', 'SECOND', 3, 4)         ->    'fiSECOrst'
INSERT('first', 'SECOND', 8)            ->    'first  SECOND'
```

LASTPOS(*needle*,*haystack*[,*start*])

Searches the string *haystack* for the string *needle*, and returns the position in *haystack* of the first character in the substring that matched *needle*. The search is started from the right side, so if *needle* occurs several times, the last occurrence is reported.

If *start* is specified, the search starts at character number *start* in *haystack*. Note that the standard only states that the search starts at the *start*th character. It is not stated whether a match can partly be to the right of the *start* position, so some implementations may differ on that point.

```
LASTPOS('be', To be or not to be')        ->    17
LASTPOS('to', to be or not to be', 10)    ->    3
LASTPOS('is', to be or not to be')        ->    0
LASTPOS('to', to be or not to be', 0)     ->    0
```

LEFT(*string*,*length*[,*padchar*])

Returns the *length* leftmost characters in *string*. If *length* (which must be a non-negative whole number) is greater than the length of *string*, the result is padded on the right with <space> (or *padchar* if that is specified) to make it the correct length.

```
LEFT('Foo bar', 5)           ->    'Foo b'
LEFT('Foo bar', 3)           ->    'Foo'
LEFT('Foo bar', 10)          ->    'Foo bar   '
LEFT('Foo bar', 10, '*')     ->    'Foo bar***'
```

LENGTH(*string*)

Returns the number of characters in *string*.

```
LENGTH('')              ->    '0'
LENGTH('Foo')           ->    '3'
LENGTH('Foo bar')       ->    '7'
LENGTH(' foo  bar ')    ->    '10'
```

LINEIN([*streamid*][,[*line*][,*count*]])

Returns a line read from a file. When only *streamid* is specified, the reading starts at the current read position and continues to the first End-Of-Line (EOL) mark. Afterwards, the current read position is set to the character after the EOL mark which terminated the read-operation. If the operating system uses special characters for EOL marks, these are not returned by as a part of the string read..

The default value for *streamid* is default input stream. The format and range of the string *streamid* are implementation dependent.

The *line* parameter (which must be a positive whole number) might be specified to set the current position in the file to the beginning of line number *line* before the read operation starts. If *line* is unspecified, the current position will not be changed before the read operation. Note that *line* is only valid for persistent steams. For transient streams, an error is reported if *line* is specified. The first line in the stream is numbered `1`.

Count specifies the number of lines to read. However, it can only take the values `0` and `1`. When it is `1` (which is the default), it will read one line. When it is `0` it will not read any lines, and a nullstring is returned. This has the effect of setting the current read position of the file if *line* was specified.

What happens when the functions finds a End-Of-File (EOF) condition is to some extent implementation dependent. The implementation may interpret the EOF as an implicit End-Of-Line (EOL) mark is none such was explicitly present. The implementation may also choose to raise the `NOTREADY` condition flag (this condition is new from REXX language level 4.00).

Whether or not *stream* must be explicitly opened before a read operation can be performed, is implementation dependent. In many implementations, a read or write operation will implicitly open the stream if not already open.

Assuming that the file `/tmp/file` contains the three lines: "*First line*", *Second line*" and "*Third line*":

```
LINEIN('/tmp/file', 1)        ->    'First line'
LINEIN('/tmp/file')           ->    'Second line'
LINEIN('/tmp/file', 1, 0)     ->    ''  /* But sets read position */
LINEIN('/tmp/file')           ->    'First line'
LINEIN()                      ->    'Hi, there!'  /* maybe */
```

LINEOUT([*streamid*][,[*string*][,*line*]])

Returns the number of lines remaining after having positioned the stream *streamid* to the start of line *line* and written out *string* as a line of text. If *streamid* is omitted, the default output stream is used. If *line* (which must be a positive whole number) is omitted, the stream will not be repositioned before the write. If *string* is omitted, nothing is written to the stream. If *string* is specified, a system-specific action is taken after it has been written to stream, to mark a new line.

The format and contents of the first parameter will depend upon the implementation and how it names streams. Consult implementation-specific documentation for more information.

If *string* is specified, but not *line*, the effect is to write *string* to the stream, starting at the current write position. If *line* is specified, but not *string*, the effect is only to position the stream at the new position. Note that the *line* parameter is only legal if the stream is persistent; you can not position the current write position for transient streams.

If neither *line* nor *string* is specified, the standard requires that the current write position is set the end of the stream, and implementation specific side-effects may occur. In practice, this means that an implementation can use this situation to do things like closing the stream, or flushing the output. Consult the implementation specific documentation for more information.

Also note that the return value of this functions may be of little or no value, If just a half line is written, `1` may still be returned, and there are no way of finding out how much (if any) of *string* was written. If *string* is not specified, the return value will always be `0`, even if `LINEOUT()` was not able to correctly position the stream.

If it is impossible to correctly write *string* to the stream, the `NOTREADY` flag will be raised. It is not defined whether or not the `NOTREADY` flag is raised when `LINEOUT()` is used for positioning, and this is not possible.

Note that if you write *string* to a line in the middle of the stream (i.e. *line* is less than the total number of lines in the stream), then the behavior is system and implementation specific. Some systems will truncate the stream after the newly written line, other will only truncate if the newly written line has a different length than the old line which it replaced, and yet other systems will overwrite and never truncate.

In general, consult your system and implementation specific documentation for more information about this function. You can safely assume very little about how it behaves.

```
LINEOUT(, 'First line')                  ->    '1'
LINEOUT('/tmp/file', 'Second line', 2)   ->    '1'
LINEOUT('/tmp/file', 'Third line')       ->    '1'
LINEOUT('/tmp/file', 'Fourth line', 4)   ->    '0'
```

LINES([*streamid*])

Returns the number of complete lines remaining in the named file *stream*. A complete line is not really as complete as the name might indicate; a complete line is zero or more characters, followed by an End-Of-Line (EOL) marker. So, if you have read half a line already, you still have a "complete" line left. Note that it is not defined what to do with a half-finished line at the end of a file. Some interpreters might interpret the End-Of-File as an implicit EOL mark too, while others might not.

The format and contents of the stream *streamid* is system and implementation dependent. If omitted, the default input stream will be used.

The standard says that if it is impossible (or maybe just difficult) to accurately count the remaining lines in a stream, `LINES()` can return `0` for no more lines, and `1` for more lines. This probably applies for all transient streams, as the interpreter can not reposition in these files, and can therefore not count the number of remaining lines. It can also apply for persistent files, if the operation of counting the lines left in the file is very time-consuming.

As a result, defensive programming indicates that you can safely only assume that this function will return either `0` or a non-zero result. If you want to use the non-zero result to more than just an indicator on whether more lines are available, you must check that it is larger than one. If so, you can safely assume that it hold the number of available lines left.

As with all the functions operating on streams, you can safely assume very little about this function, so consult the system and implementation specific documentation.

```
LINES()                 ->    '1'  /* Maybe */
LINES()                 ->    '0'  /* Maybe */
LINES('/tmp/file')      ->    '2'  /* Maybe */
LINES('/tmp/file')      ->    '1'  /* Maybe */
```

MAX(*number1*[,*number2*]...)

Takes any positive number of parameters, and will return the parameter that had the highest numerical value. The parameters may be any valid REXX number. The number that is returned, is normalized according to the current settings of `NUMERIC`, so the result need not be strictly equal to any of the parameters.

Actually, the standard says that the value returned is the first number in the parameter list which is equal to the result of adding a positive number or zero to any of the other parameters. Note that this definition opens for "strange" results if you are brave enough to play around with the settings of `NUMERIC FUZZ`.

```
MAX(1, 2, 3, 5, 4)      ->    '5'
MAX(6)                  ->    '6'
MAX(-4, .001E3, 4)      ->    '4'
MAX(1, 2, 05.0, 4)      ->    '5.0'
```

MIN(*number*[,*number*]...)

Like `MAX()`, except that the lowest numerical value is returned. For more information, see `MAX()`.

```
MAX(5, 4, 3, 1, 2)      ->    '1'
MAX(6)                  ->    '6'
MAX(-4, .001E3, 4)      ->    '-4'
MAX(1, 2, 05.0E-1, 4)   ->    '0.50'
```

OVERLAY(*string1*,*string2*[,[*start*][,[*length*][,*padchar*]]])

Returns a copy of *string2*, totally or partially overwritten by *string1*. If these are the only arguments, the overwriting starts at the first character in *string2*.

If *start* is specified, the first character in *string1* overwrites character number *start* in *string2*. *Start* must be a positive whole number, and defaults to `1`, i.e. the first character of *string1*. If the *start* position is to the right of the end of *string2*, then *string2* is padded at the right hand end to make it *start*-1 characters long, before *string1* is added.

If *length* is specified, then *string2* will be stripped or padded at the right hand end to match the specified length. For padding (of both strings) *padchar* will be used, or <space> if *padchar* is unspecified. *Length* must be non-negative, and defaults to the length of *string1*.

```
OVERLAY('NEW', 'old-value')              ->    'NEW-value'
```

```
OVERLAY('NEW', 'old-value', 3)           ->    'oldNEWlue'
OVERLAY('NEW', 'old-value', 3, 5)        ->    'oldNEW  e'
OVERLAY('NEW', 'old-value', 3, 5), '*')  ->    'oldNEW**e'
OVERLAY('NEW', 'old-value', 3, 2)        ->    'oldNEalue'
OVERLAY('NEW', 'old-value', 8)           ->    'old-valuNEW'
OVERLAY('NEW', 'old-value', 10)          ->    'old-value  NEW'
OVERLAY('NEW', 'old-value', 8,, '*')     ->    'old-value**NEW'
OVERLAY('NEW', 'old-value', 8, 5, '*')   ->    'old-value**NEW**'
```

POS(*needle*,*haystack*[,*start*])

Seeks for an occurrence of the string *needle* in the string *haystack*. If *needle* is not found, then `0` is returned. Else, the position in *haystack* of the first character in the part that matched is returned, which will be a positive whole number. If *start* (which must be a positive whole number) is specified, the search for *needle* will start at position *start* in *haystack*.

```
POS('be', 'to be or not to be')     ->    3
POS('to', 'to be or not to be', 10) ->    17
POS('is', 'to be or not to be')     ->    0
POS('to', 'to be or not to be', 18) ->    0
```

QUEUED()

Returns the number of lines currently in the external data queue (the "stack"). Note that the stack is a concept external to REXX, this function may depend on the implementation and system Consult the system specific documentation for more information.

```
QUEUED()    ->    '0'  /* Maybe */
QUEUED()    ->    '42'  /* Maybe */
```

RANDOM(*max*)
RANDOM([*min*][,[*max*][,*seed*]])

Returns a pseudo-random whole number. If called with only the first parameter, the first format will be used, and the number returned will be in the range `0` to the value of the first parameter, inclusive. Then the parameter *max* must be a non-negative whole number, not greater than 100000.

If called with more than one parameter, or with one parameter, which is not the first, the second format will be used. Then *min* and *max* must be whole numbers, and *max* can not be less than *min*, and the difference *max*-*min* can not be more than 100000. If one or both of them is unspecified, the default for *min* is `0`, and the default for *max* is `999`. Note that both *min* and *max* are allowed to be negative, as long as their difference is within the requirements mentioned.

If *seed* is specified, you may control which numbers the pseudo-random algorithm will generate. If you do not specify it, it will be set to some "random" value at the first call to `RANDOM()` (typically a function of the time). When specifying *seed*, it will effect the result of the current call to `RANDOM()`.

The standard does not require that a specific method is to be used for generating the pseudo-random numbers, so the reproducibility can only be guaranteed as long as you use the same implementation on the same machine, using the same operating system. If any of these change, a given *seed* may produce a different sequence of pseudo-random numbers.

Note that depending on the implementation, some numbers might have a slightly increased chance of turning up than other. If the REXX implementation uses a 32 bit pseudo-random generator provided by the operating system and returns the remainder after integer dividing it by the difference of *min* and *max*, low numbers are favored if the 2^32 is not a multiple of that difference. Supposing that the call is `RANDOM(100000)` and the pseudo-random generator generates any 32 bit number with equal chance, the change of getting a number in the range 0 -67296 is about 0.000010000076, while the changes of getting a number in the range 67297 -100000 is about 0.000009999843.

A much worse problem with pseudo-random numbers are that they sometimes do not tend to be random at all. Under one operating system (name withheld to protect the guilty), the system's pseudo-random routine returned numbers where the last binary digit alternated between 0 and 1. On that machine, `RANDOM(1)` would return the series 0, 1, 0, 1, 0, 1, 0, 1 etc., which is hardly random at all. You should therefore never trust the pseudo-random routine to give you random numbers.

Note that due to the special syntax, there is a big difference between using `RANDOM(10)` and `RANDOM(10,)`. The former will give a pseudo-random number in the range 0 - 10, while the latter will give a pseudo-random number in the range 10 - 999.

Also note that it is not clear whether the standard allows *min* to be equal to *max*, so to program compatible, make sure that *max* is always larger than *min*.

```
RANDOM()          –>    '123' /*Between 0 and 999*/
RANDOM(10)        –>    '5'   /*Between 0 and 10*/
RANDOM(, 10)      –>    '3'   /*Between 0 and 10*/
RANDOM(20, 30)    –>    '27'  /*Between 20 and 30*/
RANDOM(,, 12345)  –>    '765' /*Between 0 and 999, and sets seed*/
```

REVERSE(*string*)

Returns a string of the same length as *string*, but having the order of the characters reversed.

```
REVERSE('FooBar')       –>    'raBooF'
REVERSE('  Foo Bar')    –>    'raB ooF  '
REVERSE('3.14159')      –>    '95141.3'
```

RIGHT(*string*,*length*[,*padchar*])

Returns the *length* rightmost characters in *string*. If *length* (which must be a non-negative whole number) is greater than the length of *string* the result is padded on the left with the necessary number of *padchar*s to make it as long as *length* specifies. *Padchar* defaults to <space>.

```
RIGHT('Foo bar', 5)          –>    'o bar'
RIGHT('Foo bar', 3)          –>    'bar'
RIGHT('Foo bar', 10)         –>    '   Foo bar'
RIGHT('Foo bar', 10, '*')    –>    '***Foo bar'
```

SIGN(*number*)

Returns either `-1`, `0` or `1`, depending on whether *number* is negative, zero, or positive, respectively. *Number* must be a valid REXX number, and are normalized according to the current settings of `NUMERIC` before comparison.

```
SIGN(-12)         –>    '-1'
SIGN(42)          –>    '1'
SIGN(-0.00000012) –>    '-1'
SIGN(0.000)       –>    '0'
SIGN(-0.0)        –>    '0'
```

SOURCELINE([*lineno*])

If *lineno* (which must be a positive whole number) is specified, this function will return a string containing a copy of the REXX script source code on that line. If *lineno* is greater than the number of lines in the REXX script source code, an error is reported.

If *lineno* is unspecified, the number of lines in the REXX script source code is returned.

Note that from REXX language level 3.50 to 4.00, the requirements of this function were relaxed to simplify execution when the source code is not available (compiled or pre-parsed REXX). An implementation might make two simplifications: to return `0` if called without parameter. If so, any call to `SOURCELINE()` with a parameter will generate an error. The other simplification is to return a nullstring for any call to `SOURCELINE()` with a legal parameter.

Note that the code executed by the `INTERPRET` clause can not be retrieved by `SOURCELINE()`.

```
SOURCELINE()      ->    '42'  /*Maybe */
SOURCELINE(1)     ->    '/*This Rexx script will ... */'
SOURCELINE(23)    ->    'var = 12'  /*Maybe */'
```

SPACE(*string*[,[*length*][,*padchar*]])

With only one parameter *string* is returned, stripped of any trailing or leading blanks, and any consecutive blanks inside *string* translated to a single <space> character (or *padchar* if specified).

Length must be a non-negative whole number. If specified, consecutive blanks within *string* is replaced by exactly *length* instances of <space> (or *padchar* if specified). However, *padchar* will only be used in the output string, in the input string, blanks will still be the "magic" characters. As a consequence, if there exist any *padchar*s in *string*, they will remain untouched and will not affect the spacing.

```
SPACE(' Foo  bar ')           ->    'Foo bar'
SPACE(' Foo  bar ', 2)        ->    'Foo  bar'
SPACE(' Foo  bar ',, '*')     ->    'Foo*bar'
SPACE('Foo bar', 3, '-')      ->    'Foo---bar'
SPACE('Foo  bar',, 'o')       ->    'Fooobar'
```

STREAM(*streamid*[,*option*[,*command*]])

This function was added to REXX in language level 4.00. It provides a general mechanism for doing operations on streams. However, very little is specified about how the internal of this function should work, so you should consult the implementation specific documentation for more information.

The *streamid* identifies a stream. The actual contents and format of this string is implementation dependent.

The *option* selects one of several operations which `STREAM()` is to perform. The possible operations are:

[C]
: (Command) If this option is selected, a third parameter must be present, *command*, which is the command to be performed on the stream. The contents of *command* is implementation dependent. For Regina, the valid commands follow. Commands consist of one or more space separated words.

[D]
: (Description) Returns a description of the state of *streamid*. The return value is implementation dependent.

[S]
: (Status) Returns a state which describes the state of *streamid*. The standard requires that it is one of the following: `ERROR`, `NOTREADY`, `READY` and `UNKNOWN`. The meaning of these are described in the chapter; **Stream Input and Output**.

Note that the options `Description` and `Status` really have the same function, but that `Status` in general is implementation independent, while `Description` is implementation dependent.

The *command* specifies the command to be performed on *streamid*. The possible operations are:

[READ]
: Open for read access. The file pointer will be positioned at the start of the file, and only read operations are allowed. This command is Regina-specific; use **OPEN READ** in its place.

[WRITE]

Open for write access and position the current write position at the end of the file. An error is returned if it was not possible to get appropriate access. This command is Regina-specific; use **OPEN WRITE** in its place.

[APPEND]

Open for append access and position the current write position at the end of the file. An error is returned if it was not possible to get appropriate access. This command is Regina-specific; use **OPEN WRITE APPEND** in its place.

[UPDATE]

Open for append access and position the current write position at the end of the file. An error is returned if it was not possible to get appropriate access. This command is Regina-specific; use **OPEN BOTH** in its place.

[CREATE]

Open for write access and position the current write position at the start of the file. An error is returned if it was not possible to get appropriate access. This command is Regina-specific; use **OPEN WRITE REPLACE** in its place.

[CLOSE]

Close the stream, flushing any pending writes. An error is returned if it was not possible to get appropriate access.

[FLUSH]

Flush any pending write to the stream. An error is returned if it was not possible to get appropriate access.

[STATUS]

Returns status information about the stream in human readable form that Regina stores about the stream.

[FSTAT]

Returns status information from the operating system about the stream.

[RESET]

Resets the stream after an error. Only streams that are resettable can be reset.

[READABLE]

Returns 1 if the stream is readable by the user or 0 otherwise.

[WRITABLE]

Returns 1 if the stream is writeable by the user or 0 otherwise.

[EXECUTABLE]

Returns 1 if the stream is executable by the user or 0 otherwise.

[QUERY]

Returns information about the named stream. If the named stream does not exists, then the empty string is returned. This command is further broken down into the following sub-commands:

DATETIME	returns the date and time of last modification of the stream in Rexx US Date format; MM-DD-YY HH:MM:SS.
EXISTS	returns the fully-qualified file name of the specified stream.
HANDLE	returns the internal file handle of the stream. This will only return a valid value if the stream was opened explicitly or implicitly by Regina.
POSITION READ	returns the current read position of the open stream. This is expressed in characters, so returns the same value as POSITION CHAR.
POSITION WRITE	returns the current write position of the open stream. This is expressed in characters.
POSITION CHAR	returns the current read position of the open stream. This is expressed in characters.
POSITION LINE	returns the current read position of the open stream. This is expressed in lines.
POSITION SYS	returns the current read position of the open stream as the operati ng reports it. This is expressed in characters.
SIZE	returns the size, expressed in characters, of the persistent stream.
STREAMTYPE	returns the type of the stream. One of TRANSIENT, PERSISTENT or UNKNOWN is returned.
TIMESTAMP	returns the date and time of last modifcation of the stream. The format of the string returned is YYYY-MM-DD HH:MM:SS.

[OPEN]

Opens the stream in the optional mode specified. If no optional mode is specified, the default is **OPEN BOTH**.

READ	The file pointer will be positioned at the start of the file, and only read operations are allowed.
WRITE	Open for write access and position the current write pointer at the end of the file. On platforms where it is not possible to open a file for write

	without also allowing reads, the read pointer will be positioned at the start of the file. An error is returned if it was not possible to get appropriate access.
BOTH	Open for read and write access. Position the current read pointer at the start of the file, and the current write pointer at the end of the file. An error is returned if it was not possible to get appropriate access.
WRITE APPEND	Open for write access and position the write pointer at the end of the file. On platforms where it is not possible to open a file for write without also allowing reads, the read pointer will be positioned at the start of the file.
WRITE REPLACE	Open for write access and position the current write position at the start of the file. On platforms where it is not possible to open a file for write without also allowing reads, the read pointer will be positioned at the start of the file. This operation will clear the contents of the file. An error is returned if it was not possible to get appropriate access.
BOTH APPEND	Open for read and write access. Position the current read position at the start of the file, and the current write position at the end of the file. An error is returned if it was not possible to get appropriate access.
BOTH REPLACE	Open for read and write access. Position both the current read and write pointers at the start of the file. An error is returned if it was not possible to get appropriate access.

STRIP(*string*[,[*option*][,*char*]])

Returns *string* after possibly stripping it of any number of leading and/or trailing characters. The default action is to strip off both leading and trailing blanks. If *char* (which must be a string containing exactly one character) is specified, that character will be stripped off instead of blanks. Inter-word blanks (or *char*s if defined, that are not leading of trailing) are untouched.

If *option* is specified, it will define what to strip. The possible values for *option* are:

[L]

(Leading) Only strip off leading blanks, or *char*s if specified.

[T]

(Trailing) Only strip off trailing blanks, or *char*s if specified.

[B]

(Both) Combine the effect of L and T, that is, strip off both leading and trailing blanks, or *char*s if it is specified. This is the default action.

```
STRIP(‘ Foo  bar ‘)          –>    ‘Foo  bar’
STRIP(‘ Foo  bar ‘, ‘L’)     –>    ‘Foo  bar ‘
STRIP(‘ Foo  bar ‘, ‘t’)     –>    ‘ Foo  bar’
STRIP(‘ Foo  bar ‘, ‘Both’)  –>    ‘Foo  bar’
STRIP(‘0.1234500’,, ‘0’)     –>    ‘.12345’
STRIP(‘0.1234500 ‘,, ‘0’)    –>    ‘.1234500’
```

SUBSTR(*string*,*start*[,[*length*][,*padchar*]])

Returns the substring of *string* that starts at *start*, and has the length *length*. *Length* defaults to the rest of the string. *Start* must be a positive whole, while *length* can be any non-negative whole number.

It is not an error for *start* to be larger than the length of *string*. If *length* is specified and the sum of *length* and *start* minus 1 is greater that the length of *string*, then the result will be padded with *padchar*s to the specified length. The default value for *padchar* is the <space> character.

```
SUBSTR(‘Foo bar’, 3)         –>    ‘o bar’
```

```
SUBSTR('Foo bar', 3, 3)       ->    'o b'
SUBSTR('Foo bar', 4, 6)       ->    ' bar  '
SUBSTR('Foo bar', 4, 6, '*')  ->    ' bar**'
SUBSTR('Foo bar', 9, 4, '*')  ->    '****'
```

SUBWORD(*string*,*start*[,*length*])

Returns the part of *string* that starts at blank delimited word *start* (which must be a positive whole number). If *length* (which must be a non-negative whole number) is specified, that number of words are returned. The default value for *length* is the rest of the string.

It is not an error to specify *length* to refer to more words than *string* contains, or for *start* and *length* together to specify more words than *string* holds. The result string will be stripped of any leading and trailing blanks, but inter-word blanks will be preserved as is.

```
SUBWORD('To be or not  to be', 4)        ->    'not  to be'
SUBWORD('To be or not  to be', 4, 2)     ->    'not  to'
SUBWORD('To be or not  to be', 4, 5)     ->    'not  to be'
SUBWORD('To be or not  to be', 1, 3)     ->    'To be or'
```

SYMBOL(*name*)

Checks if the string *name* is a valid symbol (a positive number or a possible variable name), and returns a three letter string indicating the result of that check. If *name* is a symbol, and names a currently set variable, `VAR` is returned, if *name* is a legal symbol name, but has not a been given a value (or is a constant symbol, which can not be used as a variable name), `LIT` is returned to signify that it is a literal. Else, if *name* is not a legal symbol name the string `BAD` is returned.

Watch out for the effect of "double expansion". *Name* is interpreted as an expression evaluating naming the symbol to be checked, so you might have to quote the parameter.

```
SYMBOL('Foobar')        ->    'VAR'  /* Maybe */
SYMBOL('Foo bar')       ->    'BAD'
SYMBOL('Foo.Foo bar')   ->    'VAR' /* Maybe */
SYMBOL('3.14')          ->    'LIT'
SYMBOL('.Foo->bar')     ->    'BAD'
```

TIME([*option_out* [,*time* [*option_in*]]])

Returns a string containing information about the time. To get the time in a particular format, an *option_out* can be specified. The default *option_out* is `Normal`. The meaning of the possible options are:

[C]

(Civil) Returns the time in civil format. The return value might be "`hh:mmXX`", where `XX` are either `am` or `pm`. The `hh` part will be stripped of any leading zeros, and will be in the range 1 -12 inclusive.

[E]

(Elapsed) Returns the time elapsed in seconds since the internal stopwatch was started. The result will not have any leading zeros or blanks. The output will be a floating point number with six digits after the decimal point.

[H]

(Hours) Returns the number of complete hours that have passed since last midnight in the form "`hh`". The output will have no leading zeros, and will be in the range 0 -23.

[L]

(Long) Returns the exact time, down to the microsecond. This is called the long format. The output might be "`hh:mm:ss.mmmmmm`". Be aware that most computers do not have a clock of that accuracy, so the actual granularity you can expect, will be about a few milliseconds. The `hh`, `mm` and `ss` parts will be identical to

what is returned by the options H, M and S respectively, except that each part will have leading zeros as indicated by the format.

[M]

(Minutes) Returns the number of complete minutes since midnight, in a format having no leading zeros, and will be in the range 0 - 59.

[N]

(Normal) The output format is "hh:mm:ss", and is padded with zeros if needed. The hh, mm and ss will contain the hours, minutes and seconds, respectively. Each part will be padded with leading zeros to make it double-digit.

[R]

(Reset) Returns the value of the internal stopwatch just like the E option, and using the same format. In addition, it will reset the stopwatch to zero after its contents has been read.

[S]

(Seconds) Returns the number of complete seconds since midnight, in a format having no leading spaces, and will be in the range 0-59.

Note that the time is never rounded, only truncated. As shown in the examples below, the seconds do not get rounded upwards, even though the decimal part implies that they are closer to 59 than to 58. The same applies for the minutes, which are closer to 33 than to 32, but is truncated to 32.

None of the formats will have leading or trailing spaces.

Assuming that the time is exactly 14:32:58.987654, the following will be true:

```
TIME('C')   ->    '2:32pm'
TIME('E')   ->    '0.01200'  /* Maybe */
TIME('H')   ->    '14'
TIME('L')   ->    '14:32:58.987654'
TIME('M')   ->    '32'
TIME('N')   ->    '14:32:58'
TIME('R')   ->    '0.430221'  /* Maybe */
TIME('S')   ->    '58'
```

If the *time* option is specified, the function provides for time conversions. The optional *option_in* specifies the format in which *time* is supplied. The possible values for *option_in* are: **CHLMNS.**
The default value for *option_in* is **N**.

```
TIME('C', '11:27:21')         ->    '11:27am'
TIME('N', '11:27am', 'C')     ->    '11:27:00'
```

The time conversion capability of the TIME BIF was introduced with the ANSI standard.

TRACE([*setting*])

Returns the current value of the trace setting. If the string *setting* is specified, it will be used as the new setting for tracing, after the old value have be recorded for the return value. Note that the *setting* is not an option, but may be any of the trace settings that can be specified to the clause TRACE, except that the numeric variant is not allowed with TRACE(). In practice, this can be a word, of which only the first letter counts, optionally preceded by a question mark.

```
TRACE()     ->    'C'  /* Maybe */
TRACE('N')  ->    'C'
TRACE('?')  ->    'N'
```

TRANSLATE(*string*[,[*tablein*][,[*tableout*][,*padchar*]]])

Performs a translation on the characters in *string*. As a special case, if neither *tablein* nor *tableout* is specified, it will translate *string* from lower case to upper case. Note that this operation may depend on the language chosen, if your interpreter supports national character sets.

Two translation tables might be specified as the strings *tablein* and *tableout*. If one or both of the tables are specified, each character in *string* that exists in *tablein* is translated to the character in *tableout* that occupies the same position as the character did in *tablein*. The *tablein* defaults to the whole character set (all 256) in numeric sequence, while *tableout* defaults to an empty set. Characters not in *tablein* are left unchanged.

If *tableout* is larger than *tablein*, the extra entries are ignored. If it is smaller than *tablein* it is padded with *padchar* to the correct length. *Padchar* defaults to <space>.

If a character occurs more than once in *tablein*, only the first occurrence will matter.

```
TRANSLATE('FooBar')                              –>    'FOOBAR'
TRANSLATE('FooBar', 'ABFORabfor', 'abforABFOR') –>    'fOObAR'
TRANSLATE('FooBar', 'abfor')                     –>    'F  B  '
TRANSLATE('FooBar, 'abfor',, '#')                –>    'F##B##'
```

TRUNC(*number*[,*length*])

Returns *number* truncated to the number of decimals specified by *length*. *Length* defaults to 0, that is return an whole number with no decimal part.

The decimal point will only be present if the is a non-empty decimal part, i.e. *length* is non-zero. The number will always be returned in simple form, never exponential form, no matter what the current settings of *NUMERIC* might be. If *length* specifies more decimals than *number* has, extra zeros are appended. If *length* specifies less decimals than *number* has, the number is truncated. Note that *number* is never rounded, except for the rounding that might take place during normalization.

```
TRUNC(12.34)      –>    '12'
TRUNC(12.99)      –>    '12'
TRUNC(12.34, 4)   –>    '12.3400'
TRUNC(12.3456, 2) –>    '12.34'
```

VALUE(*symbol*[,[*value*],[*pool*]])

This function expects as first parameter string *symbol*, which names an existing variable. The result returned from the function is the value of that variable. If *symbol* does not name an existing variable, the default value is returned, and the NOVALUE condition is not raised. If *symbol* is not a valid symbol name, and this function is used to access an normal REXX variable, an error occurs. Be aware of the "double-expansion" effect, and quote the first parameter if necessary.

If the optional second parameter is specified, the variable will be set to that value, after the old value has been extracted.

The optional parameter *pool* might be specified to select a particular pool of variables to search for *symbol*. The contents and format of *pool* is implementation dependent. The default is to search in the variables at the current procedural level in REXX. Which *pool*s that are available is implementation dependent, but typically one can set variables in application programs or in the operating system.

Note that if VALUE() is used to access variable in pools outside the REXX interpreter, the requirements to format (a valid symbol) will not in general hold. There may be other requirements instead, depending on the implementation and the system. Depending on the validity of the name, the value, or whether the variable can be set or read, the VALUE() function can give error messages when accessing variables in pools other than the normal. Consult the implementation and system specific documentation for more information.

If it is used to access compound variables inside the interpreter the tail part of this function can take any expression, even expression that are not normally legal in REXX scripts source code.

By using this function, it is possible to perform an extra level of interpretation of a variable.

```
VALUE('FOO')                          –>    'bar'
VALUE('FOO', 'new')                   –>    'bar'
VALUE('FOO')                          –>    'new'
VALUE('USER', 'root', 'SYSTEM')       –>    'guest'  /* If SYSTEM exists */
VALUE('USER',, 'SYSTEM')              –>    'root'
```

VERIFY(*string*,*ref*[,[*option*][,*start*]])

With only the first two parameters, it will return the position of the first character in *string* that is not also a character in the string *ref*. If all characters in *string* are also in *ref*, it will return `0`.

If *option* is specified, it can be one of:

[N]

(Nomatch) The result will be the position of the first character in *string* that does exist in *ref*, or zero if all exist in *ref*. This is the default option.

[M]

(Match) Reverses the search, and returns the position of the first character in *string* that exists in *ref*. If none exists in *ref*, zero is returned.

If *start* (which must be a positive whole number) is specified, the search will start at that position in *string*. The default value for *start* is `1`.

```
VERIFY('foobar', 'barfo')          –>    '2'
VERIFY('foobar', 'barfo', 'M')     –>    '2'
VERIFY('foobar', 'fob', 'N')       –>    '5'
VERIFY('foobar', 'barf', 'N', 3)   –>    '3'
VERIFY('foobar', 'barf', 'N', 4)   –>    '0'
```

WORD(*string*,*wordno*)

Returns the blank delimited word number *wordno* from the string *string*. If *wordno* (which must be a positive whole number) refers to a non-existing word, then a nullstring is returned. The result will be stripped of any blanks.

```
WORD('To be or not to be', 3) –>    'or'
WORD('To be or not to be', 4) –>    'not'
WORD('To be or not to be', 8) –>    ''
```

WORDINDEX(*string*,*wordno*)

Returns the character position of the first character of blank delimited word number *wordno* in *string*, which is interpreted as a string of blank delimited words. If *number* (which must be a positive whole number) refers to a word that does not exist in *string*, then *0* is returned.

```
WORDINDEX('To be or not to be', 3)  –>    '7'
WORDINDEX('To be or not to be', 4)  –>    '10'
WORDINDEX('To be or not to be', 8)  –>    '0'
```

WORDLENGTH(*string*,*wordno*)

Returns the number of characters in blank delimited word number *number* in *string*. If *number* (which must be a positive whole number) refers to an non-existent word, then `0` is returned. Trailing or leading blanks do not count when calculating the length.

```
WORDLENGTH('To be or not to be', 3) ->    '2'
WORDLENGTH('To be or not to be', 4) ->    '3'
WORDLENGTH('To be or not to be', 0) ->    '0'
```

WORDPOS(*phrase*,*string*[,*start*])

Returns the word number in *string* which indicates at which *phrase* begins, provided that *phrase* is a subphrase of *string*. If not, `0` is returned to indicate that the phrase was not found. A phrase differs from a substring in one significant way; a phrase is a set of words, separated by any number of blanks.

For instance, "`is  a`" is a subphrase of "`This is a phrase`". Notice the different amount of whitespace between "`is`" and "`a`".

If *start* is specified, it sets the word in *string* at which the search starts. The default value for *start* is `1`.

```
WORDPOS('or not', 'to be or not to be')   ->    '3'
WORDPOS('not to', 'to be or not to be')   ->    '4'
WORDPOS('to be', 'to be or not to be')    ->    '1'
WORDPOS('to be', 'to be or not to be', 3) ->    '6'
```

WORDS(*string*)

Returns the number of blank delimited words in the *string*.

```
WORDS('To be or not to be')   ->    '6'
WORDS('Hello world')          ->    '2'
WORDS('')                     ->    '0'
```

XRANGE([*start*][,*end*])

Returns a string that consists of all the characters from *start* through *end*, inclusive. The default value for character *start* is `'00'x`, while the default value for character *end* is `'ff'x`. Without any parameters, the whole character set in "alphabetic" order is returned. Note that the actual representation of the output from `XRANGE()` depends on the character set used by your computer.

If the value of *start* is larger than the value of *end*, the output will wrap around from `'ff'x` to `'00'x`. If *start* or *end* is not a string containing exactly one character, an error is reported.

```
XRANGE('A', 'J')        ->    'ABCDEFGHIJ'
XRANGE('FC'x)           ->    'FCFDFEFF'x
XRANGE(, '05'x)         ->    '000102030405'x
XRANGE('FD'x, '04'x)    ->    'FDFEFF0001020304'x
```

X2B(*hexstring*)

Translate *hexstring* to a binary string. Each hexadecimal digits in *hexstring* will be translated to four binary digits in the result. There will be no blanks in the result.

X2C(*hexstring*)

Returns the (packed) string representation of *hexstring*. The *hexstring* will be converted bytewise, and blanks may optionally be inserted into the *hexstring* between pairs or hexadecimal digits, to divide the number into groups and improve readability. All groups must have an even number of hexadecimal digits, except the first group. If the first group has an odd number of hexadecimal digits, it is padded with an extra leading zero before conversion.

```
X2C('')                 ->    ''
X2C('466f6f 426172')    ->    'FooBar'
```

```
X2C('46  6f  6f')       ->    'Foo'
```

X2D(*hexstring*[,*length*])

Returns a whole number that is the decimal representation of *hexstring*. If *length* is specified, then *hexstring* is interpreted as a two's complement hexadecimal number consisting of the *number* rightmost hexadecimal numerals in *hexstring*. If *hexstring* is shorter than *number*, it is padded to the left with <NUL> characters (that is: '00'x).

If *length* is not specified, *hexstring* will always be interpreted as an unsigned number. Else, it is interpreted as an signed number, and the leftmost bit in *hexstring* decides the sign.

```
X2D('03 24')      ->    '792'
X2D('0310')       ->    '784'
X2D('ffff')       ->    '65535'
X2D('ffff', 5)    ->    '65535'
X2D('ffff', 4)    ->    '-1'
X2D('ff80', 3)    ->    '-128'
X2D('12345', 3)   ->    '837'
```

3. Implementation specific documentation for Regina

3.1 Deviations from the Standard

- For those built-in functions where the last parameter can be omitted, Regina allows the last comma to be specified, even when the last parameter itself has been omitted.
- The error messages are slightly redefined in two ways. Firstly, some of the have a slightly more definite text, and secondly, some new error messages have been defined.
- The environments available are described in chapter [not yet written].
- Parameter calling
- Stream I/O
- Conditions
- National character sets
- Blanks
- Stacks have the following extra functionality: DROPBUF(), DESBUF() and MAKEBUF() and BUFTYPE().
- Random()
- Sourceline
- Time
- Character sets

3.2 Interpreter Internal Debugging Functions

ALLOCATED([*option*])

Returns the amount of dynamic storage allocated, measured in bytes. This is the memory allocated by the `malloc()` call, and does not concern stack space or static variables.

As parameter it may take an *option*, which is one of the single characters:

[A]
: This is the default value if you do not specify an option. It will return a string that is the number of bytes of dynamic memory currently allocated by the interpreter.

[C]
: Returns a number that is the number of bytes of dynamic memory that is currently in use (i.e. not leaked).

[L]
: Returns the number of bytes of dynamic memory that is supposed to have been leaked.

[S]
: Returns a string that is nicely formatted and contains all the other three options, with labels. The format of this string is:

  "`Memory: Allocated=XXX, Current=YYY, Leaked=ZZZ`".

This function will only be available if the interpreter was compiled with the `TRACEMEM` preprocessor macro defined.

DUMPTREE()

Prints out the internal parse tree for the REXX program currently being executed. This output is not very interesting unless you have good knowledge of the interpreter's internal structures.

DUMPVARS()

This routine dumps a list of all the variables currently defined. It also gives a lot of information which is rather uninteresting for most users.

LISTLEAKED()

List out all memory that has leaked from the interpreter. As a return value, the total memory that has been listed is returned. There are several option to this function:

[N]
: Do not list anything, just calculate the memory.

[A]
: List all memory allocations currently in use, not only that which has been marked as leaked.

[L]
: Only list the memory that has been marked as leaked. This is the default option.

TRACEBACK()

Prints out a traceback. This is the same routine which is called when the interpreter encounters an error. Nice for debugging, but not really useful for any other purposes.

3.3 REXX UNIX Interface Functions

CHDIR(*string*)

Set *string* as current working directory.

A separate function is needed for this task in the current implementation. But when commands are implemented using pipes/sockets instead of the C function `system()`, this will not be needed. Then the REXX interpreter and its subprocesses have different current directories.

GETENV(*environmentvar*)

Returns the named UNIX environment variable. If this variable is not defined, a nullstring is returned. It is not possible to use this function to determine whether the variable was unset, or just set to the nullstring.

This function is now obsolete, instead you should use:

```
VALUE( environmentvar, ,'SYSTEM' )
```

UNIXERROR(*errorno*)

This function returns the string associated with the `errno` error number that *errorno* specifies. When some UNIX interface function returns an error, it really is a reference to an error message which can be obtained through `UNIXERROR`.

This function is just an interface to the `strerror()` function call in UNIX, and the actual error messages might differ with the operating system.

This function is now obsolete, instead you should use:

```
ERRORTEXT( 100 + errorno )
```

Conditions

In this chapter, the REXX concept of "conditions" is described. Conditions allow the programmer to handle abnormal control flow, and enable him to assign special pieces of REXX code to be executed in case of certain incidences.

- *In the first section the concept of conditions is explained.*
- *Then, there is a description of how a standard condition in REXX would work, if it existed.*
- *In the third section, all the existing conditions in REXX are presented, and the differences compared to the standard condition described in the previous section are listed.*
- *The fourth sections contains a collections of random notes on the conditions in REXX.*
- *The last section describes differences, extensions and peculiarities in Regina on the of subject conditions, and the lists specific behavior.*

1. What are Conditions

In this section, the concept of "conditions" are explained: What they are, how they work, and what they mean in programming.

1.1 What Do We Need Conditions for?

1.2 Terminology

First, let's look at the terminology used in this chapter. If you don't get a thorough understanding of these terms, you will probably not understand much of what is said in the rest of this chapter.

[Incident:]
A situation, external or internal to the interpreter, which it is required to respond to in certain pre-defined manners. The interpreter recognizes incidents of several different types. The incident will often have a character of "suddenness", and will also be independent of the normal control flow.

[Event:]
Data Structure describing one incident, used as a descriptor to the incident itself.

[Condition:]
Names the REXX concept that is equivalent to the incident.

[Raise a Condition:]
The action of transforming the information about an incident into an event. This is done after the interpreter senses the condition. Also includes deciding whether to ignore or produce an event.

[Handle a Condition:]
The act of executing some pre-defined actions as a response to the event generated when a condition was raised.

[(Condition) Trap:]
Data Structure containing information about how to handle a condition.

[(Trap) State:]

Part of the condition trap.

[(Condition) Handler:]
Part of the condition trap, which points to a piece of REXX code which is to be used to handle the condition.

[(Trap) Method:]
Part of the condition trap, which defined how the condition handler is to be invoked to handle the condition.

[Trigger a Trap:]
The action of invoking a condition handler by the method specified by the trap method, in order to handle a condition.

[Trap a Condition:]
Short of trigger a trap for a particular condition.

[Current Trapped Condition:]
The condition currently being handled. This is the same as the most recent trapped condition on this or higher procedure level.

[(Pending) Event Queue:]
Data Structure storing zero or more events in a specific order. There are only one event queue. The event queue contains events of all condition types, which have been raised, but not yet handled.

[Default-Action:]
The pre-defined default way of handling a condition, taken if the trap state for the condition raised is OFF.

[Delay-Action:]
The pre-defined default action taken when a condition is raised, and the trap state is DELAY.

2. The Mythical Standard Condition

REXX Language Level 4.00 has six different conditions. However, each of these is a special case of a mythical, non-existing, standard condition. In order to better understand the real conditions, we start by explaining how a standard condition work.

In the examples below, we will call our non-existing standard condition MYTH. Note that these examples will not be executable on any REXX implementation.

2.1 Information Regarding Conditions (data structures)

There are mainly five conceptual data structures involved in conditions.

[Event queue.]
There is one interpreter-wide queue of pending conditions. Raising a condition is identical to adding information about the condition to this queue (FIFO). The order of the queue is the same order in which the conditions are to be handled.

Every entry in the queue of pending conditions contains some information about the event: the line number of the REXX script when the condition was raised, a descriptive text and the condition type.

[Default-Action.]
To each, there exists information about the default-action to take if this condition is raised but the trap is in state OFF. This is called the “default-action”. The standard default-action is to ignore the condition, while some conditions may abort the execution.

[Delay-Action.]

Each condition will also have delay-action, which tells what to do if the condition is raised when condition trap is in state DELAY. The standard delay-action is to queue the condition in the queue of pending conditions, while some conditions may ignore it.

[Condition traps.]
For each condition there is a trap which contains three pieces of status information: the state; the handler; and the method. The state can be ON, OFF or DELAY.

The handler names the REXX label in the start of the REXX code to handle the event. The method can be either SIGNAL or CALL, and denotes the method in which the condition is to be handled. If the state is OFF, then neither handler nor method is defined.

[Current Trapped Condition.]
This is the most recently handled condition, and is set whenever a trap is triggered. It contains information about method, which condition, and a context-dependent description. In fact, the information in the current trapped condition is the same information that was originally put into the pending event queue.

Note that the event queue is a data structure connected to the interpreter itself. You operate on the same event queue, independent of subroutines, even external ones. On the other hand, the condition traps and the current trapped condition are data structures connected to each single routine. When a new routine is called, it will get its own condition traps and a current trapped condition. For internal routines, the initial values will be the same values as those of the caller. For external routines, the values are the defaults.

The initial value for the event queue is to be empty. The default-action and the delay-action are static information, and will always retain their values during execution. The initial values for the condition traps are that they are all in state OFF. The initial value for the current trapped condition is that all information is set to the nullstring to signalize that no condition is currently being trapped.

2.2 How to Set up a Condition Trap

How do you set the information in a condition trap? You do it with a SIGNAL or CALL clause, with the ON or OFF subkeyword. Remember that a condition trap contain three pieces of information? Here are the rules for how to set them:

- To set the trap method, use either SIGNAL or CALL as keyword.
- To set state to ON or OFF, use the appropriate subkeyword in the clause. Note that there is no clause or function in REXX, capable of setting the state of a trap to DELAY.
- To set the condition handler, append the term " NAME *handler*" to the command. Note that this term is only legal if you are setting the state to ON; you can not specify a handler when setting the state to OFF.

The trap is said to be "enabled" when the state is either ON or DELAY, and "disabled" when the state is OFF. Note that neither the event queue, nor the current trapped condition can be set explicitly by REXX clauses. They can only be set as a result of incidents, when raising and trapping conditions.

It sounds very theoretical, doesn't it? Look at the following examples, which sets the trap MYTH:

```
/* 1 */ SIGNAL ON MYTH NAME TRAP_IT
/* 2 */ SIGNAL OFF MYTH
/* 3 */ CALL ON MYTH NAME MYTH_TRAP
/* 4 */ CALL ON MYTH
/* 5 */ CALL OFF MYTH
```

Line 1 sets state to ON, method to SIGNAL and handler to TRAP_IT. Line 2 sets state to OFF, handler and method becomes undefined. Line 3 sets state to ON, method to CALL, and handler to MYTH_TRAP. Line 4 sets state to ON, method to CALL and handler to MYTH (the default). Line 5 sets state to OFF, handler and method become undefined.

Why should method and handler become undefined when the trap in state OFF? For two reasons: firstly, these values are not used when the trap is in state OFF; and secondly, when you set the trap to state ON, they are redefined. So it really does not matter what they are in state OFF.

What happens to this information when you call a subroutine? All information about traps are inherited by the subroutine, provided that it is an internal routine. External routines do not inherit any information about traps, but use the default values. Note that the inheritance is done by copying, so any changes done in the subroutine (internal or external), will only have effect until the routine returns.

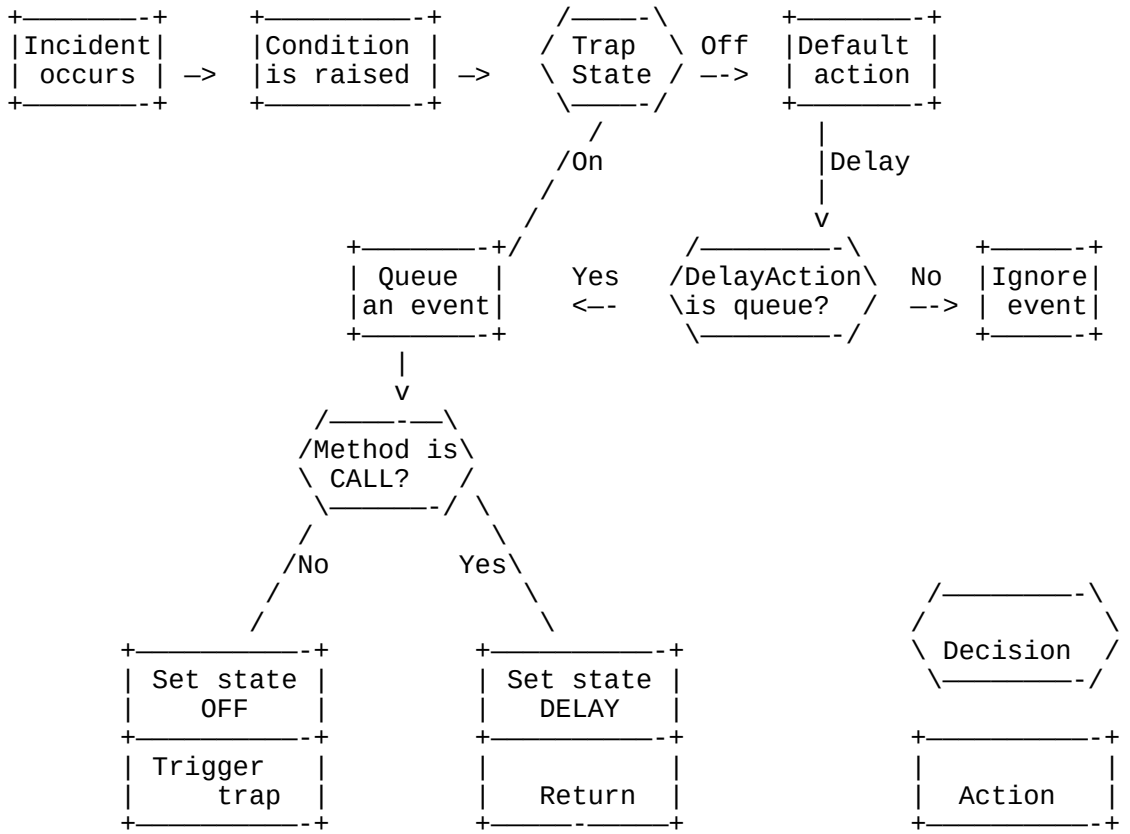
2.3 How to Raise a Condition

How do you raise a condition? Well, there are really no explicit way in REXX to do that. The conditions are raised when an incident occurs. What sort of situations that is, depends on the context. There are in general three types of incidents, classified by the origin of the event:

- Internal origin. The incident is only dependent on the behavior of the REXX script. The SYNTAX condition is of this type.
- External origin. The REXX script and the interpreter has really no control over when this incident. It happens completely independent of the control of the REXX script or interpreter. The HALT condition is of this type.
- Mixed origin. The incident is of external origin, but the situation that created the incident, was an action by the REXX script or the interpreter. The ERROR condition is of this type: the incident is a command returning error, but it can only occur when the interpreter is executing commands.

For conditions trapped by method CALL, standard REXX requires an implementation to at least check for incidents and raise condition at clause boundaries. (But it is allowed to do so elsewhere too; although the actual triggering must only be performed at clause boundaries.) Consequently, you must be prepared that in some implementations, conditions trappable by method CALL might only be raised (and the trap triggered) at clause boundaries, even if they are currently trapped by method SIGNAL.

The six standard conditions will be raised as result of various situations, read the section describing each one of them for more information.



When an incident occurs and the condition is raised, the interpreter will check the state of the condition trap for that particular condition at the current procedure level.

- If the trap state is `OFF`, the default-action of the condition is taken immediately. The "standard" default-action is to ignore the condition.

- If the trap state is `DELAY`, the action will depend on the delay-action of that condition. The standard delay-action is to ignore, then nothing further is done. If the delay-action is to queue, the interpreter continues as if the state was `ON`.

- If the state of the trap is `ON`, an event is generated which describes the incident, and it is queued in the pending event queue. The further action will depend on the method of trapping.

- If the method is `CALL`, the state of the trap will be set to `DELAY`. Then the normal execution is resumed. The idea is that the interpreter will check the event queue later (at a clause boundary), and trigger the appropriate trap, if it finds any events in the event queue.

- Else, if method of trapping is `SIGNAL`, then the action taken is this: First set the trap to state `OFF`, then terminate clause the interpreter was executing at this procedure level. Then it explicitly trigger the condition trap.

This process has be shown in the figure above. It shows how an incident makes the interpreter raise a condition, and that the state of the condition trap determines what to do next. The possible outcomes of this process are: to take the default-action; to ignore if delay-action is not to queue; to just queue and the continue execution; or to queue and trigger the trap.

2.4 How to Trigger a Condition Trap

What are the situations where a condition trap might be triggered? It depends on the method currently set in the condition trap.

If the method is `SIGNAL`, then the interpreter will explicitly trigger the relevant trap when it has raised the condition after having sensed the incident. Note that only the particular trap in question will be triggered in this case; other traps will not be triggered, even if the pending event queue is non-empty.

In addition, the interpreter will at each clause boundary check for any pending events in the event queue. If the queue is non-empty, the interpreter will not immediately execute the next normal statement, but it will handle the condition(s) first. This procedure is repeated until there are no more events queued. Only then will the interpreter advance to execute the next normal statement.

Note that the REXX standard does not require the pending events to be handled in any particular order, although the model shown in this documentation it will be in the order in which the conditions were raised. Consequently, if one clause generates several events that raise conditions before or at the next clause boundary, and these conditions are trapped by method `CALL`. Then, the order on which the various traps are triggered is implementations-dependent. But the order in which the different instances of the same condition is handled, is the same as the order of the condition indicator queue.

2.5 Trapping by Method `SIGNAL`

Assume that a condition is being trapped by method `SIGNAL`, that the state is `ON` and the handler is `MYTH_TRAP`. The following REXX clause will setup the trap correctly:

```
SIGNAL ON MYTH NAME MYTH_TRAP
```

Now, suppose the `MYTH` incident occurs. The interpreter will sense it, queue an event, set the trap state to `OFF` and then explicitly trigger the trap, since the method is `SIGNAL`. What happens when the trap is triggered?

- It collects the first event from the queue of pending events. The information is removed from the queue.
- The current trapped condition is set to the information removed from the pending event queue.
- Then, the interpreter simulates a SIGNAL clause to the label named by trap handler of the trap for the condition in question.
- As all SIGNAL clauses, this will have the side-effects of setting the SIGL special variable, and terminating all active loops at the current procedure level.

That's it for method SIGNAL. If you want to continue trapping condition MYTH, you have to execute a new SIGNAL ON MYTH clause to set the state of the trap to ON. But no matter how quick you reset the trap, you will always have a short period where it is in state OFF. This means that you can not in general use the method SIGNAL if you really want to be sure that you don't loose any MYTH events, unless you have some control over when MYTH condition may arise.

Also note that since the statement being executed is terminated; all active loops on the current procedure level are terminated; and the only indication where the error occurred is the line number (the line may contain several clauses), then it is in general impossible to pick up the normal execution after a condition trapped by SIGNAL. Therefore, this method is best suited for a "graceful death" type of traps. If the trap is triggered, you want to terminate what you were doing, and pick up the execution at an earlier stage, e.g. the previous procedure level.

2.6 Trapping by Method CALL

Assume that the condition MYTH is being trapped by method CALL, that the state is ON and the handler is MYTH_HANDLER.

The following REXX clause will setup the trap correctly:

```
CALL ON MYTH NAME MYTH_HANDLER
```

Now, suppose that the MYTH incident occurs. When the interpreter senses that, it will raise the MYTH condition. Since the trap state is ON and the trap method is CALL, it will create an event and queue it in the pending event queue and set the trap state to DELAY. Then it continues the normal execution. The trap is not triggered before the interpreter encounters the next clause boundary. What happens then?

- At the every clause boundaries, the interpreter check for any pending events in the event queue. If one is found, it is handled. This action is done repeatedly, until the event queue is empty.
- It will simulate a normal function call to the label named by the trap handler. As with any CALL clause, this will set the special variable SIGL to the line of from which the call was made. This is done prior to the call. Note that this is the current line at the time when the condition was raised, not when it was triggered. All other actions normally performed when calling a subroutine are done. Note that the arguments to the subroutine are set to empty.
- However, just before execution of the routine starts, it will remove the first event in the pending event queue, the information is instead put into the current trapped condition. Note that the current trapped condition is information that is saved across subroutine calls. It is set **after** the condition handler is called, and will be local to the condition handler (and functions called by the condition handler). To the "caller" (i.e. the procedure level active when the trap was triggered), it will seem as if the current trapped condition was never changed.
- Then the condition handler finishes execution, and returns by executing the RETURN clause. Any expression given as argument to RETURN will be ignored, i.e. the special variable RESULT will not be set upon return from a condition handler.
- At the return from the condition handler, the current trapped condition and the setup of all traps are restored, as with a normal return from subroutine. As a special case, the state of the trap just triggered, will not be put back into DELAY state, but is set to state ON.

- Afterwards (and before the next normal clause), the interpreter will again check for more events in the event queue, and it will not continue on the REXX script before the queue is empty.

During the triggering of a trap by method CALL at a clause boundary, the state of the trap is not normally changed, it will continue to be DELAY, as was set when the condition was raised. It will continue to be in state DELAY until return from the condition handler, at which the state of the trap in the caller will be changed to ON. If, during the execution of the condition trap, the state of the condition being trapped is set, that change will only last until the return from the condition handler.

Since new conditions are generally delayed when an condition handler is executing, new conditions are queued up for execution. If the trap state is changed to ON, the pending event queue will be processed as named at the next clause boundary. If the state is changed to OFF, the default action of the conditions will be taken at the next clause boundary.

2.7 The Current Trapped Condition

The interpreter maintains a data structure called the current trapped condition. It contains information relating the most recent condition trapped on this or higher procedure level. The current trapped condition is normally inherited by subroutines and functions, and restored after return from these.

- When trapped by method SIGNAL the current trapped condition of the current procedure level is set to information describing the condition trapped.

- When trapped by method CALL, the current trapped condition at the procedure level which the trap occurred at, is not changed. Instead, the current trapped condition in the condition handler is set to information describing the condition.

The information stored in the current trapped condition can be retrieved by the built-in function CONDITION(). The syntax format of this function is:

```
CONDITION(option)
```

where *option* is an option string of which only the first character matters. The valid options are: Condition name, Description, Instruction and State. These will return: the name of the current trapped condition; the descriptive text; the method; and the current state of the condition, respectively. The default *option* is Instruction. See the documentation on the built-in functions. See also the description of each condition below.

Note that the State option do not return the state at the time when the condition was raised or the trap was triggered. It returns the current state of the trap, and may change during execution. The other information in the current trapped condition may only change when a new condition is trapped at return from subroutines.

3. The Real Conditions

We have now described how the standard condition and condition trap works in REXX. Let's look at the six conditions defined which do exist. Note that none of these behaves exactly as the standard condition.

3.1 The SYNTAX condition

The SYNTAX condition is of internal origin, and is raised when any syntax or runtime error is discovered by the REXX interpreter. It might be any of the situations that would normally lead to the abortion of the program and the report of a REXX error message, except error message number 4 (*Program interrupted*), which is handled by the HALT condition.

There are several differences between this condition and the standard condition:

- It is not possible to trap this condition with the method CALL, only method SIGNAL. The reason for this is partly that method CALL tries to continue execution until next boundary before triggering the trap. That might not be possible with syntax or runtime errors.

- When this condition is trapped, the special variable RC is set to the REXX error number of the syntax or runtime error that caused the condition. This is done just before the setting of the special variable SIGL.

- The default action of this condition if the trap state is OFF, is to abort the program with a traceback and error message.

- There is not delay-action for condition SYNTAX, since it can not be trapped by method CALL, and consequently never can get into state DELAY.

The descriptive text returned by CONDITION() when called with the Description option for condition SYNTAX, is implementation dependent, and may also be a nullstring. Consult the implementation-specific documentation for more information.

3.2 The HALT condition

The HALT condition of external origin, which is raised as a result of an action from the user, normally a combination of keys which tries to abort the program. Which combination of keys will vary between operating systems. Some systems might also simulate this event by other means than key combinations. Consult system for more information.

The differences between HALT and the standard condition are:

- The default-action for the HALT condition is to abort execution, as though a REXX runtime error number 4 (*Program interrupted*) had been reported. But note that SYNTAX will never be raised if HALT is not trapped.

- The delay-action of this condition is to ignore, not queue.

The standard allows the interpreter to limit the search for situations that would set the HALT condition, to clause boundaries. As a result, the response time from pressing the key combination to actually raising the condition or triggering the trap may vary, even if HALT is trapped by method SIGNAL. If a clause for some reason has blocked execution, and never finish, you may not be able to break the program.

The descriptive text returned by CONDITION() when called with the Description option for condition HALT, is implementation dependent, and may also be a nullstring. In general, it will describe the way in which the interpreter was attempted halted, in particular if there are more than one way to do raise a HALT condition. Consult the implementation documentation for more information.

3.3 The ERROR condition

The ERROR is a condition of mixed origin, it is raised when a command returns a return value which indicates error during execution. Often, commands return a numeric value, and a particular value is considered to mean success. Then, other values might raise the ERROR condition.

Differences between ERROR and the standard condition:

- The delay action of ERROR is to ignore, not to queue.

- The special variable RC is always set before this condition is raised. So even if it is trapped by method SIGNAL, you can rely on RC to be set to the return value of the command.

Unfortunately, there is no universal standard on return values. As stated, they are often numeric, but some operating system use non-numeric return values. For those which do use numeric values, there are no standard telling which values and ranges are considered errors and which are considered success. In fact, the interpretation of the value might differ between commands within the same operating system.

Therefore, it is up to the REXX implementation to define which values and ranges that are considered errors. You must expect that this information can differ between implementations as well as between different environments within one implementation.

The descriptive text returned by CONDITION() when called with the Description option for condition ERROR, is the command which caused the error. Note that this is the command as the environment saw it, not as it was entered in the REXX script source code.

3.4 The FAILURE condition

The FAILURE is a condition of mixed origin, it is raised when a command returns a return value which indicates failure during execution, abnormal termination, or when it was impossible to execute a command. It is a subset of the ERROR condition, and if it is in state OFF, then the ERROR condition will be raised instead. But note that an implementation is free to consider all return codes from commands as ERRORs, and none as FAILURES. In that case, the only situation where a FAILURE would occur, is when it is impossible to execute a command.

Differences between FAILURE and the standard condition:

- The delay action of FAILURE is to ignore, not to queue.

- The special variable RC is always set before this condition is raised. So even if it is trapped by method SIGNAL, you can rely on RC to be set to the return value of the command, or the return code that signalize that the command was impossible to execute.

As for ERROR, there is no standard the defines which return values are failures and which are errors. Consult the system and implementation independent documentation for more information.

The descriptive text returned by CONDITION() when called with the Description option for condition FAILURE, is the command which caused the error. Note that this is the command as the environment saw it, not as it was entered in the REXX script source code.

3.5 The NOVALUE condition

The NOVALUE condition is of internal origin. It is raised in some circumstances if the value of an unset symbol (which is not a constant symbol) is requested. Normally, this would return the default value of the symbol. It is considered bad programming practice not to initialize variables, and setting the NOVALUE condition is one method of finding the parts of your program that uses this programming practice.

Note however, there are only three instances where this condition may be raised: that is when the value of an unset (non-constant) symbol is used requested: in an expression; after the VAR subkeyword in a PARSE clause; and as an indirect reference in either a template, a DROP or a PROCEDURE clause. In particular, this condition is not raised if the VALUE() or SYMBOL() built-in functions refer to an unset symbol.

Differences between NOVALUE and the standard condition are:

- It may only be trapped by method SIGNAL, never method CALL. This requirement might seem somewhat strange, but the idea is that since an implementation is only forced to check for conditions trapped by method CALL at clause boundaries, incidences that may occur at any point within clauses (like NOVALUE) can only be trapped by method SIGNAL. (However, condition NOTREADY can occur within a clause, and may be trapped by method CALL so this does not seem to be absolute consistent.)

- There is not delay-action for condition NOVALUE, since it can not be trapped by method CALL, and consequently never can get into state DELAY.

The descriptive text returned by calling CONDITION() with the Description option, is the derived (i.e. tail has be substituted if possible) name of the variable that caused the condition to be raised.

3.6 The NOTREADY condition

The condition NOTREADY is a condition of mixed origin. It is raised as a result of problems with stream I/O. Exactly what causes it, may vary between implementations, but some of the more probable causes are: waiting for more I/O on transient streams; access to streams not allowed; I/O operation would block if attempted; etc. See the chapter; **Stream Input and Output** for more information.

Differences between NOTREADY and the standard condition are:

- It will be ignored rather than queued if condition trap is in state DELAY.
- This condition differs from the rest in that it can be raised during execution of a clause, but can still be trapped by method CALL.

The descriptive text returned by CONDITION() when called with the Description option for condition NOTREADY, is the name of the stream which caused the problem. This is probably the same string that you used as the first parameter to the functions that operates on stream I/O. For the default streams (default input and output stream), the string returned by CONDITION() will be nullstrings.

Note that if the NOTREADY trap is in state DELAY, then all I/O for files which has tried to raise NOTREADY within the current clause will be simulated as if operation had succeeded.

4. Further Notes on Conditions

4.1 Conditions under Language Level 3.50

The concept of conditions was very much expanded from REXX language level 3.50 to level 4.00. Many of the central features in conditions are new in level 4.00, these include:

- The CALL method is new, previously only the SIGNAL method was available, which made it rather difficult to resume execution after a problem. As a part of this, the DELAY state has been added too.
- The condition NOTREADY has been added, to allow better control over problems involving stream I/O.
- The built-in function CONDITION() has been added, to allow extraction of information about the current trapped condition.

4.2 Pitfalls when Using Condition Traps

There are several pitfalls when using conditions:

- Remember that some information are saved across the functions. Both the current trapped condition and the settings of the traps. Consequently, you can not set a trap in a procedure level from a lower level. (I.e. calling a subroutine to set a trap is will not work.)
- Remember that SIGL is set when trapped by method CALL. This means that whenever a condition might be trapped by CALL, the SIGL will be set to a new value. Consequently, never trust the contents of the SIGL variable for more than one clause at a time. This is very frustrating, but at least it will not happen often. When it do happen, though, you will probably have a hard time debugging it.
- Also remember that if you use the PROCEDURE clause in a condition handler called by method CALL, remember to EXPOSE the special variables SIGL if you want to use it inside the condition handler. Else it will be shadowed by the PROCEDURE.

4.3 The Correctness of this Description

In this description of conditions in REXX, I have gone further in the description of how conditions work, their internal data structures, the order in which things are executed etc., than the standard does. I have tried to interpret the set of distinct statements that is the documentation on condition, and design a complete and consistent system describing how such conditions work. I have done this to try to clarify an area of REXX which at first glance is very difficult and sometimes non-intuitive.

I hope that the liberties I have taken have helped describe conditions in REXX. I do not feel that the adding of details that I have done in any way change how conditions work, but at least I owe the reader to list which concepts that are genuine REXX, and which have been filled in by me to make the picture more complete. These are not a part of the standard REXX.

- REXX does not have anything called a standard condition. There just "are" a set of conditions having different attributes and values. Sometimes there are default values to some of the attributes, but still the are no default condition.

- The terms "event" and "incident" are not used. Instead the term "condition" is somewhat overloaded to mean several things, depending on the situation. I have found it advantageous to use different terms for each of these concepts.

- Standard REXX does not have condition queue, although a structure of such a kind is needed to handled the queuing of pending conditions when the trap state is DELAY.

- The values default-action and delay-action are really non-existing in the Standard REXX documentation. I made them up to make the system more easy to explain.

- The two-step process of first raising the flag, and then (possibly at a later stage) triggering the trap, is not really a REXX concept. Originally, REXX seems to allow implementations to select certain places of the interpreter where events are sought for. All standard conditions that can be called by method CALL, can be implemented by checking only at clause boundaries.

- Consequently, a REXX implementation can choose to trigger the trap immediately after a condition are raised (since conditions are only raised immediately before the trap would trigger anyway). This is also the common way used in language level 3.50, when only method SIGNAL was implemented.

- Unfortunately, the introduction of the state DELAY forces the interpreter to keep a queue of pending conditions, so there is nothing to gain on insisting that raising should happen immediately before triggering. And the picture is even more muddied when the NOTREADY condition is introduced. Since it explicitly allows raising of condition to be done during the clause, even though the triggering of the trap must happen (if method is CALL) at the end of the clause.

I really hope that these changes has made the concept of conditions easier to understand, not harder. Please feel free to flame me for any of these which you don't think is representative for REXX.

5. Conditions in Regina

Here comes documentation that are specific for the Regina implementation of REXX.

5.1 How to Raise the HALT condition

The implementation connect the HALT condition to an external event, which might be the pressing of certain key combination. The common conventions of the operating system will dictate what that combination of keystrokes is.

Below is a list, which describes how to invoke an event that will raise the HALT condition under various the operating systems which Regina runs under.

- Under various variants of the **Unix** operating system, the HALT event it connected to the signal “interrupt” (SIGINT). Often this signal is bound to special keystrokes. Depending on your version of Unix, this might be <ctrl>-<c> (mostly BSD-variants) or the <del> key (mostly System V). It is also possible to send this signal from the command line, in general using the program kill(1); or from program, in general using the call signal(3). Refer to your Unix documentation for more information.

- Under **VAX/VMS**, the key sequence <ctrl>-<c> is used to raise the HALT condition in the interpreter.

5.2 Extended built-in functions

Regina has a few extra built-in functions that are added to support the debugging of the interpreter. Under some circumstances, these might also be useful when debugging REXX scripts. Note that these functions are not a part of standard REXX and should never be used when portability is required. The functions are:

RAISE_COND(*condition*)
: is used to explicitly raise a *condition* during execution of a REXX script. The interpreter will accept the execution of this function as an event, just as if the event had occurred. Returns the nullstring.

COND_INFO([*condition*])
: is a function that will return information about the current settings of the condition indicator for *condition*, including the state of the flag, and the contents of the pending queue. If called without a parameter, it will return a <space>-separated list of those conditions which have non-empty pending condition queue.

TRAP_INFO([*condition*])
: is a functions that returns the status information about a trigger at the current procedure level. The information returned will be the state, the method and the condition handler. If called without a parameter, a <space>-separated list of condition enabled (state ON or DELAY) at the current procedure level, is returned.

These functions are described in detail elsewhere. Note that these functions will only be available if the interpreter was compiled with the certain preprocessor flag set. If the code was included in the compilation, the availability of these function will still be dependent on the selection of extensions with the clause OPTIONS, where the extension DBG_FUNCS should be chosen. See chapter on extensions for more information.

5.3 Extra Condition in Regina

Regina has some other extra conditions. These conditions are:

- A condition DEBUG, that is very similar to the condition HALT. The condition is raised as a result of an event of external origin, generally a special combination of keystrokes is pressed.

- The default-action of this condition is to set the trace mode to Normal and interactive. Consequently, the user will generally get into interactive tracing at the next clause boundary. This way, the user may be able to stop the program during execution, and perform debugging.

- The delay-action of this condition is to ignore it.

- On Unix machines, this is the signal QUIT (SIGQUIT), which is normally bound the <ctrl>-\ key. Just like condition HALT, this might also be simulated from the command line, or from other programs. Consult the Unix documentation for more information. On VAX/VMS machines, this event is normally bound to the <ctrl>-<y> key.

- This extended condition will only be available if the extension COND_DEBUG has been chosen.

Whether or not the conditions listed here are available, may also depend on whether particular preprocessor flag was set during compilation. For more information, see the chapter on extensions.

5.4 Various Other Existing Extensions

Here is a list of other current extensions in Regina. See chapter on extensions for more information.

- Regina allows the condition NOVALUE to be trapped by method CALL, which is not allowed according to the standard.

- This extension will only be available if the extension CALL_ON_NOVALUE has been chosen, and the code was compiled with certain preprocessor flags set.

- If NOVALUE is being trapped by method CALL, the current clause will be completed as if NOVALUE was not trapped at all, returning the default value for an unset symbol as variable value.

6. Possible Future extensions

- Here is a list of possible future extensions to REXX which has not been implemented into Regina. Some of these exist in other implementations of REXX, and some of them are just suggestions or ideas thrown around by various people.

- Another extension could have been included, but have been left out so far. It is the delay-action, which in standard REXX can be either to ignore or to queue. There is at least one other action that make sense: to replace. That is, when a trap is in state DELAY, and a new condition has been raised, the pending queue is emptied, before the new condition is queued. That way, the new condition will effectively replace any conditions already in the queue.

- If there are several new conditions raised while the condition handler is executing (and the trap state is DELAY), only the very last of them is remembered.

- It should be possible to set the state for a trap to DELAY, so that any new instances of the condition is handles by the delay-action. As a special case, the SYNTAX condition trap might not be set in state DELAY

Stream Input and Output

And the streams thereof shall be turned into pitch
Isaiah 33:21

For every one that asketh receivedth;
and he that seeketh findth;
and to him that knocketh it shall be opened.
Matthew 7:8

This chapter treats the topic of input from and output to streams using the built-in functions. An overview of the other parts of the input/output (I/O) system is also given but not discussed in detail. At the end of the chapter there are sections containing implementation-specific information for this topic.

1. Background and Historical Remarks

Stream I/O is a problem area for languages like REXX. They try to maintain compatibility for all platforms (i.e. to be non-system-specific), but the basic I/O capabilities differ between systems, so the simplest way to achieve compatibility is to include only a minimal, common subset of the functionality of all platforms. With respect to the functionality of the interface to their surrounding environment, non-system-specific script languages like REXX are inherently inferior to system specific script languages which are hardwired to particular operating systems and can benefit from all their features.

Although REXX formally has its own I/O constructs, it is common for some platforms that most or all of the I/O is performed as operating system commands rather than in REXX. This is how it was originally done under VM/CMS, which was one of the earliest implementations and which did not support REXX's I/O constructs. There, the EXECIO program and the stack (among other methods) are used to transfer data to and from a REXX program.

Later, the built-in functions for stream I/O gained territory, but lots of implementations still rely on special purpose programs for doing I/O. The general recommendation to REXX programmers is to use the built-in functions instead of special purpose programs whenever possible; that is the only way to make compatible programs.

2. REXX's Notion of a Stream

REXX regards a stream as a sequence of characters, conceptually equivalent to what a user might type at the keyboard. Note that a stream is not generally equivalent to a file. [MCGH:DICT] defines a file as "a collection of related records treated as a unit," while [OX:CDICT] defines it as "Information held on backing store [...] in order (a) to enable it to persist beyond the time of execution of a single job and/or (b) to overcome space limitations in main memory." A stream is defined by [OX:CDICT] as "a flow of data characterized by relative long duration and constant rate."

Thus, a file has a flavor of persistency, while a stream has a flavor of sequence and momentarily. For a stream, data read earlier may already have been lost, and the data not yet read may not be currently defined; for instance the input typed at a keyboard or the output of a program. Even though much of the REXX literature use these two terms interchangeably (and after all, there is some overlap), you should bear in mind that there is a difference between them.

In this documentation, the term "file" means "a collection of persistent data on secondary storage, to which random access and multiple retrieval are allowed." The term "stream" means a sequential flow of data from a file or from a sequential device like a terminal, tape, or the output of a program. The term stream is also used in its strict REXX meaning: a handle to/from which a flow of data can be written/read.

3. Short Crash-Course

REXX I/O is very simple, and this short crash course is probably all you need in a first-time reading of this chapter. But note that that, we need to jump a bit ahead in this section.

To read a line from a stream, use the `LINEIN()` built-in function, which returns the data read. To write a stream, use the `LINEOUT()` built-in function, and supply the data to be written as the second parameter. For both operations, give the name of the stream as the first parameter. Some small examples:

```
contents = linein( 'myfile.txt' )
call lineout 'yourfile.txt', 'Data to be written'
```

The first of these reads a line from the stream `myfile.txt`, while the second writes a line to the stream `yourfile.txt`. Both these calls operate on lines and they use a system specific end-of-line marker as a delimiter between lines. The marker is tagged on at the end of any data written out, and stripped off any data read.

Opening a stream in REXX is generally done automatically, so you can generally ignore that in your programs. Another useful method is repositioning to a particular line:

```
call linein 'myfile.txt', 12, 0
call lineout 'yourfile.txt',, 13
```

Where the first of these sets the current read position to the start of line 12 of the stream; the second sets the current write position to the start of line 13. Note that the second parameter is empty, that means no data is to be written. Also note that the current read and write positions are two independent entities; setting one does not affect the other.

The built-in functions `CHARIN()` and `CHAROUT()` are similar to the ones just described, except that they are character-oriented, i.e. the end-of-line delimiter is not treated as a special character.

Examples of use are:

```
say charin( 'myfile.txt', 10 )
call charout 'logfile', 'some data'
```

Here, the first example reads 10 characters, starting at the current input position, while the second writes the eleven characters of "some data" to the file, without an end-of-file marker afterwards.

It is possible to reposition character-wise too, some examples are:

```
call charin 'myfile',, 8
call charout 'foofile,, 10
```

These two clauses repositions the current read and write positions of the named files to the 8^{th} and 10^{th} characters, respectively.

4. Naming Streams

Unlike most programming languages, REXX does not use file handles; the name of the stream is also in general the handle (although some implementations add an extra level of indirection). You must supply the name to all I/O functions operating on a stream. However, internally, the REXX interpreter is likely to use the native file pointers of the operating system, in order to improve speed. The name specified can generally be the name of an operating system file, a device name, or a special stream name supported by your implementation.

The format of the stream name is very dependent upon your operating system. For portability concerns, you should try not to specify it as a literal string in each I/O call, but set a variable to the stream name, and use that variable when calling I/O functions. This reduces the number of places you need to make changes if you need to port the program to another system. Unfortunately, this approach increases the need for `PROCEDURE EXPOSE`, since the variable containing the files name must be available to all routines using file I/O for that particular file, and all their non-common ancestors.

Example: Specifying file names

The following code illustrates a portability problem related to the naming of streams. The variable `filename` is set to the name of the stream operated on in the function call.

```
filename = '/tmp/MyFile.Txt'
say ' first line is' linein( filename )
say 'second line is' linein( filename )
say ' third line is' linein( filename )
```

Suppose this script, which looks like it is written for Unix, is moved to a VMS machine. Then, the stream name might be something like `SYS$TEMP:MYFILE.TXT`, but you only need to change the script at one particular point: the assignment to the variable `filename`; as opposed to three places if the stream name is hard-coded in each of the three calls to `LINEIN()`.

If the stream name is omitted from the built-in I/O functions, a default stream is used: input functions use the default input stream, while output functions use the default output stream. These are implicit references to the default input and output streams, but unfortunately, there is no standard way to explicitly refer to these two streams. And consequently, there is no standard way to refer to the default input or output stream in the built-in function `STREAM()`.

However, most implementations allow you to access the default streams explicitly through a name, maybe the nullstring or something like `stdin` and `stdout`. However, you must refer to the implementation-specific documentation for information about this.

Also note that standard REXX does not support the concept of a default error stream. On operating systems supporting this, it can probably be accessed through a special name; see system-specific information. The same applies for other special streams.

Sometimes the term "default input stream" is called "standard input stream," "default input devices," "standard input," or just "stdin."

The use of stream names instead of stream descriptors or handles is deeply rooted in the REXX philosophy: Data structures are text strings carrying information, rather than opaque data blocks in internal, binary format. This opens for some intriguing possibilities. Under some operating systems, a file can be referred to by many names. For instance, under Unix, a file can be referred to as `foobar`, `./foobar` and `./././foobar`. All which name the same file, although a REXX interpreter may be likely to interpret them as three different streams, because the names themselves differ. On the other hand, nothing prevents an interpreter from discovering that these are names for the same stream, and treat them as equivalent (except concerns for processing time). Under Unix, the problem is not just confined to the use of `./` in file names, hard-links and soft-links can produce similar effects, too.

Example: Internal file handles

Suppose you start reading from a stream, which is connected to a file called `foo`. You read the first line of `foo`, then you issue a command, in order to rename `foo` to `bar`. Then, you try to read the next line from `foo`. The REXX program for doing this under Unix looks something like:

```
signal on notready
line1 = linein( 'foo' )
'mv foo bar'
line2 = linein( 'foo' )
```

Theoretically, the file `foo` does not exist during the second call, so the second read should raise the `NOTREADY` condition. However, a REXX interpreter is likely to have opened the stream already, so it is performing the reading on the file descriptor of the open file. It is probably not going to check whether the file exists before each I/O operation (that would require a lot of extra checking). Under most operating systems, renaming a file will not invalidate existing file descriptors. Consequently, the interpreter is likely to continue to read from the original `foo` file, even though its has changed.

Example: Unix temporary files

On some systems, you can delete a file, and still read from and write to the stream connected to that file. This technique is shown in the following Unix specific code:

```
tmpfile = '/tmp/myfile'
call lineout tmpfile, ''
call lineout tmpfile,, 1
'rm' tmpfile
call lineout tmpfile, 'This is the first line'
```

Under Unix, this technique is often used to create temporary files; you are guaranteed that the file will be deleted on closing, no matter how your program terminates. Unix deletes a file whenever there are no more references to it. Whether the reference is from the file system or from an open descriptor in a user process is irrelevant. After the `rm` command, the only reference to the file is from the REXX interpreter. Whenever it terminates, the file is deleted—-since there are no more references to it.

Example: Files in different directories

Here is yet another example of how using the filename directly in the stream I/O functions may give strange effects. Suppose you are using a system that has hierarchical directories, and you have a function `CHDIR()` which sets a current directory; then consider the following code:

```
call chdir '../dir1'
call lineout 'foobar', 'written to foobar while in dir1'
call chdir '../dir2'
call lineout 'foobar', 'written to foobar while in dir2'
```

Since the file is implicitly opened while you are in the directory `dir1`, the file `foobar` refers to a file located there. However, after changing the directory to `dir2`, it may seem logical that the second call to `LINEOUT()` operates on a file in `dir2`, but that may not be the case. Considering that these clauses may come a great number of lines apart, that REXX has no standard way of closing files, and that REXX only have one file table (i.e. open files are not local to subroutines); this may open for a significant astonishment in complex REXX scripts.

Whether an implementation treats `././foo` and `./foo` as different streams is system-dependent; that applies to the effects of renaming or deleting the file while reading or writing, too. See your interpreter's system-specific documentation.

Most of the effects shown in the examples above are due to insufficient isolation between the filename of the operating system and the file handle in the REXX program. Whenever a file can be explicitly opened and bound to a file handle, you should do that in order to decrease the possibilities for strange side effects.

Interpreters that allow this method generally have an `OPEN()` function that takes the name of the files to open as a parameter, and returns a string that uniquely identifies that open file within the current context; e.g. an index into a table of open files. Later, this index can be used instead of the filename.

Some implementations allow only this indirect naming scheme, while others may allow a mix between direct and indirect naming. The latter is likely to create some problems, since some strings are likely to be both valid direct and indirect file ids.

5. Persistent and Transient Streams

REXX knows two different types of streams: persistent and transient. They differ conceptually in the way they can be operated, which is dictated by the way they are stored. But there is no difference in the data you can read from or write to them (i.e. both can used for character- or line-wise data), and both are read and written using the same functions.

[Persistent streams]
(often referred to just as "files") are conceptually stored on permanent storage in the computer (e.g. a disk), as an array of characters. Random access to and repeated retrieval of any part of the stream are allowed for persistent streams. Typical example of persistent streams are normal operating system files.

[Transient streams]
are typically not available for random access or repeated retrieval, either because it is not stored permanently, but read as a sequence of data that is generated on the fly; or because they are available from a sequential storage (e.g. magnetic tape) where random access is difficult or impossible. Typical examples of transient streams are devices like keyboards, printers, communication interfaces, pipelines, etc.

REXX does not allow any repositioning on transient streams; such operations are not conceptually meaningful; a transient stream must be treated sequentially. It is possible to treat a persistent stream as a transient stream, but not vice versa. Thus, some implementations may allow you to open a persistent stream as transient. This may be useful for files to which you have only append access, i.e. writes can only be performed at the end of file. Whether you can open a stream in a particular mode, or change the mode of a stream already open depends on your implementation.

Example: Determining stream type

Unfortunately, there is no standard way to determine whether a given file is persistent or transient. You may try to reposition for the file, and you can assume that the file is persistent if the repositioning succeeded, like in the following code:

```
streamtype: procedure
      signal on notready
      call linein arg(1), 1, 0
      return 'persistent'          /* unless file is empty */
notready:
      return 'transient'
```

Although the idea in this code is correct, there are unfortunately a few problems. First, the NOTREADY condition can be raised by other things than trying to reposition a transient stream; e.g. by any repositioning of the current read position in an empty file, if you have write access only, etc. Second, your implementation may not have NOTREADY, or it may not use it for this situation.

The best method is to use a STREAM() function, if one is available. Unfortunately, that is not very compatible, since no standard stream commands are defined.

6. Opening a Stream

In most programming languages, opening a file is the process of binding a file (given by a file name) to an internal handle. REXX is a bit special, since conceptually, it does not use stream handles, just stream names. Therefore, the stream name is itself also the stream handle, and the process of opening streams becomes apparently redundant. However, note that a number of implementations allow explicit opening, and some even require it.

REXX may open streams "on demand" when they are used for the first time. However, this behavior is not defined in TRL, which says the act of opening the stream is not a part of REXX [TRL2]. This might be interpreted as open-on-demand or that some system-specific program must be executed to open a stream.

Although an open-on-demand feature is very practical, there are situations where you need to open streams in particular modes. Thus, most systems have facilities for explicitly opening a file. Some REXX interpreters may require you to perform some implementation-specific operation before accessing streams, but most are likely to just open them the first time they are referred to in an I/O operation.

There are two main approaches to explicit opening of streams. The first uses a non-standard built-in function normally called `OPEN()`, which generally takes the name of the file to open as the first parameter, and often the mode as the second parameter. The second approach is similar, but uses the standard built-in function `STREAM()` with a `Command` option.

Example: Not closing files

Since there are no open or close operation, a REXX interpreter never knows when to close a stream, unless explicitly told so. It can never predict when a particular stream is to be used next, so it has to keep the current read and write positions in case the stream is to be used again. Therefore, you should always close the streams when you are finished using them. Failure to do so, will fill the interpreter with data about unneeded streams, and more serious, it may fill the file table of your process or system. As a rule, any REXX script that uses more than a couple of streams, should close every stream after use, in order to minimize the number of simultaneously open streams. Thus, the following code might eventually crash for some REXX interpreters:

```
do i=1 to 300
      call lineout 'file.'||i, 'this is file number' i
end
```

A REXX interpreter might try to defend itself against this sort of open-many-close-none programming, using of various programming techniques; this may lead to other strange effects. However, the main responsibility for avoiding this is with you, the REXX script programmer.

Note that if a stream is already open for reading, and you start writing to it, your implementation may have to reopen it in order to open for both reading and writing. There are mainly two strategies for handling this. Either the old file is closed, and then reopened in the new mode, which may leave you with read and write access to another file. Or a new file handle is opened for the new mode, which may leave you with read and write access to two different files.

These are real-world problems which are not treated by the ideal description of TRL. A good implementation should detect these situations and raise `NOTREADY`.

7. Closing a Stream

As already mentioned, REXX does not have an explicit way of opening a stream. Nor does it have an explicit way of closing a stream. There is one semi-standard method: If you call `LINEOUT()`, but omit both the data to be written and the new current write position, then the implementation is defined to set the current write position to the end-of-file. Furthermore, it is allowed by TRL to do something "magic" in addition. It is not explicitly defined what this magic is, but TRL suggests that it may be closing the stream, flushing the stream, or committing changes done previously to the stream.

In SAA, the definition is strengthened to state that the "magic" is closing, provided that the environment supports that operation.

A similar operating can be performed by calling `CHAROUT()` with neither data nor a new position. However, in this case, both TRL and SAA leave it totally up to the implementation whether or not the file is to be closed. One can wonder whether the changes for `LINEOUT()` in SAA with respect to TRL should also have been done to `CHAROUT()`, but that this was forgotten.

TRL2 does not indicate that `LINEIN()` or `CHARIN()` can be used to close a string. Thus, the closest one gets to a standard way of closing input files is to call e.g. `LINEOUT()`; although it is conceptually suspect to call an output routine for an input file. The historical reasons for this omission are perhaps that flushing output files is vital , while

the concept of flushing is irrelevant for input files; flushing is an important part of closing a file, and that explains why closing is only indicated for output files.

Thus, the statement:

```
call lineout 'myfile.txt'
```

might be used to close the stream `myfile.txt` in some implementations. However, it is not guaranteed to close the stream, so you cannot depend on this for scripts of maximum portability, but it's better than nothing. However, note that if it closes the stream, then also the current read position is affected. If it merely flushes the stream, then only the current write position is likely to be affected.

8. Character-wise and Line-wise I/O

Basically, the built-in REXX library offers two strategies of reading and writing streams: line-wise and character-wise. When reading line-wise, the underlying storage method of the stream must contain information which describes where each line starts and ends.

Some file systems store this information as one or more special characters; while others structure the file in a number of records; each containing a single line. This introduces a slightly subtle point; even though a stream `foo` returns the same data when read by `LINEIN()`on two different machines; the data read from `foo` may differ between the same two machines when the stream is read by `CHARIN()`, and vice versa. This is so because the end-of-line markers can vary between the two operating systems.

Example: Character-wise handling of EOL

Suppose a text file contains the following three lines (ASCII character set is assumed):

```
first
second
third
```

and you first read it line-wise and then character-wise. Assume the following program:

```
file = 'DATAFILE'
foo = ''
do i=1 while chars(file)>0
      foo = foo || c2x(charin(file))' '
end
say foo
```

When the file is read line-wise, the output is identical on all machines, i.e. the three lines shown above. However, the character-wise reading will be dependent on your operating system and its file system, thus, the output might e.g. be any of:

```
66 69 72 73 74 73 65 6F 63 6E 64 74 68 69 72 64 66 69 72 73 74 0A

66 69 72 73 74 0A
73 65 6F 63 6E 64 0A
74 68 69 72 64 0A

66 69 72 73 74 0D 0A
73 65 6F 63 6E 64 0D 0A
74 68 69 72 64 0D 0A
```

If the machine uses records to store the lines, the first one may be the result; here, only the data in the lines of the file is returned. Note that the boxes in the output are put around the data generated by the actual line contents. What is outside the boxes is generated by the end-of-line character sequences.

The second output line is typical for Unix machines. They use the newline ASCII character as line separator, and that character is read immediately after each line. The last line is typical for MS-DOS, where the line separator character sequence is a carriage return following by a newline (ASCII `'0D'x` and `'0A'x`).

For maximum portability, the line-wise built-in functions (`LINEIN()`, `LINEOUT()` and `LINES()`) should only be used for line-wise streams. And the character-wise built-in functions (`CHARIN()`, `CHAROUT()` and `CHARS()`) should only be used for character-wise data. You should in general be very careful when mixing character- and line-wise data in a single stream; it does work, but may easily lead to portability problems.

The difference between character- and line-wise streams are roughly equivalent to the difference between binary and text streams, but the two concepts are not totally equivalent. In a binary file, the data read is the actual data stored in the file, while in a text file, the character sequences used for denoting end-of-line and end-of-file markers may be translated to actions or other characters during reading.

The end-of-file marker may be differently implemented on different systems. On some systems, this marker is only implicitly present at the end-of-file—-which is calculated from the file size (e.g. Unix). Other systems may put a character signifying end-of-file at the end (or even in the middle) of the file (e.g. <Ctrl-Z> for MS-DOS). These concepts vary between operating systems, interpreters should handle each concept according to the customs of the operating system. Check the implementation-specific documentation for further information. In any case, if the interpreter treats a particular character as end-of-file, then it only gives special treatment to this character during line-wise operations. During character-wise operations, no characters have special meanings.

9. Reading and Writing

Four built-in functions provide line- and character-oriented stream reading and writing capabilities: `CHARIN()`, `CHAROUT()`, `LINEIN()`, `LINEOUT()`.

`[CHARIN()]`
: is a built-in function that takes up to three parameters, which are all optional: the name of the stream to read from, the start point, and the number of characters to read. The stream name defaults to the default input stream, the start point defaults to the current read position, the number of characters to read defaults to one character. Leave out the second parameter in order to avoid all repositioning. During execution, data is read from the stream specified, and returned as the return value.

`[LINEIN()]`
: is a built-in function that takes three parameters too, and they are equivalent to the parameters of `CHARIN()`. However, if the second parameter is specified, it refer to a line position, rather than a character position; it refers to the character position of the first character of that line. Further, the third parameter can only be `0` or `1`, and refers to the number of lines to read; i.e. you cannot read more than one line in each call. The line read is returned by the function, or the nullstring if no reading was requested.

`[LINEOUT()]`
: is a built-in function that takes three parameters too, the first is the name of the stream to write to, and defaults to the default output stream. The second parameter is the data to be written to the file, and if not specified, no writing occurs. The third parameter is a line-oriented position in the file; if the third parameter is specified, the current position is repositioned at before the data (if any) is written. If data is written, an end-of-line character sequence is appended to the output stream.

`[CHAROUT()]`
: is a built-in function that is used to write characters to a file. It is identical to `LINEOUT()`, except that the third parameter refers to a character position, instead of a line position. The second difference is that an end-of-line character sequence is not appended at the end of the data written.

Example: Counting lines, words, and characters

The following REXX program emulates the core functionality of the `wc` program under Unix. It counts the number of lines, words, and characters in a file given as the first argument.

```
file = arg(1)
parse value 0 0 0 with lines words chars
do while lines(file)>0
      line = linein(file)
      lines = lines + 1
      words = words + words(line)
      chars = chars + length(line)
end
say 'lines='lines 'words='words 'chars='chars
```

There are some problems. For instance, the end-of-line characters are not counted, and a last improperly terminated line is not counted either.

10. Determining the Current Position

Standard REXX does not have any seek call that returns the current position in a stream. Instead, it provides two calls that returns the amount of data remaining on a stream. These two built-in functions are `LINES()` and `CHARS()`.

- The `LINES()` built-in function returns the number of complete lines left on the stream given as its first parameter. The term "complete lines" does not really matter much, since an implementation can assume the end-of-file to implicitly mean an end-of-line.

- The `CHARS()` built-in function returns the number of character left in the stream given as its first parameter.

This is one of the concepts where REXX I/O does not map very well to C I/O and vice versa. While REXX reports the amount of data from the current read position to the end of stream, C reports the amount of data from the start of the file to the current position. Further, the REXX method only works for input streams, while the C method works for both input and output files. On the other hand, C has no basic constructs for counting remaining or reposition at lines of a file.

Example: Retrieving current position

So, how does one find the current position in a file, when only allowed to do normal repositioning? The trick is to reposition twice, as shown in the code below.

```
ftell: procedure
      parse arg filename
      now = chars(filename)
      call charin filename, 0, 1
      total = chars(filename)
      call charin filename, 0, total-now
      return total-now
```

Unfortunately, there are many potential problems with this code. First, it only works for input files, since there is no equivalent to `CHARS()` for output files. Second, if the file is empty, none of the repositioning work, since it is illegal to reposition at or after end-of-file for input files—-and the end-of-file is the first position of the file. Third, if the current read position of the file is at the end of file (e.g. all characters have been read) it will not work for similar reasons as for the second case. And fourth, it only works for persistent files, since transient files do not allow repositioning.

Example: Improved `ftell` function

An improved version of the code for the `ftell` routine (given above), which tries to handle these problems is:

```
ftell: procedure
      parse arg filename
      signal on notready name not_persist
      now = chars(filename)
      signal on notready name is_empty
      call charin filename, 0, 1
      total = chars()
      if now>0 then
            call charin filename, 0, total-now+1
      else if total>0 then
            call charin filename, 1, total
      else
            nop /* empty file, should have raised NOTREADY */
      return total-now+1

not_presist: say filename 'is not persistent'; return 0

is_empty: say filename 'is empty'; return 0
```

The same method can be used for line-oriented I/O too, in order to return the current line number of an input file. However, a potential problem in that case is that the routine leaves the stream repositioned at the start of the current line, even if it was initially positioned to the middle of a line. In addition, the line-oriented version of this `ftell` routine may prove to be fairly inefficient, since the interpreter may have to scan the whole file twice for end-of-line character sequences.

11. Positioning Within a File

REXX supports two strategies for reading and writing streams: character-wise, and line-wise, this section describes how a program can reposition the current positions for each these strategies. Note that positioning is only allowed for persistent streams.

For each open file, there is a current read position or a current write position, depending on whether the file is opened for reading or writing. If the file is opened for reading and writing simultaneously, it has both a current read position and a current write position, and the two are independent and in general different. A position within a file is the sequence number of the byte or line that will be read or written in the next such operation.

Note that REXX starts numbering at one, not zero. Therefore, the first character and the first line of a stream are both numbered one. This differs from several other programming languages, which starts numbering at zero.

Just after a stream has been opened, the initial values of the current read position is the first character in the stream, while the current write position is the end-of-file, i.e. the position just after the last character in the stream. Then, reading will return the first character (or line) in the stream, and writing will append a new character (or line) to the stream.

These initial values for the current read and write positions are the default values. Depending on your REXX implementation, other mechanisms for explicitly opening streams (e.g. through the `STREAM()` built-in function) may be provided, and may set other initial values for these positions. See the implementation-specific documentation for further information.

When setting the current read position, it must be set to the position of an existing character in the stream; i.e. a positive value, not greater than the total number of characters in the stream. In particular, it is illegal to set the current read position to the position immediately after the last character in the stream; although this is legal in many other programming languages and operating systems, where it is known as "seeking to the end-of-file".

When setting the current write position, it too must be set to the position of an existing character in the stream. In addition, and unlike the current read position, the current write position may also be set to the position immediately following the last character in the stream. This is known as "positioning at the end-of-file", and it is the initial value for the current write position when a stream is opened. Note that you are not allowed to reposition the current write

position further out beyond the end-of-file—-which would create a “hole” in the stream—-even though this is allowed in many other languages and operating systems.

Depending on your operating system and REXX interpreter, repositioning to after the end-of-file may be allowed as an extension, although it is illegal according to TRL2. You should avoid this technique if you wish to write portable programs.

REXX only keeps one current read position and one current write position for each stream. So both line-wise and character-wise reading as well as positioning of the current read position will operate on the same current read position, and similarly for the current write position.

When repositioning line-wise, the current write position is set to the first character of the line positioned at. However, if positioning character-wise so that the current read position is in the middle of a line in the file, a subsequent call to `LINEIN()` will read from (and including) the current position until the next end-of-line marker. Thus, `LINEIN()` might under some circumstances return only the last part of a line. Similarly, if the current write position has been positioned in the middle of an existing line by character-wise positioning, and `LINEOUT()` is called, then the line written out becomes the last part of the line stored in the stream.

Note that if you want to reposition the current write position using a line count, the stream may have to be open for read, too. This is because the interpreter may have to read the contents of the stream in order to find where the lines start and end. Depending on your operating system, this may even apply if you reposition using character count.

Example: Repositioning in empty files

Since the current read position must be at an existing character in the stream, it is impossible to reposition in or read from an empty stream. Consider the following code:

```
filename = ‘/tmp/testing’
call lineout filename,, 1   /* assuming truncation */
call linein filename, 1, 0
```

One might believe that this would set the current read and write positions to the start of the stream. However, assume that the `LINEOUT()` call truncates the file, so that it is zero bytes long. Then, the last call can never be legal, since there is no byte in the file at which it is possible to position the current read position. Therefore, a `NOTREADY` condition is probably raised.

Example: Relative repositioning

It is rather difficult to reposition a current read or write position relative to the current position. The only way to do this within the definition of the standard is to keep a counter which tells you the current position. That is, if you want to move the current read position five lines backwards, you must do it like this:

```
filename = ‘/tmp/data’
linenum = 0 ;
say linein(filename,10); linenum = 10
do while random(100)>3
      say linein(filename); linenum = linenum+1
end
call linein(filename,linenum-5,0); linenum = linenum-5
```

Here, the variable `linenum` is updated for each time the current read position is altered. This may not seem to difficult, and it is not in most cases. However, it is nearly impossible to do this in the general case, since you must keep an account of both line numbers and character numbers. Setting one may invalidate the other: consider the situation where you want to reposition the current read position to the 10^{th} character before the 100^{th} line in the stream. Except from mixing line-wise and character-wise I/O (which can have strange effects), this is nearly impossible. When repositioning character-wise, the line number count is invalidated, and vice versa.

The “only” proper way of handling this is to allow one or more (non-standard) `STREAM()` built-in function operations that returns the current character and line count of the stream in the interpreter.

Example: Destroying linecount

This example shows how overwriting text to the middle of a file can destroy the line count. In the following code, we assume that the file `foobar` exists, and contains ten lines which are "`first line`", `second line`, etc. up to "`tenth line`". Then consider the following code:

```
filename = 'foobar'
say linein(filename, 5)   /* says 'fifth line' */
say linein(filename)      /* says 'sixth line' */
say linein(filename)      /* says 'seventh line' */
call lineout filename, 'This is a very long line', 5
say linein(filename, 5)   /* says 'This is a very long line' */
say linein(filename)      /* says 'venth line' */
say linein(filename)      /* says 'eight line' */
```

As you can see from the output of this example, the call to `LINEOUT()` inserts a long line and overwrites the fifth and sixth lines completely, and the seventh line partially. Afterwards, the sixth line is the remaining part of the old seventh line, and the new seventh line is the old eighth line, etc.

12. Errors: Discovery, Handling, and Recovery

TRL2 contains two important improvements over TRL1 in the area of handling errors in stream I/O: the `NOTREADY` condition and the `STREAM()` built-in function. The `NOTREADY` condition is raised whenever a stream I/O operation did not succeed. The `STREAM()` function is used to retrieve status information about a particular stream or to execute a particular operation for a stream.

You can discover that an error occurred during an I/O operation in one of the following ways: a) it may trigger a `SYNTAX` condition; b) it may trigger a `NOTREADY` condition; or c) it may just not return that data it was supposed to. There is no clear border between which situations should trigger `SYNTAX` and which should trigger `NOTREADY`. Errors in parameters to the I/O functions, like a negative start position, is clearly a `SYNTAX` condition, while reading off the end-of-file is equally clearly a `NOTREADY` condition. In between lay more uncertain situations like trying to position the current write position after the end-of-file, or trying to read a non-existent file, or using an illegal file name.

Some situations are likely to be differently handled in various implementations, but you can assume that they are handled as either `SYNTAX` or `NOTREADY`. Defensive, portable programming requires you to check for both. Unfortunately, `NOTREADY` is not allowed in TRL1, so you have to avoid that condition if you want maximum compatibility. And due to the very lax restrictions on implementations, you should always perform very strict verification on all data returned from any file I/O built-in function.

If neither are trapped, `SYNTAX` will terminate the program while `NOTREADY` will be ignored, so the implementor's decision about which of these to use may even depend on the severity of the problem (i.e. if the problem is small, raising `SYNTAX` may be a little too strict). Personally, I think `SYNTAX` should be raised in this context only if the value of a parameter is outside its valid range for all contexts in which the function might be called.

Example: General `NOTREADY` condition handler

Under TRL2 the "correct" way to handle `NOTREADY` conditions and errors from I/O operations is unfortunately very complex. It is shown in this example, in order to demonstrate the procedure:

```
myfile = 'MYFILE.DAT'
signal on syntax name syn_handler
call on notready name IO_handler
do i=1 to 10 until res=0
      res = lineout(myfile, 'line #'i)
```

```
      if (res=0) then
            say 'Call to LINEOUT() didn"t manage to write out data'
end
exit

IO_handler:
syn_handler:
      file = condition('D')
      say condition('C') 'raised for file' file 'at line' sigl':'
      say '  ' sourceline(sigl)
      say '   State='stream(file,'S') 'reason:' stream(file,'D')
      call lineout( condition( 'D' ))   /* try to close */
      if condition('C')=='SYNTAX' then
            exit 1
      else
            return
```

Note the double checking in this example: first the condition handler is set up to trap any `NOTREADY` conditions, and then the return code from `LINEOUT()` is checked for each call.

As you can see, there is not really that much information that you can retrieve about what went wrong. Some systems may have additional sources from which you can get information, e.g. special commands for the `STREAM()` built-in function, but these are non-standard and should be avoided when writing compatible programs.

13. Common Differences and Problems with Stream I/O

This section describes some of the common traps and pitfalls of REXX I/O.

13.1 Where Implementations are Allowed to Differ

TRL is rather relaxed in its specifications of what an interpreter must implement of the I/O system. It recognizes that operating systems differ, and that some details must be left to the implementor to decide, if REXX is to be effectively implemented. The parts of the I/O subsystem of REXX where implementations are allowed to differ, are:

- The functions `LINES()` and `CHARS()` are not required to return the number of lines or characters left in a stream. TRL says that if it is impossible or difficult to calculate the numbers, these functions may return `1` unless it is absolutely certain that there are no more data left. This leads to some rather kludgy programming techniques.

- Implementations are allowed to ignore closing streams, since TRL does not specify a way to do this. Often, the closing of streams is implemented as a command, which only makes it more incompatible.

- Check the implementation-specific documentation before using the function `LINEOUT(file)` for closing files.

- The difference in the action of closing and flushing a file, can make a REXX script that works under one implementation crash under another, so this feature is of very limited value if you are trying to write portable programs.

TRL says that because the operating system environments will differ a lot, and an efficient and useful interpreter is the most important goal, implementations are allowed to deviate from the standard in any respect necessary in the domain of I/O [TRL2]. Thus, you should never assume anything about the I/O system, as the "rules" listed in TRL are only advisory.

13.2 Where Implementations might Differ anyway

In the section above, some areas where the standard allows implementations to differ are listed. In an ideal world, that ought to be the only traps that you should need to look out for, but unfortunately, the world is not ideal. There are

several areas where the requirements set up by the standard is quite high, and where implementations are likely to differ from the standard.

These areas are:

- Repositioning at (for the current write position) or beyond the end-of-file may be allowed. On some systems, to prohibit that would require a lot of checking, so some systems will probably skip that check. At least for some operating systems, the act of repositioning after end-of-file is a useful feature.

- Under Unix, it can be used for creating a dynamically sized random access file; do not bother about how much space is allocated for the file, just position to the correct "sloth" and write the data there. If the data file is sparse, holes might occur in the file; that is parts of the file which has not been written, and which is all zeros (and which are therefore not stored on disk.

- Some implementations will use the same position for both the current read position and the current write position to overcome these implementations. Whenever you are doing a read, and the previous operation was a write (or vice versa), it is may prove useful to reposition the current read (or write) position.

- There might be a maximum linesize for your REXX interpreter. At least the 50Kb limit on string length may apply.

- Handling the situation where another program writes data to a file which is used by the REXX interpreter for reading.

13.3 `LINES()` and `CHARS()` are Inaccurate

Because of the large differences between various operating systems, REXX allows some fuzz in the implementation of the `LINES()` and `CHARS()` built-in functions. Sometimes, it is difficult to calculate the number of lines or characters in a stream; generally because the storage format of the file often requires a linear search through the whole stream to determine that number. Thus, REXX allows an implementation to return the value `1` for any situation where the real number is difficult or impossible to determine. Effectively, an implementation can restrict the domain of return values for these two functions only `1` and `0` from these two functions.

Many operating systems store lines using a special end-of-line character sequence. For these systems, it is very time-consuming to count the number of lines in a file, as the file must be scanned for such character sequences. Thus, it is very tempting for an implementor to return the value `1` for any situation where there are more than zero lines left.

A similar situation arises for the number of characters left, although it is more common to know this number, thus it is generally a better chance of `CHARS()` returning the true number of characters left than `LINES()` returning the true number of lines left.

However, you can be fairly sure that if an implementation returns a number greater than `1`, then that number is the real number of lines (or characters) left in the stream. And simultaneously, if the number returned is `0`, then there is no lines (or characters) left to be read in the stream. But if the number is `1`, then you will never know until you have tried.

Example: File reading idiom

This example shows a common idiom for reading all contents of a file into REXX variables using the `LINES()` and `LINEIN()` built-in functions.

```
i = 1
signal on notready
lleft = lines(file)
do while lleft>0
      do i=i to i+lleft
            line.i = linein(file)
      end
      lleft = lines(file)
```

```
end
notready:
lines.0 = i-1
```

Here, the two nested loops iterates over all the data to be read. The innermost loop reads all data currently available, while the outermost loop checks for more available data. Implementations having a `LINES()` that return only `0` and `1` will generally iterate the outermost loop many times; while implementations that returns the "true" number from `LINES()` generally only iterates the outermost loop once.

There is only one place in this code that `LINEIN()` is called. The `I` variable is incremented at only one place, and the variable `LINES.0` is set in one clause, too. Some redundancy can be removed by setting the `WHILE` expression to:

```
do while word(value('lleft',lines(file)) lleft,2)>0
```

The two assignments to the `LLEFT` variable must be removed. This may look more complicated, but it decreases the number of clauses having a call to `LINES()` from two till one. However, it is less certain that this second solution is more efficient, since using `VALUE()` built-in function can be inefficient over "normal" variable references.

13.4 The Last Line of a Stream

How to handle the last line in a stream is sometimes a problem. If you use a system that stores end-of-lines as special character sequences, and the last part of the data of a stream is an unterminated line, then what is returned when you try to read that part of data?

There are three possible solutions: First, it may interpret the end-of-file itself as an implicit end-of-line, in this case, the partial part of the line is returned, as if it was properly terminated. Second, it may raise the `NOTREADY` condition, since the end-of-file was encountered during reading. Third, if there is any chance of additional data being appended, it may wait until such data are available. The second and third approaches are suitable for persistent and transient files, respectively.

The first approach is sometimes encountered. It has some problems though. If the end of a stream contains the data `ABC<NL>XYZ`, then it might return the string `XYZ` as the last line of the stream. However, suppose the last line was an empty line, then the last part of the stream would be: `ABC<NL>`. Few would argue that there is any line in this stream after the line `ABC`. Thus, the decision whether the end-of-file is an implicit end-of-line depends on whether the would-be last line has zero length or not.

An pragmatic solution is to let the end-of-file only be an implicit end-of-file if the characters immediately in front of it are not an explicit end-of-line character sequence.

However, TRL gives some indications that an end-of-file is not an implicit end-of-line. It says that `LINES()` returns the number of complete lines left, and that `LINEIN()` returns a complete line. On the other hand, the end-of-line sequence is not rigidly defined by TRL, so an implementor is almost free to define end-of-line in just about any terms that are comfortable. Thus, the last line of a stream may be a source of problem if it is not explicitly terminated by an end-of-line.

13.5 Other Parts of the I/O System

This section lists some of the other parts of REXX and the environments around REXX that may be considered a part of the I/O system.

`[Stack.]`

The stack be used to communicate with external environments. At the REXX side, the interface to the stack is the instructions `PUSH`, `PULL`, `PARSE PULL`, and `QUEUE`; and the built-in function `QUEUED()`. These can be used to communicate with external programs by storing data to be transferred on the stack.

`[The STREAM() built-in function.]`

This function is used to control various aspects about the files manipulated with the other standard I/O functions. The standard says very little about this function, and leaves it up to the implementor to specify the rest. Operations like opening, closing, truncating, and changing modes

[The SAY instruction.]

The SAY instruction can be used to write data to the default output stream. If you use redirection, you can indirectly use it to write data to a file.

[The ADDRESS instruction.]

The ADDRESS instruction and commands can be used to operate on files, depending on the power of your host environments and operating system.

[The VALUE() built-in function.]

The function VALUE(), when used with three parameters, can be used to communicate with external host environments and the operating system. However, this depends on the implementation of your interpreter.

[SAA API.]

The SAA API provides several operations that can be used to communicate between processes. In general, SAA API allows you to perform the operations listed above from a binary program written in a language other than REXX.

And of course, I/O is performed whenever a REXX program or external function is started.

13.6 Implementation-Specific Information

This section describes some implementations of stream I/O in REXX. Unfortunately, this has become a very large section, reflecting the fact that stream I/O is an area of many system-specific solutions.

In addition, the variations within this topic are rather large. Regina implements a set of functions that are very close to that of TRL2. The other extreme are ARexx and BRexx, which contain a set of functions which is very close to the standard I/O library of the C programming language.

13.7 Stream I/O in Regina 0.07a

Regina implements stream I/O in a fashion that closely resembles how it is described in TRL2. The following list gives the relevant system-specific information.

[Names for standard streams.]

Regina uses <stdout> and <stdin> as names for the standard output and input streams. Note that the angle brackets are part of the names. You may also access the standard error stream (on systems supporting this stream) under the name <stderr>. In addition, the nullstring is taken to be equivalent to an empty first parameter in the I/O-related built-in functions.

[Implicit opening.]

Regina implicitly opens any file whenever it is first used.

If the first operation is a read, it will be opened in read-only mode. If the first operation is a write, it is opened in read-write mode. In this case if the read-write opening does not succeed, the file is opened in write-only mode. If the file exists, the opening is non-destructive, i.e. that the file is not truncated or overwritten when opened, else it is created if opened in read-write mode.

If you name a file currently open in read-only mode in a write operation, Regina closes the file, and reopens it in read-write mode. The only exception is when you call LINEOUT() with both second and third arguments unspecified, which always closes a file, both for reading and writing. Similarly, if the file was opened in write-only mode, and you use it in a read operation, Regina closes and reopens in read-write mode.

This implicit reopening is enabled by default. You can turn it off by unsetting the extension ExplicitOpen.

[Separate current positions.]

The environment in which Regina operates (ANSI C and POSIX) does not allow separate read and write positions, but only supplies one position for both operations. Regina handles this by maintaining the two

positions internally, and move the "real" current position back and forth depending on whether a read or write operation is next.

[Swapping out file descriptors.]

In order to defend itself against "open-many-close-none" programming, Regina tries to "swap out" files that have been unused for some time. Assume that your operating system limits Regina to 100 simultaneously open files; when your try to open your 101st file, Regina closes the least recently used stream, and recycles its descriptor for the new file. You can enable or disable this recycling with the `SwapFilePtr` extension.

During this recycling, Regina only closes the file in the operating system, but retains all vital information about the file itself. If you re-access the file later, Regina reopens it, and positions the current read and write positions at the correct (i.e. previous) positions. This introduces some uncertainties into stream processing. Renaming a file affects it only if it gets swapped out. Since the swap operation is something the users do not see, it can cause some strange effects.

Regina will not allow a transient stream to be swapped out, since they often are connected to some sort of active partner in the other end, and closing the file might kill the partner or make it impossible to reestablish the stream. So only persistent files are swapped out. Thus, you can still fill the file table in Regina.

[Explicit opening and closing.]

Regina allows streams to be explicitly opened or closed through the use of the built-in function `STREAM()`. The exact syntax of this function is described in section stream. Old versions of Regina supported two non-standard built-in functions `OPEN()` and `CLOSE()` for these operations. These functions are still supported for compatibility reasons, but might be removed in future releases. Their availability is controlled by the `OpenBif` and `CloseBif` extensions.

[Truncation after writing lines.]

If you reposition line-wise the current write position to the middle of a file, Regina truncates the file at the new position. This happens whether data is written during the `LINEOUT()` or not. If not, the file might contain half a line, some lines might disappear, and the linecount would in general be disrupted. The availability of this behavior is controlled by `LineOutTrunc`, which is turned on by default.

Unfortunately, the operation of truncating a file is not part of POSIX, and it might not exist on all systems, so on some rare systems, this truncating will not occur. In order to be able to truncate a file, your machine must have the `ftruncate()` system call in C. If you don't have this, the truncating functionality is not available.

[Caching info on lines left.]

When Regina executes the built-in function `LINES()` for a persistent stream, it caches the number of lines left as an attribute to the stream. In subsequent calls to `LINEIN()`, this number is updated, so that subsequent calls to `LINES()` can retrieve the cached number instead of having to re-scan the rest of the stream, provided that the number is still valid. Some operations will invalidate the count: repositioning the current read position; reading using the character oriented I/O, i.e. `CHARIN()`; and any write operation by the same interpreter on the stream. Ideally, any write operation should invalidate the count, but that might require a large overhead before any operation, in order to check whether the file has been written to by other programs.

This functionality can be controlled by the extension called `CacheLineNo`, which is turned on by default. Note that if you turn that off, you can experience a serious decrease in performance.

The following extra built-in functions relating to stream I/O are defined in Regina. They are provided for extra support and compatibility with other systems. Their support may be discontinued in later versions, and they are likely to be moved to a library of extra support.

CLOSE(streamid)

Closes the stream named by *streamid*. This stream must have been opened by implicit open or by the `OPEN` function call earlier. The function returns `1` if there was any file to close, and `0` if the file was not opened. Note that the return value does not indicate whether the closing was successful. You can use the extension named `CloseBif` with the `OPTIONS` instruction to select or remove this function. This function is now obsolete, instead you should use:

```
STREAM( streamid, 'Command', 'CLOSE' )
```

CLOSE(*myfile*)	1	if stream was open
CLOSE('NOSUCHFILE')	0	if stream didn't exist

OPEN(streamid,access)

Opens the stream named *streamid* with the access *access*. If *access* is not specified, the access `R` will be used. *access* may be the following characters. Only the first character of the *access* is needed.

[R]
(Read) Open for read access. The file pointer will be positioned at the start of the file, and only read operations are allowed.

[W]
(Write) Open for write access and position the current write position at the end of the file. An error is returned if it was not possible to get appropriate access.

The return value from this function is either `1` or `0`, depending on whether the named stream is in opened state after the operation has been performed.

Note that if you open the files "`foobar`" and "`./foobar`" they will point to the same physical file, but Regina interprets them as two different streams, and will open a internal file descriptor for each one. If you try to open an already open stream, using the same name, it will have no effect.

You can use the extension `OpenBif` with the `OPTIONS` instruction to control the availability of this function. This function is now obsolete, but is still kept for compatibility with other interpreters and older versions of Regina. Instead, with Regina you should use:

```
STREAM( streamid, 'C', 'READ'|'WRITE'|'APPEND'|'UPDATE' )
```

OPEN(myfile, 'write')	1	maybe, if successful
OPEN(passwd, 'Write')	0	maybe, if no write access
OPEN('DATA', 'READ')	0	maybe, if successful

The return value from this function is either `1` or `0`, depending on whether the named stream is in opened state after the operation has been performed.

13.8 Functionality to be Implemented Later

This section lists the functionality not yet in Regina, but which is intended to be added later. Most of these are fixes to problems, compatibility modes, etc.

[Indirect naming of streams.]
Currently, streams are named directly, which is a convenient. However, there are a few problems: for instance, it is difficult to write to a file which name is `<stdout>`, simply because that is a reserved name. To fix this, an indirect naming scheme will be provided through the `STREAM()`< built-in function. The functionality will resemble the `OPEN()` built-in function of ARexx.

[Consistence in filehandle swapping.]
When a file handle is currently swapped out in order to avoid filling the system file table, very little checking of consistency is currently performed. At least, vital information about the file should be retained, such as the inode and file system for Unix machines retrieval by the `fstat()` call. When the file is swapped in again, this information must be checked against the file which is reopened. If there is a mismatch, `NOTREADY` should be raised. Similarly, when reopening a file because of a new access mode is requested, the same checking should be performed.

[Files with holes.]

Regina will be changed to allow it to generate files with holes for system where this is relevant. Although standard REXX does not allow this, it is a very common programming idiom for certain systems, and should be allowed. It will, however, be controllable through a extension called `SparseFiles`.

13.9 Stream I/O in ARexx 1.15

ARexx differs considerably from standard REXX with respect to stream I/O. In fact, none of the standard stream functionality of REXX is available in ARexx. Instead, a completely distinct set of functions are used. The differences are so big, that it is useless to describe ARexx stream I/O in terms of standard REXX stream I/O, and everything said so far in this chapter is irrelevant for ARexx. Therefore, we explain the ARexx functionality from scratch.

All in all, the ARexx file I/O interface resembles the functions of the Standard C I/O library, probably because ARexx is written in C, and the ARexx I/O functions are "just" interfaces to the underlying C functions. You may want to check up the documentation for the ANSI C I/O library as described in [ANSIC], [KR], and [PJPlauger].

ARexx uses a two level naming scheme for streams. The file names are bound to a stream name using the `OPEN()` built-in function. In all other I/O functions, only the stream name is used.

`OPEN(`*`name`*`,`*`filename`*`[,`*`mode`*`])`

You use the `OPEN()` built-in function to open a stream connected to a file called *filename* in AmigaDOS. In subsequent I/O calls, you refer to the stream as *name*. These two names can be different.

The *name* parameter cannot already be in use by another stream. If so, the `OPEN()` function fails. Note that the *name* parameter is case-sensitive. The *filename* parameter is not strictly case-sensitive: the case used when creating a new file is preserved, but when referring to an existing file, the name is case-insensitive. This is the usual behavior of AmigaDOS.

If any of the other I/O operations uses a stream name that has not been properly opened using `OPEN()`, that operation fails, because ARexx has no auto-open-on-demand feature.

The optional parameter *mode* can be any of `Read`, `Write`, or `Append`. The mode `Read` opens an existing file and sets the current position to the start of the file. The mode `Append` is identical to `Read`, but sets the current positions to the end-of-file. The mode `Write` creates a new file, i.e. if a file with that name already exists, it is deleted and a new file is created. Thus, with `Write` you always start with an empty file. Note that the terms "read," "write," and "append" are only remotely connected to the mode in which the file is opened. Both reading and writing are allowed for all of these three modes; the mode names only reflect the typical operations of these modes.

The result from `OPEN()` is a boolean value, which is `1` if a file by the specified *name* was successfully opened during the `OPEN()` call, and `0` otherwise.

The number of simultaneously open files is no problem because AmigaDOS allocates files handles dynamically, and thus only limited by the available memory. One system managed 2000 simultaneously open files during a test.

OPEN('infile', 'work:DataFile')	1	if successful
OPEN('work', 'RAM:FooBar', 'Read')	0	if didn't exist
OPEN('output', 'TmpFile', 'W')	1	(re)creates file

`CLOSE(`*`name`*`)`

You use the `CLOSE()` built-in function to close a stream. The parameter *name* must match the first parameter in a call to `OPEN()` earlier in the same program, and must refer to an open stream. The return value is a boolean value that reflects whether there was a file to close (but not whether it was successfully closed).

CLOSE('infile')	1	if stream was previously open
CLOSE('outfile')	0	if stream wasn't previously open

WRITELN(*name*,*string*)

The WRITELN() function writes the contents of *string* as a line to the stream *name*. The *name* parameter must match the value of the first parameter in an earlier call to OPEN(), and must refer to an open stream. The data written is all the characters in *string* immediately followed by the newline character (ASCII <Ctrl-J> for AmigaDOS).

The return value is the number of characters written, including the terminating newline. Thus, a return value of 0 indicates that nothing was written, while a value which is one more than the number of characters in *string* indicates that all data was successfully written to the stream.

When writing a line to the middle of a stream, the old contents is written over, but the stream is not truncated; there is no way to truncate a stream with the ARexx built-in functions. This overwriting can leave partial lines in the stream.

WRITELN('tmp', 'Hello, world!')	1 4	if successful
WRITELN('work', 'Hi there')	0	nothing was written
WRITELN('tmp', 'Hi there')	5	partially successful

WRITECH(*name*,*string*)

The WRITECH() function is identical to WRITELN(), except that the terminating newline character is not added to the data written out. Thus, WRITELN() is suitable for line-wise output, while WRITECH() is useful for character-wise output.

WRITECH('tmp', 'Hello, world!')	1 3	if successful
WRITECH('work', 'Hi there')	0	nothing was written
WRITECH('tmp', 'Hi there')	5	partially successful

READLN(*name*)

The READLN() function reads a line of data from the stream referred to by *name*. The parameter *name* must match the first parameter of an earlier call to OPEN(), i.e. it must be an open stream.

The return value is a string of characters which corresponds to the characters in the stream from and including the current position forward to the first subsequent newline character found. If no newline character is found, the end-of-file is implicitly interpreted as a newline and the end-of-file state is set. However, the data returned to the user never contains the terminating end-of-line.

To differ between the situation where the last line of the stream was implicitly terminated by the end-of-file and where it was explicitly terminated by an end-of-line character sequence, use the EOF() built-in function. The EOF() returns 1 in the former case and 0 in the latter case.

There is a limit in ARexx on the length of lines that you can read in one call to READLN(). If the length of the line in the stream is more than 1000 characters, then only the first 1000 characters are returned. The rest of the line can be read by additional READLN() and READCH() calls. Note that whenever READLN() returns a string of exactly 1000 characters, then no terminating end-of-line was found, and a new call to READLN() must be executed in order to read the rest of the line.

READLN('tmp')	Hello world!	maybe
READLN('work')		maybe, if unsuccessful

READCH(*name*[,*length*])

The READCH() built-in function reads characters from the stream named by the parameter *name*, which must correspond to the first parameter in a previous call to OPEN(). The number of characters read is given by *length*, which must be a non-negative integer. The default value of *length* is 1.

The value returned is the data read, which has the length corresponding to the *length* parameter if no errors occurred.

There is a limit in ARexx for the length of strings that can be read in one call to READCH(). The limit is 65535 bytes, and is a limitation in the maximum size of an ARexx string.

READCH('tmp',3)	Hel	maybe
READCH('tmp')	l	maybe
READCH('tmp',6)	o worl	maybe

EOF(*name*)

The EOF() built-in function tests to see whether the end-of-file has been seen on the stream specified by *name*, which must be an open stream, i.e. the first parameter in a previous call to OPEN().

The return value is 1 if the stream is in end-of-file mode, i.e. if a read operation (either READLN() or READCH()) has seen the end-of-file during its operation. However, reading the last character of the stream does not put the stream in end-of-file mode; you must try to read at least one character past the last character. If the stream is not in end-of-file mode, the return value is 0.

Whenever the stream is in end-of-file mode, it stays there until a call to SEEK() is made. No read or write operation can remove the end-of-file mode, only SEEK() (and closing followed by reopening).

EOF('tmp')	0	maybe
EOF('work ')	1	maybe

SEEK(*name*,*offset*[,*mode*])

The SEEK() built-in function repositions the current position of the file specified by the parameter *name*, which must correspond to an open file, i.e. to the first parameter of a previous call to OPEN(). The current position in the file is set to the byte referred to by the parameter *offset*. Note that *offset* is zero-based, so the first byte in the file is numbered 0. The value returned is the current position in the file after the seek operation has been carried through, using Beginning mode.

If the current position is attempted set past the end-of-file or before the beginning of the file, then the current position is not moved, and the old current position is returned. Note that it is legal to position at the end-of-file, i.e. the position immediately after the last character of the file. If a file contains 12 characters, the valid range for the resulting new current position is 0—12.

The last parameter, *mode*, can take any of the following values:

Beginning, Current, or End. It specify the base of the seeking, i.e. whether it is relative to the first byte, the end-of-file position, or the old current position. For instance: for a 20 byte file with current position 3, then offset 7 for base Beginning is equivalent to offset —13 for base End and offset 4 for Current. Note that only the first character of the *mode* parameter is required, the rest of that parameter is ignored.

SEEK('tmp', 12, 'B')	12	if successful
SEEK('tmp', -4, 'Begin')	12	if previously at 12
SEEK('tmp', -10, 'E')	20	if length is 30
SEEK('tmp', 5)	17	if previously at 12
SEEK('tmp', 5, 'Celcius')	17	only first character in mode matters
SEEK('tmp', 0, 'B')	0	always to start of file

13.10 Main Differences from Standard REXX

Now, as the functionality has been explained, let me point out the main conceptual differences from standard REXX; they are:

[Current position.]
ARexx does not differ between a current read and write position, but uses a common current position for both reading and writing. Further, this current position (which it is called in this documentation) can be set to any byte within the file, and to the end-of-file position. Note that the current position is zero-based.

[Indirect naming.]
The stream I/O operations in ARexx do not get a parameter which is the name of the file. Instead, ARexx uses an indirect naming scheme. The `OPEN()` built-in function binds a REXX stream name for a file to a named file in the AmigaDOS operating system; and later, only the REXX stream name is used in other stream I/O functions operating on that file.

[Special stream names.]
There are two special file names in ARexx: `STDOUT` and `STDIN`, which refer to the standard input file and standard output file. With respect to the indirect naming scheme, these are not file names, but names for open streams; i.e. they can be used in stream I/O operations other than `OPEN()`. For some reason, is it possible to close `STDIN` but not `STDOUT`.

[NOTREADY not supported.]
ARexx has no `NOTREADY` condition. Instead, you must detect errors by calling `EOF()` and checking the return codes from each I/O operations.

[Other things missing.]
In ARexx, all files must be explicitly opened. There is no way to reposition line-wise, except for reading lines and keeping a count yourself.

Of course, ARexx also has a lot of functionality which is not part of standard REXX, like relative repositioning, explicit opening, an end-of-file indicator, etc. But this functionality is descriptive above in the descriptions of extended built-in functions, and it is of less interest here.

When an ARexx script has opened a file in `Write` mode, other ARexx scripts are not allowed to access that file. However, if the file is opened in `Read` or `Append` mode, then other ARexx scripts can open the file too, and the same state of the contents of the file is seen by all scripts.

Note that it is difficult to translate between using standard REXX stream I/O and ARexx stream I/O. In particular, the main problem (other than missing functionality in one of the systems) is the processing of end-of-lines. In standard REXX, the end-of-file is detected by checking whether there is more data left, while in ARexx one checks whether the end-of-file has been read. The following is a common standard REXX idiom:

```
while lines('file')>0   /* for each line available */
        say linein('file')   /* process it */
end
```

In ARexx this becomes:

```
tmp = readln('file')    /* attempt to read first line */
do until eof('file')    /* if EOF was not seen */
      say tmp              /* process line */
      tmp = readln('file') /* attempt to read next line */
end
```

It is hard to mechanically translate between them,

because of the lack of an `EOF()` built-in function in standard REXX, and the lack of a `LINES()` built-in function in ARexx.

Note that in the ARexx example, an improperly terminated last line is not read as an independent line, since `READLN()` searches for an end-of-line character sequence. Thus, in the last invocation `tmp` is set to the last

unterminated line, but EOF() returns true too. To make this different, make the UNTIL subterm of the DO loop check for the expression EOF('file') && TMP<>".

The limit of 1000 characters for READLN() means that a generic line reading routine in ARexx must be similar to this:

```
readline: procedure
      parse arg filename
      line = ''
      do until length(tmpline)<1000
            tmpline = readln(filename)
            line = line || tmpline
      end
      return line
```

This routine calls READLN() until it returns a line that is shorter than 1000 characters. Note that end-of-file checking is ignored, since READLN() returns an empty string a the end-of-stream.

13.11 Stream I/O in BRexx 1.0b

BRexx contains a set of I/O which shows very close relations with the C programming language I/O library. In fact, you should consider consulting the C library documentation for in-depth documentation on this functionality.

BRexx contains a two-level naming scheme: in REXX, streams are referred to by a stream handle, which is an integer; in the operating system files are referred to by a file name, which is a normal string. The function OPEN() is used to bind a file name to a stream handle. However, BRexx I/O functions generally have the ability to get a reference either as a file name and a stream handle, and open the file if appropriate. However, if the name of a file is an integer which can be interpreted as a file descriptor number, it is interpreted as a descriptor rather than a name. Whenever you use BRexx and want to program robust code, always use OPEN() and the descriptor.

If a file is opened by specifying the name in a I/O operation other than OPEN(), and the name is an integer and only one or two higher than the highest current file descriptor, strange things may happen.

Five special streams are defined, having the pseudo file names: <STDIN>, <STDOUT>, <STDERR>, <STDAUX>, and <STDPRN>; and are assigned pre-defined stream handles from 0 to 4, respectively. These refer to the default input, default output, and default error output, default auxiliary output, and printer output. The two last generally refer to the COM1: and LPT1: devices under MS-DOS. Either upper or lower case letter can be used when referring to these four special names.

However, note that if any of these five special files are closed, they can not be reopened again. The reopened file will be just a normal file, having the name e.g. <STDOUT>.

There is a few things you should watch out for with the special files. I/O involving the <STDAUX> and <STDPRN> can cause the Abort, Retry, Ignore message to be shown once for each character that was attempted read or written. It can be boring and tedious to answer R or I if the text string is long. If A is answered, BRexx terminates.

You should never write data to file descriptor 0 (<STDIN>), apparently, it will only disappear. Likewise, never read data to file descriptors 1 and 2 (<STDOUT> and <STDERR>), the former seems to terminate the program while the latter apparently just returns the nullstring. Also be careful with reading from file descriptors 3 and 4, since your program may hang if no data is available.

OPEN(*file*,*mode*)

The OPEN() built-in function opens a file named by *file*, in mode *mode*, and returns an integer which is the number of the stream handle assigned to the file. In general, the stream handle is a non-negative integer, where 0 to 4 are pre-defined for the default streams. If an error occurred during the open operation, the value -1 is returned.

The *mode* parameter specifies the mode in which the file is opened. It consists of two parts: the access mode, and the file mode. The access mode part consists of one single character, which can be r for read, w for write, and a for

append. In addition, the + character can be appended to open a file in both read and write mode. The file mode part can also have of one additional character which can be t for text files and b for binary files. The t mode is default.

The following combinations of + and access mode are possible:

r is non-destructive open for reading; w is destructive open for write-only mode; a is non-destructive open for in append-only mode, i.e. only write operations are allowed, and all write operations must be performed at the end-of-file; r+ is non-destructive open for reading and writing; w+ is destructive open for reading and writing; and a+ is non-destructive open in append update, i.e. reading is allowed anywhere, but writing is allowed only at end-of-file. Destructive mode means that the file is truncated to zero length when opened.

In addition, the b and t characters can be appended in order to open the file in binary or text mode.

These modes are the same as under C, although the t mode character is strictly not in ANSI C. Also note that r, w, and a are mutually exclusive, but one of them must always be present. The mode + is optional, but if present, it must always come immediately after r, w, or a. The t and b modes are optional and mutually exclusive; the default is t. If present, t or b must be the last character in the mode string.

open('myfile','w')	7	perhaps
open('no.such.file','r')	-	if non-existent
	1	
open('c:tmp', 'r+b')	6	perhaps

If two file descriptors are opened to the same file, only the most recently of them works. However, if the most recently descriptor is closed, the least recently starts working again. There may be other strange effects too, so try avoid reopening a file that is already open.

CLOSE(*file*)

The CLOSE() built-in function closes a file that is already open. The parameter *file* can be either a stream handle returned from OPEN() or a file name which has been opened (but for which you do not known the correct stream handle).

The return value of this function seems to be the nullstring in all cases.

close(6)	if open
close(7)	if not open
close('foobar')	perhaps

EOF(*file*)

The EOF() built-in function checks the end-of-file state for the stream given by *file*, which can be either a stream descriptor or a file name. The value returned is 1 if the end-of-file status is set for the stream, and 0 if it is cleared. In addition, the value -1 is returned if an error occurred, for instance if the file is not open.

The end-of-file indicator is set whenever an attempt was made to read at least one character past the last character of the file. Note that reading the last character itself will not set the end-of-file condition.

eof(foo)	0	if not at eof
eof('8')	1	if at eof
eof('no.such.file')	-1	if file isn't open

READ([*file*][,*length*])

The READ() built-in function reads data from the file referred to by the *file* parameter, which can be either a file name or a stream descriptor. If it is a file name, and that file is not currently open, then BRexx opens the file in mode

`rt`. The default value of the first parameter is the default input stream. The data is read from and including the current position.

If the *length* parameter is not specified, a whole line is read, i.e. reading forwards to and including the first end-of-line sequence. However, the end-of-line sequence itself is not returned. If the *length* parameter is specified, it must be a non-negative integer, and specified the number of characters to read.

The data returned is the data read, except that if *length* is not specified, the terminating end-of-line sequence is stripped off. If the last line of a file contains a string unterminated by the end-of-string character sequence, then the end-of-file is implicitly interpreted as an end-of-line. However, in this case the end-of-file state is entered, since the end-of-stream was found while looking for an end-of-line.

read('foo')	one line	reads a complete line
read('foo',5)	anoth	reads parts of a line
read(6)	er line	using a file descriptor
read()	hello there	perhaps, reads line from default input stream

WRITE([*file*][,[*string*][,*dummy*]])

The `WRITE()` built-in function writes a string of data to the stream specified by the *file* parameter, or by default the default output stream. If specified, *file* can be either a file name or a stream descriptor. If it is a file name, and that file is not already open, it is opened using `wt` mode.

The data written is specified by the *string* parameter.

The return value is an integer, which is the number of bytes written during the operation. If the file is opened in text mode, all ASCII newline characters are translated into ASCII `CRLF` character sequences. However, the number returned is not affected by this translation; it remains independent of any text of binary mode. Unfortunately, errors while writing is seldom trapped, so the number returned is generally the number of character that was supposed to be written, independent of whether they was actually written or not.

If a third parameter is specified, the data is written as a line, i.e. including the end-of-line sequence. Else, the data is written as-is, without any end-of-line sequence. Note that with BRexx, the third parameter is considered present if at least the comma in front of it—-the second comma—-is present. This is a bit inconsistent with the standard operations of the `ARG()` built-in function. The value of the third parameter is always ignored, only its presence is considered.

If the second parameter is omitted, only an end-of-line action is written, independent of whether the third parameter is present or not.

write('bar','data')	4	writes four bytes
write('bar','data','nl')	4+??	write a line
write('bar','data',)	4+??	same as previous

SEEK(*file*[,[*offset*][,*origin*]])

The `SEEK()` built-in function moves the current position to a location in the file referred to by *file*. The parameter *file* can be either a file name (which must already be open) or a stream descriptor. This function does not implicitly open files that is not currently open.

The parameter *offset* determines the location of the stream and must be an integer. It defaults to zero. Note that the addressing of bytes within the stream is zero-based.

The third parameter can be any of `TOF`, `CUR`, or `EOF`, in order to set the reference point in which to recon the *offset* location. The three strings refer to top-of-file, current position, and end-of-file, and either upper or lower case can be used. The default value is ???.

The return value of this function is the absolute position of the position in the file after the seek operation has been performed.

The SEEK() function provides a very important additional feature. Whenever a file opened for both reading and writing has been used in a read operation and is to be used in a write operation next (or vice versa), then a call to SEEK() must be performed between the two I/O calls. In other words, after a read only a seeking and reading may occur; after a write, only seeking and writing may occur; and after a seek, reading, writing, and seeking may occur.

13.12 Problems with Binary and Text Modes

Under the MS-DOS operating system, the end-of-line character sequence is <CR><LF>, while in C, the end-of-line sequence is only <LF>. This opens for some very strange effects.

When an MS-DOS file is opened for read in text mode by BRexx, all <CR><LF> character sequences in file data are translated to <LF> when transferred into the C program. Further, BRexx, which is a C program, interprets <LF> as an end-of-line character sequence. However, if the file is opened in binary mode, then the first translation from <CR><LF> in the file to <LF> into the C program is not performed. Consequently, if a file that really is a text file is opened as a binary file and read line-wise, all lines would appear to have a trailing <CR> character.

Similarly, <LF> written by the C program is translated to <CR><LF> in the file. This is always done when the file is opened in text mode. When the file is opened in binary mode, all data is transferred without any alterations. Thus, when writing lines to a file which is opened for write in binary mode, the lines appear to have only <LF>, not <CR><LF>. If later opened as a text file, this is not recognized as an end-of-line sequence.

Example: Differing end-of-lines

Here is an example of how an incorrect choice of file type can corrupt data. Assume BRexx running under MS-DOS, using <CR><LF> as a end-of-line sequence in text files, but the system calls translating this to <LF> in the file I/O interface. Consider the following code.

```
file = open('testfile.dat', 'wt')      /* text mode */
call write file, '45464748'x, 'dummy'  /* i.e. 'abcd' */
call write file, '65666768'x, 'dummy'  /* i.e. 'ABCD' */
call close file
file = open('testfile.dat', 'rb')      /* binary mode */
say c2x(read(file))                    /* says '454647480D' */
say c2x(read(file))                    /* says '656667680D' */
call close file
```

Here, two lines of four characters each are written to the file, while when reading, two lines of five characters are read. The reason is simply that the writing was in text mode, so the end-of-line character sequence was <CR><LF>; while the reading was in binary mode, so the end-of-line character sequence was just <LF>. Thus, the <CR> preceding the <LF> is taken to be part of the line during the read.

To avoid this, be very careful about using the correct mode when opening files. Failure to do so will almost certainly give strange effects.

Extensions

This chapter describes how extensions to Regina are implemented. The whole contents of this chapter is specific for Regina.

1. Why Have Extensions

Why do we need extensions? Well, there are a number of reasons, although not all of these are very good reasons:

- Adaptations to new environments may require new functionality in order to easily interface to the operating system.
- Extending the language with more power, to facilitate programming.
- Sometimes, a lot of time can be saved if certain assumptions are met, so an extension might be implemented to allow programmers to take shortcuts.
- When a program is ported from one platform to another, parts of the code may depend of non-standard features not available on the platform being ported to. In this situation, the availability of extensions that implement the feature may be of great help to the programmer.
- The implementor had some good idea during development.

Extensions arise from holes in the functionality. Whether they will survive or not depends on how they are perceived by programmers; if perceived as useful, they will probably be used and thus supported in more interpreters.

2. Extensions and Standard REXX

In standard REXX, the `OPTIONS` instruction provides a “hook” for extensions. It takes any type of parameters, and interprets them in a system-dependent manner.

The format and legal values of the parameters for the `OPTIONS` instruction is clearly implementation dependent [TRL2, p62].

3. Specifying Extensions in Regina

In Regina there are three level of extensions. Each independent extension has its own name. Exactly what an independent extension is, will depend on the viewer, but a classification has been done, and is listed at the end of this chapter.

At the lowest level are these “atomic” extensions. Then there are some “meta-extensions”. These are collections of other extensions which belongs together in some manner. If you need the extension for creating “buffers” on the stack, it would be logical to use the extension to remove buffers from the stack too. Therefore, all the individual extensions for operations that handle buffers in the stack can be named by such a “meta-extensions”. At the end of this chapter, there is a list of all the meta-extensions, and which extensions they include.

At the top is “standards”. These are sets of extensions that makes the interpreter behave in a fashion compatible with some standard. Note that “standard” is used very liberally, since it may refer to other implementations of REXX. However, this description of how the extensions are structured is only followed to some extent. Where practical, the structure has been deviated.

4. The Trouble Begins

There is one very big problem with extensions. If you want to be able to turn them on and off during execution, then your program has to be a bit careful.

More and more REXX interpreters (including Regina seem to do a parsing when the interpreter is started. The “old” way was to postpone the parsing of each clause until it was actually executed. This leads to the problem mentioned.

Suppose you want to use an extension that allows a slightly different syntax, for the sake of the argument, let us assume that you allow an expression after the SELECT keyword. Also assume that this extension is only allowed in extended more, not in “standard mode”. However, since Regina parses the source code only once (typically at the starts of the program), the problem is a catch-22: the extension can only be turned on after parsing the program, but it is needed before parsing. This also applies to a lot of other REXX interpreters, and all REXX compilers and preprocessors.

If the extension is not turned on during parsing, it will generate a syntax error, but the parsing is all done before the first clause is executed. Consequently, this extension can not be turned on during execution, it has to be set before the parsing starts.

Therefore, there are two alternative ways to invoke a set of extensions.

- It can be invoked by using the `-e` option to the interpreter. The word following the option is the extension or standard to invoke. Multiple `-e` options can be specified.

- It can be invoked by setting the environment variable `REXXEXTS`, which must be a string of the same format as the parameters to the `OPTIONS` clause.

5. The Format of the `OPTIONS` clause

The format of the `OPTIONS` clause is very simple, it is followed by any REXX string expression, which is interpreted as a set of space separated words. The words are treated strictly in order from left to right, and each word can change zero or more extension settings.

Each extension has a name. If the word being treated matches that name, that extension will be turned on. However, if the word being treated matches the name of an extension but has the prefix `NO`, then that extension is turned off. If the word does not match any extensions, then it is simply ignored, without creating any errors or raising any conditions.

Example: Extensions changing parsing

An example of the same is the `UPPER` instruction. In the following piece of code the same clause is interpreted in two completely different ways:

```
options ‘NOUPPER’
do i=1 to 2
      if i=2 then options ‘UPPER’
      upper foo bar
end
```

In the first iteration of the loop, the clause starting with the token UPPER will be a command, issuing the string resulting from evaluating the expression upper foo bar. However, in the second iteration of the loop, the same clause is interpreted as an UPPER instruction. Since these two statements has very different syntax, it seems impossible to handle both in the same program. Regina tries to handle this by "allowing" both syntaxes when parsing the source code, and selecting the right one when interpreting the statement in question.

Regina's frequent usage of extensions may slow down execution. To illustrate how this can happen, consider the OPEN() extra built-in function. As this is an extension, it might be dynamically included and excluded from the scope of currently defined function. Thus, if the function is used in a loop, it might be in the scope during the first iteration, but not the second. Thus, Regina can not cache anything relating to this function, since the cached information may be outdated later. As a consequence, Regina must look up the function in the table of functions for each invocation. To avoid this, you can set the extension CACHEEXT, which tells Regina to cache info whenever possible, without regards to whether this may render useless later executions of OPTIONS.

6. Why You Should Seriously Consider Not Using Extensions

7. The Fundamental Extensions

Here is a description of all "atomic" extensions in Regina:

[BUFTYPE_BIF]
: Allows calling the built-in function BUFTYPE(), which will write out all the contents of the stack, indicating the buffers, if there are any. The idea is taken from VM/CMS, and its command named BUFTYPE.

[CACHEEXT]
: Tells Regina that information should be cached whenever possible, even when this will render future execution of the OPTIONS instruction useless. Thus, if you use e.g. the OPEN() extra built-in function, and you set CACHEEXT, then you may experience that the OPEN() function does not disappear from the current scope when you set the NOOPEN_BIF extension.

  Whether or not a removal of an extension really do happen is unspecified when CACHEEXT has been called at least once. Effectively, info cached during the period when CACHEEXT was in effect might not be "uncached". The advantage of CACHEEXT is efficiency when you do not need to do a lot of toggling of some extension.

[CLOSE protect_BIF]
: Allows the CLOSE() extra built-in function, which allows the program to explicitly close a stream.

[DESBUFprotect_BIF]
: Allows calling the built-in function DESBUF(), to remove all contents and all buffers from the stack. This function is an idea taken from the program by the same name under VM/CMS.

[DROPBUFprotect_BIF]
: Allows calling the built-in function DROPBUF(), to removed one of more buffers from the stack. This function is an idea take from the program by the same name under VM/CMS.

[FIND_BIF]
: Allows calling the FIND() extra built-in function, which is a compatibility function with VM/CMS. This function is really equivalent to POS(), but the parameters are somewhat reversed, and some find FIND() more intuitive. Besides, this extension helps porting.

[FLUSHSTACK]
: Tells the interpreter that whenever a command clause instructs the interpreter to flush the commands output on the stack, and simultaneously take the input from the stack, then the interpreter will not buffer the output but flush it to the real stack before the command has terminated. That way, the command may read its own

output. The default setting for Regina is not to flush, i.e. NOFLUSHSTACK, which tells interpreter to temporary buffer all output lines, and flush them to the stack when the command has finished.

[LINEOUTTRUNC]
This options tells the interpreter that whenever the LINEOUT() built-in function is executed for a persistent file, the file will be truncated after the newly written line, if necessary. This is the default setting of Regina, unless your system does not have the ftruncate() system call. The complement option is NOLINEOUTTRUNC.

[MAKEBUF_BIF]
Allows calling the built-in function MAKEBUF(), to create a buffer on the stack. This function is an idea taken from a program by the same name under VM/CMS.

[OPEN_BIF]
Adds the extra built-in function OPEN() , which is used for explicitly opening streams.

[PRUNE_TRACE]
Makes deeply nested routines be displayed at one line. Instead of indenting the trace output at a very long line (possibly wrapping over several lines on the screen). It displays [...] at the start of the line, indicating that parts of the white space of the line has been removed.

8. Meta-extensions

[BUFFERS]
Combination of BUFTYPE_BIF, DESBUF_BIF, DROPBUF_BIF and MAKEBUF_BIF.

[FILEIO]
Introduces some commonly used extra features for handling files. This is a combination of OPEN_BIF() and CLOSE_BIF(), which allow the programmer to explicitly open and close files.

9. Semi-standards

[CMS]
A set of extensions that stems from the VM/CMS operating system. Basically, this includes the most common extensions in the VM/CMS version of REXX, in addition of some functions that perform task normally done with commands under VM/CMS.

[VMS]
A set of interface functions to the VMS operating system. Basically, this makes the REXX programming under VMS as powerful as programming directly in DCL.

[UNIX]
A set of interface functionality to the Unix operating system. Basically, this includes some functions that are normally called as commands when programming Unix shell scripts. Although it is possible to call these as commands in Regina, there are considerable speed improvements in implementing them as built-in functions.

10. Standards

[ALL]

[ANSI]

[DEFAULT]

[NONE]

[SAA]

[TRL1]
REXX Language level 3.50, as described in [TRL1].

[TRL2]
REXX Language level 4.00, as described in [TRL2].

Also, for those of these standards that have a accepted REXX language level number, that number can be used, provided that it matches character by character (i.e. not by numeric value). Thus, you can use `3.50` as a synonym for `TRL1`, and `4.00` as a synonym for `TRL2`.

Option	ALL	ANSI	DEF	NONE	SAA	TRL1	TRL2
BUFTYPE_BIF	yes	??	yes	no	??	no	no
CLOSE_BIF	yes	??	yes	no	??	no	no
CACHEEXT	no	no	no	no	no	no	no
DESBUF_BIF	yes	??	yes	no	??	no	no
DROPBUF_BIF	yes	??	yes	no	??	no	no
FIND_BIF	yes	??	yes	no	??	no	no
FLUSHSTACK	yes	??	no	no	??	no	no
LINEOUTTRUNC	yes	??	yes	no	??	no	no
MAKEBUF_BIF	yes	??	yes	no	??	no	no
OPEN_BIF	yes	??	yes	no	??	no	no
PRUNE_TRACE	yes	no	yes	no	no	no	no
UPPER_CLAUSE	yes	??	yes	no	??	no	no

Note that the standard and default interpreter is a REXX language level 4.00 interpreter. All other functionality is extensions. In fact, the features in 4.00 that does not exist in 3.50 are “inverse” extensions, i.e. the extension is to remove the functionality only in 4.00.

The Stack

In this chapter, the stack and operations manipulating the stack are discussed. Since the stack is external to the REXX language, there are large differences between implementations with respect to the stack. These differences are attempted described in the latter part of this chapter.

Another goal of this chapter is to try to describe both the "real" standards and some of the most commonly used de facto standards related to stack operation. Where something is not a part of any defined standard, this is clearly labeled. Also, some liberties have been taken in order to create a coherent vocabulary on a field where very little standardization has taken place.

1. Background and history

In the various definitions of REXX, there are numerous references to the "stack" (often called the "external data queue", or just the "queue"). It is a structure capable of storing information, but it is not a part of the REXX language itself. Rather, it is a part of the external environment supporting a REXX implementation.

Originally, the references to the stack was introduced into REXX because of the strong binding between REXX and IBM mainframes in the early history of REXX [BMARKS]. Most (all?) of the operating systems for these machines support a stack, and many of their script programming idioms involve the stack. Therefore, it was quite natural to introduce an interface to the stack into REXX, and consequently today many of the programming paradigms of REXX involve a stack.

Unfortunately, this introduced an element of incompatibility into REXX, as the stack is not in general supported for other operating systems. Consequently, REXX implementors often must implement a stack as well of the core REXX interpreter. Since no authoritative definition of the stack exists, considerable differences between various implementations. Ironically, although the stack was introduced to help communication between separate programs, the interpreter-specific implementations of stacks may actually be a hindrance against compatibility between different interpreters.

The stack may have "seemed like a good idea at the time", but in hindsight, it was probably a bad move, since it made REXX more dependent on the host operating system and its interfaces.

2. General functionality of the stack

This section describes the functionality generally available in implementations of stacks. The basic functionality described here will be complemented with information on specific implementations later. Unless explicitly labeled otherwise, this functionality is available in all standards treated in this documentation.

2.1 Basic functionality

Below is listed the general functionality of the stack, in order of decreasing compatibility. I.e. the functionality listed first is more likely to be a part of all implementations than the ones listed at the end of the list.

- The stack is a data structure, which strings can either be inserted into or extracted from. The strings in the stack are stored in a linear order. Extraction and insertion works at a granularity of a complete string, i.e. it is not possible to insert or extract parts of string.

- The stack has two ends: a top and a bottom. New strings can be inserted into the stack in both ends, but strings can only be extracted from the top of the stack.

- There exists a way of counting the number of strings currently stored in the stack.

A stack is often compared with the pile of plates you often find in cantinas. It allows you to either add new plates at the top of the pile or take old plates from the top. When a plate is taken from the pile, it will be the most recently plate (that is still present) added to the pile. Stack operating in REXX work the same way, although there also allow "plates" to be added to the bottom of the pile.

- There might be an implementation-specific limit on the length and number of strings stored in the stack. Ideally, the maximum length will be fairly large, at least 2**16, although some implementations are likely to enforce shorter limits. Similarly, there might be a limit on the number of strings that can be simultaneously stored in the stack. Ideally, there should be no such limit.

- It is natural that there are limits imposed on the amount of memory occupied by the strings in the stack. Some implementations are likely to reserve a fixed (but perhaps configurable) amount of memory for this purpose while others can dynamically re-size the stack as long as enough memory is available.

- Some implementations might restrict the set of characters allowed in strings in the stack, although ideally, all characters should be allowed, even characters normally used for end-of-line or end-of-string.

This documentation use the term "string", while "line" is in common use elsewhere. The term is used because the strings in the stack are not inherently interpreted as lines (having an implied end-of-line), only as a string.

Note that the stack itself is not a part of REXX, only the parts which interface to the stack.

Example: Using the stack to transfer parameters

This is a common REXX idiom used in several situations for special parameter passing. The following code illustrates its use:

```
do i=1 to 10              /* for each parameter string        */
      queue string.1         /* put the string on the stack   */
end
call subrout 10           /* call the subroutine              */
exit

subrout: procedure        /* the definition of the subroutine */
      do j=1 to arg(1)       /* for each parameter passed */
            parse pull line.j   /* retrieve the parameter */
      end
      ...                    /*do something with the parameters*/
      return
```

In this example, ten parameter strings are transferred to the subroutine SUBROUT. The parameters are stored in the stack, and only the number of parameters are transferred as a "real" argument.

There are several advantages: first, one avoids problems related to exposing variable names. Since the data is stored on the stack, there is no need to refer to the variable names and bind the variables in the subroutine to variables in the caller routine. In [TRL1], indirect references to variables in PROCEDURE EXPOSE is illegal, and this method circumvent the problem.

Two other ways around this problem is to use INTERPRET for the PROCEDURE EXPOSE instruction in order to dynamically determine which variables to expose; or to use the VALUE() built-in function (with its two first parameters). The former is incompatible with TRL2, while the latter is incompatible with TRL1. Using the stack can solve the problem in a fashion compatible with both standards. Anyway, if the called routine is an external routine, then exposing does not work, so using the stack to transfer values may be the only solution.

Another advantage of this idiom; TRL only requires implementations to support 10 parameters for subroutines. Although there are no reasons why an implementation should set a limit for the number of parameters a routine can get, you should use another mechanism than arguments when the number of strings is greater than 10. Using the stack fixes this.

2.2 LIFO and FIFO stack operations

As already mentioned, the stack is a linear list of strings. Obviously, this list has two ends. Strings can only be extracted from one end, while strings can be added to both ends.

If a set of new strings are added to the same end as they are later extracted from, the strings will be extracted in the reversed order with respect to the order in which they were added. This is called stacking "LIFO", which means "last-in-first-out", meaning that the last string stacked, will be the first string extracted, i.e. reversal of the order.

Similarly, when a set of strings are stacked in the end opposite to the end which they are later extracted from, they will be extracted in the same order in which they were stacked. This is referred to as "FIFO" stacking, meaning "first-in-first-out".

The FIFO method of stacking is also sometimes referred to as "queueing", while the LIFO method is sometimes referred to as "stacking" or "pushing".

2.3 Using multiple buffers in the stack

The concept of buffers and everything directly related to buffers lay without the domain of standard REXX. Thus, this section describes a de facto standard.

Some implementations support "buffers", which are a means of focusing on a part of the stack. When creating a new buffer, the old contents of the stack is somewhat insulated from the effects of stack operations. When the buffer is removed, the state of the old buffer i restored, to some extent: Whenever a string is read from the stack, and the topmost buffer on the stack is empty, then that buffer will be destroyed. Consequently, if this situation has arisen, dropping buffers will not restore the state of the stack before the buffer was created.

The functionality of buffers, and their effect on other stack operations may differ considerably between implementations.

Whenever a queuing operations is performed (e.g. by the `QUEUE` instruction), then the new string is inserted into the bottom of the topmost buffer, not the bottom of the stack. This is the same if the stack has no buffers, but else, the outcome of the queuing operation can be very different.

With IBM mainframe operating systems like CMS, buffers can be inserted on the top of the stack. To perform buffer operations, operating system commands are used. It may be instructional to list the buffer operations of CMS:

`[DESBUF]`
: Removes all strings and buffers from the stack, and leaves the stack clean and empty. It is often used instead of repeated calls to `DROPBUF`. It always returns the value zero.

`[DROPBUF]`
: Removes zero or more buffers from the stack. It takes one parameter which can be omitted, and which must be an integer position if specified, and is the assigned number of the bottom-most buffer to be removed, i.e. that buffer and all buffers above it (and of course, all the strings in these buffers) are to be removed. If the parameter is not specified, only the topmost buffer is removed. The return valued is always zero, unless an error occurred.

`[MAKEBUF]`
: Makes a new buffer on the stack, starting at the current top of the stack. The return code (as stored in the special variable `RC`) is the number of buffers currently on the stack after the new buffer has been added. Obviously, this will be a positive integer. This program takes no parameters.

One might regard a buffer as a sort of bookmark, which is inserted into the stack, so that a subsequent `DROPBUF` command can remove the stack down to a particular such bookmark.

When such a mark is located on the top of the stack, and a PULL instruction is executed, the buffer mark is implicitly destroyed when the PULL instruction reads the string below the buffer mark. This is to say that a buffer can be destroyed by either a DESBUF command, a DROPBUF command, or a read from the stack (by either the PULL or PARSE PULL instructions).

2.4 The zeroth buffer

Normally, data pushed on the stack is added to the top of the stack. When a stack contains only one buffer, the strings in that buffer are the strings stored above that buffer-mark. The strings below it are not part of the first buffer; instead, they are said to belong to the zeroth buffer.

Thus, all strings from the bottom of the stack, up till the first buffer mark (or the top of the stack if no buffers exist) is said to be the strings in the zeroth buffer. However, note that the zeroth buffer is only defined implicitly. Thus, it can not really be removed by calling DROP; only the strings in the zeroth buffer are removed. Afterwards, the zeroth buffer will still contain all strings at the bottom of the stack, up till the first buffer mark (if existing).

Example: Process all strings in the stack

This is a common REXX idiom, where a loop iterates over all the strings currently in the stack, but otherwise leave the stack untouched. Supposing the routine PROCESS() exists, and do to processing with its parameter and return the processed string:

```
do i=1 to 5                /* just to fill the stack      */
      push 'line #' i
end

do queued()                /* foreach line in the stack   */
      parse pull line         /* fetch the line              */
      queue process(line)     /* put back the processed line */
end
```

Here, it is important to use QUEUE to put the strings back into the stack, not PUSH, else the loop will iterate the correct number of times, but only operate on the same data string. It is also important that the stack does not contain any buffers. Since QUEUE will insert into the bottom of the topmost buffer, the loop would iterate the correct number of times, but only on a part of the stack. Thus, the topmost part of the strings in the stack would be processed multiple times.

Example: How to empty the stack

The following short example shows how you can most easily empty the stack:

```
do i=1 to 5               /* Just to fill the stack */
      push 'line #' i
end

do queued()               /* For each line in the stack */
      pull                   /* Remove the line from the stack */
end
```

This is trivially simple, but there are several interesting and subtle notes to make about this example. First, if the number of strings in the stack is likely to change, due to some external process, then the DO clause should perhaps better be written as:

```
do i=1 to 5               /* Just to file the stack */
      push 'line #' i
end

do while queued()>0       /* While the stack is not empty */
      pull                   /* Remove a line from the stack */
```

```
end
```

This will in general mean more work for the interpreter, as it is now required to check the number of strings in the stack for each iteration, while for the previous code fragment, the number of strings is only checked once. Another point is that this might not remove all buffers from the stack. Suppose the zeroth buffer is empty, i.e. there exists an buffer which was put on the stack when the stack was empty. This buffer is removed in any of the following situations: calling DESBUF, calling DROPBUF (sometimes), or reading a string below the buffer mark. Since there are no strings below the buffer mark, pulling a string from the stack would make the interpreter read from the keyboard, and hang the interpreter.

Thus, the only "safe" way to remove the string and buffers from the stack, without side effects, is to call DESBUF or DROPBUF. On the other hand, if you only want to make sure that there are no strings in the buffer, the method described here is more suitable, since it is far more compatible (although possibly not so efficient). But anyway, buffers are not a compatible construct, so it does not matter so much.

2.5 Creating new stacks

The description of multiple stack operations in this section, is not part of standard REXX. Thus, this section describes a de facto standard and you may find that few implementations support these operations.

Just as the operations described above let the REXX programmer use multiple buffers within one stack, there exists another set of operations which let the programmer create multiple stacks. There is really nothing fancy about this, except that a command will swap the stack the interpreter correctly uses with another stack.

To the interpreter this is really equivalent to a situation where a command empties the current stack, and sets up a new stack. When one stack is empty, and the REXX program tries to read from the stack, the request will not "overflow" to the previous stack (as requests to an empty buffer "overflows" to the previous buffer). Thus, the use of multiple stacks has even less direct impact on REXX interpreters than multiple buffers.

Here, it is instructive to list the commands operating multiple stacks that exists. This list has been taken from the MVS environment, according to [REXXSAA].

[DELSTACK]
: Is used to remove the most currently stack, and make the most recent of the saved stacks the current stack. When there are no saved stacks, the current stack is emptied.

[NEWSTACK]
: Creates a new stack, which becomes the current stack. The old current stack is put on the top of the list of saved stacks, and can be retrieved as the current stack by a subsequent DELSTACK.

[QBUF]
: Counts the number of buffers in the current stack, and returns that number as the return value. A REXX program starting this command can retrieve this value as the special variable RC.

[QELEM]
: Counts the number of strings (i.e. elements) in the current stack, and returns that value as the return value of the command. This value can be retrieved in REXX as the special variable RC. This operation is equivalent to the QUEUED() built-in function in REXX; it has been probably included for the benefit of other script languages that have less functionality than REXX.

[QSTACK]
: Counts the number of stacks (including the current stack) and returns the value as the return value from the command. This number can be retrieved in REXX as the special variable RC.

One can regard multiple buffers and stacks as two ways of insulating the stack; where multiple stacks are a deeper and more insulating method than buffers. Note that each stack can contain multiple buffers, while a buffer can not contain any stacks. The term "hard buffers" has been used about multiple stacks, as opposed to normal buffers, which are sometimes called "soft buffers".

Also note that neither multiple stacks nor buffers are part of standard REXX, so you might come across implementations that support only multiple stacks, only buffers, or even none of them.

Example: Counting the number of buffers

In order to count the number of buffers on the stack, the following method can be used (Regina syntax has been used for buffer handling). This method is equivalent to the QBUF command described above.

```
buffers = makebuf() - 1
call dropbuf
```

This will store the number of buffers in the stack in the variable buffers. However, just as for the other examples using buffers, this example also suffers from the fact that buffer handling is fairly non-standard. Thus, you will have to adapt the code to whatever system you want to use.

3. The interface between REXX and the stack

As defined in TRL, the interface to the stack consists of the PARSE PULL, PULL, PUSH, and QUEUE instructions; and the QUEUED() built-in function.

There exists a binary interface to the stack in SAA, see the chapter on the SAA API interface. This interface consists of the RXMSQ exit handler and the QUENAME value of the RXSHV_PRIV request of the RexxVariablePool() function of the variable pool interface.

4. Strategies for implementing stacks

As mentioned, stacks are rarely a part of the operating system. Therefore, under most operating systems, REXX interpreters have to implement their own stacks. There are several strategies for doing this, some which are listed below.

[In the operating system.]
This is of course "the right way" to do it. However, it requires that the definition of the operating system is such that stacks are supported. Currently, only IBM mainframe-based systems support stack, together with a few other systems that have included stacks as a consequence of making REXX a main scripting language (Amiga and OS/2 come to mind).

[As a device driver.]
This is really just a variation of making the stack a part of the operating system. However, in some systems, drivers can be added very easily to the system. Drivers are often filesystem-based, in which case driver-based stack operations must operate on a file or pseudo-file. But for some systems, adding a driver requires much more profound changes, reconfiguration, and often system privileges. In all cases, drivers are likely to be very system specific.

[As a daemon.]
A "daemon" is background process that does some housekeeping service, e.g. handling mail from remote systems. Implementing a stack as a daemon is only slightly simpler than using a driver, but the main idea is the same for both approaches.

[In the interpreter.]
Using this approach, the stack is built into the interpreter as a sort of extension. This is often the simplest way, since it require very little coordination with other programs during run-time. The main problem is that the stack becomes private to the interpreter, so two interpreters can not use the same stack; not even if they are two invocations of the same interpreter.

These items are listed in the order of how closely they are coupled to the operating system: the first items are very closely, while the last items are loosely coupled. The more closely coupled the implementation of a stack is coupled to

the operating system, the better is the chance that several interpreters on the same system can communicate in a compatible way, using the stack.

There is room for several hybrid solutions, based on the four fundamental approaches. For instance, a built-in stack can also act as a daemon.

Example: Commands takes input from the stack

In the example above, the routine that is called takes its arguments from the stack. Similarly, commands to an external environment can get their arguments in the same way. Here is an example of how to do it:

```
queue 'anonymous'               /* the username */
queue 'user@node'               /* the password */
queue 'dir'                     /* first command */
queue 'exit'                    /* second command */
address command 'FTP flipper.pvv.unit.no'
```

Although this is very convenient in some situations, there is also considerable disadvantages with this method: There is no real interactive communication between the interpreter and the command; i.e. all input meant for the command must be set up before the command itself is invoked. Consequently, if one of the input lines to the command provokes an error, there is very little error handling facility. Commonly, such an error might start a cascade of errors, as the remaining input lines are likely to be invalid, or even be interpreted in a context different from what they were intended.

As with all commands involving the stack, it is important to push or queue the correct order.

Using this technique, a program can "fool" a command to do almost anything, by storing the correct input on the stack. However, there is a big disadvantage: Since the stack is implementation-dependent, it is not certain that a command will take its input from the stack. For some systems, this is the default, while for other systems, this is only possible through some explicit action. Some systems might not even allow commands to take their input from the stack at all.

Example: "Execing" commands

Many script programming languages can only execute commands while still running, or at most start a new command immediately after the termination (like the `exec()` system call in Unix). However, the stack can be used on some systems to set up the system to execute one or more commands after the current script terminates. Here is an example:

```
push 'ls'          /* finally execute 'ls' */
push 'who'         /* then execute 'who'   */
push 'pwd'         /* first execute 'pwd'  */
exit 0
```

Supposing that the system reads its commands from the stack if the stack is not empty, then this script will terminate after having set up the stack so that the three commands `pwd`, `who` and `ls` will be run in that sequence. Note the order, if `QUEUE` had been used, the order would be the opposite, which is perhaps more intuitive (assuming the topmost buffer is empty).

As with the example above, this too is only relevant for some systems, thus is not very compatible, and you should be careful when using it. It also suffers from the lack of interactivity, error handling, and the importance of the order in which the strings are pushed or queued. For all practical reasons, this is just a special case.

Using the stack to "leave behind" command names and input only works for systems where command interpreters and commands reads their input from the stack. This is in general true for IBM mainframe systems, but very few other systems.

5. Specific implementations of stacks

Below is listed implementation-specific documentation, with respect to stacks, for some interpreters.

5.1 Implementation of the stack in Regina 0.05h

In Regina, the stack is implemented as an integral, private part of the interpreter. The advantage of this is that stack operations are very fast. On the other hand, it means that two interpreters running on the same machine does not use the same stack. Further, it means that a program can not on its own initiative communicate with the stack; such piping must be set up by the interpreter at the invocation time of the program.

Whenever the REXX programmer wants to execute a command and let that command either flush the output to the stack, or read its input from the stack, this has to be arranged by the interpreter itself. In Regina this is normally done by prepending or appending certain terms to the command to be executed.

Consider the following command clauses for Regina:

```
'ls >LIFO'
'who >FIFO'
'LIFO> wc'
'ps'
'LIFO> sort >FIFO'
```

For all these commands, the "piping" terms are stripped off the command string before the command is sent to the command interpreter of the operating system. Thus, the command interpreter only sees the commands `ls`, `who`, `wc`, and `sort`. The terms stripped off, are used as indicators of how the input and output is to be coupled with the stack. Note that it is important not to confuse the redirection of output to the stack and input from the stack in Regina with the redirection of the Unix shells. The two can be mixed in command lines, but are still two different concepts.

The first command will execute the `ls` command, and redirect the output from it to the stack in a LIFO fashion. The second executes the command `who` and redirects the output to the stack to, but in a FIFO fashion. The third command executes the `wc`, but lets the standard input of that command come from the stack. Actually, it is irrelevant whether `FIFO>` or `LIFO>` is used for input; the strings are read from the top of the stack in both cases. The fourth command is a plain `ps` command without any redirection to or from the stack. The last command executes the `sort` program and lets it read its input from the stack, and redirect the output to the stack.

Regina allows a command to take both an input and an output "redirection" to a stack, as showed in the last example above. However, it also guarantees that the output is not available in the stack before the command has terminated. The output from the command is stored in a temporary stack, and flushed to the ordinary stack after the command is terminated. Thus, the command will not start to read its own output.

Note that this temporary buffering of command output is the default behavior, which might be set up to something different at your site.

In addition, you can change it through the `OPTIONS` instruction, by using either `FLUSHSTACK` or `BUFFERSTACK` as "parameters".

Furthermore, Regina supports the standard TRL REXX stack interface functionality, like `PARSE PULL`, `PULL`, `QUEUE`, `PUSH`, the `QUEUED()` built-in function, and the SAA API stack interface. In addition, there are a few extra built-in functions, which are supposed to provide compatibility with other REXX implementations. These are:

Again, note the difference between Regina's redirection and Unix redirection. In Regina, only the terms `LIFO>` and `FIFO>` (when first in the command string), and the terms `>LIFO` and `>FIFO` (when last in the command string), will be interpreted as redirection directives. These terms will be stripped off the command string. All other redirection directives will be left untouched. If you should happen to need to redirect output from a Unix command to the file `FIFO` or `LIFO`, then you can append a space at the end. That will make Regina ignore the redirection term and the space is ignored by Unix.

Note that this particular form of redirection of command input and output will most probably disappear in future versions of Regina, where it will probably be replaced by an extended ADDRESS instruction.

BUFTYPE()

This function is used for displaying the contents of the stack. It will display both the string and notify where the buffers are displayed. It is meant for debugging, especially interactive, when you need to obtain information about the contents of the stack. It always returns the nullstring, and takes no parameters.

Here is an example of the output from calling BUFTYPE (note that the second and fourth buffers are empty):

```
==> Lines: 4
==> Buffer: 3
“fourth line pushed, in third buffer”
==> Buffer: 2
==> Buffer: 1
“third line pushed, in first buffer”
==> Buffer: 0
“second line pushed, in ‘zeroth’ buffer”
“first line pushed, in ‘zeroth’ buffer”
==> End of Stack
```

BUFTYPE()

DESBUF()

This function removes all buffers on the stack, it is really just a way of clearing the whole stack for buffers as well as strings. Functionally, it is equivalent to executing DROPBUF with a parameter of 0. (Actually, this is a lie, since DROPBUF is not able to take zero as a parameter. Rather, it is equivalent to executing DROPBUF with 1 as parameter and then executing DROPBUF without a parameter, but this is a subtle point.) It will return the number of buffers left on the stack after the function has been executed. This should be 0 in all cases.

```
DESBUF()
```

DROPBUF([*number*])

This function will remove zero or more buffers from the stack. Called without a parameter, it will remove the topmost buffer from the stack, provided that there were at least one buffer in the stack. If there were no buffers in the stack, it will remove all strings in the stack, i.e. remove the zeroth buffer.

If the parameter *number* was specified, and the stack contains a buffer with an assigned number equal to *number*, then that buffer itself, and all strings and buffers above it on the stack will be removed; but no strings or buffers below the numbered buffer will be touched. If *number* refers to a buffer that does not exist in the stack; no strings or buffers in the stack is touched.

As an extra extension, in Regina the DROPBUF() built-in function can be given a non-positive integer as parameter. If the name is negative then it will convert that number to its absolute value, and remove that many buffers, counted from the top. This is functionally equivalent to repeating DROPBUF() without parameters for so many times as the absolute value of the negative number specifies. Note that using -0 as parameter is equivalent to removing all strings and buffers in the stack, since -0 is equivalent to normal 0. The number is converted during evaluation of parameters prior to the call to the DROPBUF() routine, so the sing is lost.

The value returned from this function is the number of buffers left on the stack after the buffers to be deleted have been removed. Obviously, this will be a non-negative integer. This too, deviates from the behavior of the DROPBUF command under CMS, where zero is always returned.

DROPBUF(3) 2 remove buffer 3 and 4

DROPBUF(4) 0 no buffers on the stack
DROPBUF() 4 if there where 5 buffers

MAKEBUF()

Creates a new buffer on the stack, at the current top of the stack. Each new buffer will be assigned a number; the first buffer being assigned the number `1`. A new buffer will be assigned a number which is one higher than the currently highest number of any buffer on the stack. In practice, this means that the buffers are numbered, with the bottom-most having the number `1` and the topmost having a number which value is identical to the number of buffers currently in the stack.

The value returned from this function is the number assigned to the newly created buffer. The assigned number will be one more than the number of buffers already in the stack, so the numbers will be "recycled". Thus, the assigned numbers will not necessarily be in sequence.

MAKEBUF() 1 if no buffers existed
MAKEBUF() 6 if 5 buffers existed

Interfacing Rexx to other programs

This chapter describes an interface between a REXX interpreter and another program, typically written in C or another high level, compiled language. It is intended for application programmers who are implementing REXX support in their programs. It describes the interface known as the REXX SAA API.

1. Overview of functions in SAA

The functionality of the interface is divided into some main areas:

- Subcommand handlers
  which trap and handle a command to an external environment.
- External function handlers
  extend the REXX language with external functions
- Interpreting
  REXX scripts, either from a disk file, or from memory.
- Variable interface
  which makes it possible to access the variables in the interpreter, and allows operations like setting, fetching and dropping variables.
- System exits
  which are used to hook into certain key points in the interpreter while it executes a script.

In the following sections each of these areas are described in detail, and a number of brief but complete examples are given at the end of the chapter.

The description is of a highly technical nature, since it is assumed that the reader will be an application programmer seeking information about the interface. Therefore, much of the content is given as prototypes and C style datatype definitions. Although this format is cryptic for non-C programmers, it will convey exact, compact, and complete information to the intended readers. Also, the problems with
ambiguity and incompleteness that often accompany a descriptive prose text are avoided.

1.1 Include Files and Libraries

All the C code that uses the REXX application interface, must include a special header file that contains the necessary definitions. This file is called rexxsaa.h. Where you will find this file,
will depend on you system and which compiler you use.

Also, the interface part between the application and the REXX interpreter may be implemented as a library, which you link with the application using the functions described in this chapter. The name of
this library, and its location might differ from system to system. Under Unix, this library can be implemented as a static (libregina.a) or dynamic library (libregina.[so|sl]). Under other platforms Regina is also be implemented as a static or dynamic library.

1.2 Preprocessor Symbols

Including a header file ought to be enough; unfortunately, that is not so. Each of the domains of functionality listed above are defined in separate *sections'* in the rexxsaa.h header file. In order for
these to be made available, certain preprocessor symbols have to be set. For instance, you have to include the following definition:

```
#define INCL_RXSHV
```

in order to make available the definitions and datatypes concerning the variable pool interface. The various definitions that can be set are:

- INCL_RXSUBCOM
  Must be defined in order to get the prototypes, datatypes and symbols needed for the subcommand interface of the API.
- INCL_RXFUNC
  Must be defined in order to get the prototypes, datatypes and symbols needed for the external function interface of the API.
- INCL_RXSYSEXIT
  Must be defined in order to get the prototypes, datatypes, and symbols needed for the system exit functions
- INCL_RXSHV
  Must be set in order to get the prototypes, symbols and datatype definitions necessary to use the REXX variable pool.

1.3 Allocating and De-allocating Space

For several of the functions described in this chapter, the application calling them must allocate or de-allocate dynamic memory. Depending on the operating system, compiler and REXX interpreter, the method for these allocations and de-allocations might vary. Regina uses malloc() and free() in all these situations.

1.4 Data structures and data types

In this section, some data structures and datatypes relevant to the application interface to REXX are defined and described. The datatypes defined are:

- RXSTRING
  Holds a REXX string.
- RXSYSEXIT
  Holds a definition of a system exit handler. Used when starting a REXX script with RexxStart(), and when defining the system exit handlers.

The datatypes used in the SAA API are defined in rexxsaa.h. They are:

```
typedef char CHAR ;
typedef short SHORT ;
typedef long LONG ;
typedef char *PSZ ;
typedef CHAR *PCHAR ;
typedef SHORT *PSHORT ;
typedef LONG *PLONG ;
typedef unsigned char UCHAR ;
typedef unsigned short USHORT ;
typedef unsigned long ULONG ;
typedef USHORT *PUSHORT ;
typedef char *PCH ;
typedef unsigned char *PUCHAR ;
typedef ULONG APIRET;
typedef APIRET (APIENTRY *PFN)();
```

One other item needs mentioning; APIENTRY. This value is used to specify the linkage type on OS/2 and Win32 platforms. It is assumed that this value #defined by inclusion of compiler-specific header files in rexxsaa.h. Under Unix, this is #defined to nothing.

1.4.1 The RXSTRING structure

The SAA API interface uses *Rexx string* which are stored in the structure RXSTRING. There is also a datatype PRXSTRING, which is a pointer to RXSTRING. Their definitions are:

```
typedef struct {
    unsigned char *strptr ;    /* Pointer to string contents */
    unsigned long strlength ;  /* Length of string */
} RXSTRING ;

typedef RXSTRING *PRXSTRING ;
```

The strptr field is a pointer to an array of characters making up the contents of the *Rexx string*', while strlength holds the number of characters in that array.

Unfortunately, there are some inconsistencies in naming of various special kinds of strings. In REXX (TRL), a ''*null string*" is a string that has zero length. On the other hand, the SAA API operates with two kinds of special strings: *null strings* and *zero length strings*. The latter is a string with zero length (equals null strings in REXX), while the former is a sort of *undefined* or *empty* string, which denotes a string without a value. The *null strings* of SAA API are used to denote unspecified values (e.g. a parameter left out in a subroutine call). In this chapter, when the terms *null strings* and *zero length strings* are italicized, they refer to the SAA API style meaning.

A number of macros are defined, which simplifies operations on RXSTRINGs for the programmer. In the list below, all parameters called x are of type RXSTRING.

- MAKERXSTRING(x,content,length)]
  The parameter content must be a pointer to char, while length is integer. The x parameter will be set to the contents and length supplied. The only operations are assignments; no new space is allocated and the contents of the string is not copied.
- RXNULLSTRING(x)]
  Returns true only if x is a *null string*.
  i.e. x.strptr is NULL.
- RXSTRLEN(x)]
  Returns the length of the string x as an unsigned long. Zero is returned both when x is a *null string* or a *zero length string*.
- RXSTRPTR(x)]
  Returns a pointer to the first character in the string x, or NULL if x is a *null string*. If x is a *zero length string*, and non-NULL pointer is returned.
- RXVALIDSTRING(x)]
  Returns true only if x is neither a *null string* nor a *zero length string*
  i.e. x must have non-empty contents.
- RXZEROLENSTRING(x)]
  Returns true only if x is a *zero length string*.
  i.e. x.strptr is non-NULL, and x.strlength is zero.

These definitions are most likely to be defined as preprocessor macros, so you should never *call* them with *parameters* having any side effects. Also note that at least MAKERXSTRING() is likely to be implemented as two statements, and might not work properly if following
e.g. an if statement. Check the actual definitions in the rexxsaa.h header file before using them in a fancy context.

One definition of these might be (don't rely on this to be the case with your implementation):

```
#define MAKERXSTRING(x,c,l) ((x).strptr=(c),(x).strlength=(l))
#define RXNULLSTRING(x)     (!(x).strptr)
#define RXSTRLEN(x)         ((x).strptr ? (x).strlength : 0UL)
#define RXSTRPTR(x)         ((x).strptr)
#define RXVALIDSTRING(x)    ((x).strptr && (x).strlength)
#define RXZEROLENSTRING(x)  ((x).strptr && !(x).strlength)
```

Note that these definitions of strings differ from the normal definition in C programs; where a string is an array of characters, and its length is implicitly given by a terminating ASCII NUL character. In the RXSTRING definition, a string can contain any character, including an ASCII NUL, and the length is explicitly given.

1.4.2 The RXSYSEXIT structure

This structure is used for defining which system exit handlers are to handle which system exits. The two relevant datatypes are defined as:

```
typedef struct {
    unsigned char *sysexit_name ;
    short sysexit_code ;
} RXSYSEXIT ;

typedef RXSYSEXIT *PRXSYSEXIT ;
```

In this structure, sysexit_name is a pointer to the ASCII NUL terminated string containing the name of a previously registered (and currently active) system exit handler. The sysexit_code field is main function code of a system exit.

The system exits are divided into main functions and sub-functions. An exit is defined to handle a main function, and must thus handle all the sub-functions for that main function. All the functions and sub-functions are listed in the description of the EXIT structure.

2. The Subcommand Handler Interface

This sections describes the subcommand handler interface, which enables the application to trap commands in a REXX script being executed and handle this commands itself.

2.1 What is a Subcommand Handler

A subcommand handler is a piece of code, that is called to handle a command to an external environment in REXX. It must be either a subroutine in the application that started the interpreter, or a subroutine in a dynamic link library. In any case, when the interpreter needs to execute a command to an external environment, it will call the subcommand handler, passing the command as a parameter.
Typically, an application will set up a subcommand handler before starting a REXX script. That way, it can trap and handle any command being executed during the course of the script.

Each subcommand handler handles one environment, which is referred to by a name. It seems to be undefined whether upper and lower case letters differ in the environment name, so you should assume they differ. Also, there might be an upper limit for the length of an environment name, and some letters may be illegal as part of an environment name.

Regina allows any letter in the environment name, except ASCII NUL; and sets no upper limit for the length of an environment name. However, for compatibility reasons, you should avoid *uncommon* letters and keep the length of the name fairly short.

The prototype of a subcommand handler function is:

```
APIRET APIENTRY handler(
    PRXSTRING command,
    PUSHORT flags,
    PRXSTRING returnstring
) ;
```

After registration, this function is called whenever the application is to handle a subcommand for a given environment. The value of the parameters are:

[command]
: The command string that is to be executed. This is the resulting string after the command expression has been evaluated in the REXX interpreter. It can not be empty, although it can be a *zero-length-string*.

[flags]

Points to an unsigned short which is to receive the status of the completion of the handler. This can be one of the following: RXSUBCOM_OK, RXSUBCOM_ERROR, or RXSUBCOM_FAILURE. The contents will be used to determine whether to raise any condition at return of the subcommand. Do not confuse it with the return value.

[returnstring]

Points to a RXSTRING which is to receive the return value from the subcommand. Passing the return value as a string makes it possible to return non-numeric return codes. As a special case, you might set returnstring.strptr to NULL, instead of specifying a return string of the ASCII representation of zero.

Note that it is not possible to return *nothing* in a subcommand, since this is interpreted as zero. Nor is it possible to return a numeric return code as such; you must convert it to ASCII representation before you return.

The returnstring string will provide a 256 byte array which the programmer might use if the return data is not longer that that. If that space is not sufficient, the handler can provide another area itself. In that case, the handler should not de-allocate the default area, and the new area should be allocated in a standard fashion.

2.2 The RexxRegisterSubcomExe() function

This function is used to register a subcommand handler with the interface. The subcommand handler must be a procedure located within the code of the application. After registration, the REXX interpreter can execute subcommands by calling the subcommand handler with parameters describing the subcommand.

The prototype for RexxRegisterSubcomExe() is:

```
APIRET APIENTRY RexxRegisterSubcomExe(
    PSZ EnvName,
    PFN EntryPoint,
    PUCHAR UserArea
) ;
```

All the parameters are input, and their significance are:

[EnvName]

Points to an ASCII NUL terminated character string which defines the name of the environment to be registered. This is the same name as the REXX interpreter uses with the ADDRESS clause in order to select an external environment.

[EntryPoint]

Points to the entrypoint of the subcommand handler routine for the environment to be registered. See the section on Subcommand Handlers for more information. There is an upper limit for the length of this name.

[UserArea]

Pointer to an 8 byte area of information that is to be associated with this environment. This pointer can be NULL if no such area is necessary.

The areas pointed to by EnvName and UserArea are copied to a private area in the interface, so the programmer may de-allocate or reuse the area used for these parameters after the call has returned.

The RexxRegisterSubcom() returns an unsigned long, which carries status information describing the outcome of the operation. The status will be one of the RXSUBCOM values:

[RXSUBCOM_OK]

The subcommand handler was successfully registered.

[RXSUBCOM_DUP]

The subcommand handler was successfully registered. There already existed another subcommand handler which was registered with RexxRegisterSubcomDll(), but this will be shadowed by the newly registered handler.

[RXSUBCOM_NOTREG]

Due to some error, the handler was not registered. Probably because a handler for EnvName was already defined at a previous call to RexxRegisterSubcomExe().

[RXSUBCOM_NOEMEM]

The handler was not registered, due to lack of memory.

[RXSUBCOM_BADTYPE]

Indicates that the handler was not registered, due to one or more of the parameters having invalid values.

2.3 The RexxRegisterSubcomDll() function

This function is used to set up a routine that is located in a module in a dynamic link library, as a subcommand handler. Some operating systems don't have dynamic linking, and thus cannot make use of this facility. The prototype of this function is:

```
APIRET APIENTRY RexxRegisterSubcomDll(
    PSZ EnvName,
    PSZ ModuleName,
    PFN EntryPoint,
    PUCHAR UserArea,
    ULONG DropAuth
) ;
```

This function is not yet supported by Regina.

2.4 The RexxDeregisterSubcom() function

This function is used to remove a particular environment from the list of registered environments. The prototype of the function is:

```
APIRET APIENTRY RexxDeregisterSubcom(
    PSZ EnvName,
    PSZ ModuleName
) ;
```

Both parameters are input values:

[EnvName]

Pointer to ASCII NUL terminated string, which represents the name of the environment to be removed.

[ModuleName]

Also an ASCII NUL terminated string, which points to the name of the module containing the subcommand handler of the environment to be deleted.

The list of defined environments is searched, and if an environment matching the one named by the first parameter are found, it is deleted.

The returned value from RexxDeregisterSubcom() can be one of:

[RXSUBCOM_OK]

The subcommand handler was successfully deleted.

[RXSUBCOM_NOTREG]

The subcommand handler was not found.

[RXSUBCOM_BADTYPE]

One or more of the parameters had illegal values, and the operation was not carried through.

Most systems that do have dynamic linking have no method for reclaiming the space used by dynamically linked routines. So, even if you were able to load a *dll*, there are no guarantees that you will be able to unload it.

2.5 The RexxQuerySubcom() function

This function retrieves information about a previously registered subcommand handler. The prototype of the function is:

```
APIRET APIENTRY RexxQuerySubcom(
    PSZ EnvName,
    PSZ ModuleName,
    PUSHORT Flag,
    PUCHAR UserWord
) ;
```

The significance of the parameters are:

[EnvName]
: Pointer to an ASCII NUL terminated character string, which names the subcommand handler about which information is to be returned.

[ModuleName]
: Pointer to an ASCII NUL terminated character string, which names a dynamic link library. Only the named library will be searched for the subcommand handler named by EnvName. This parameter must be NULL if all subcommand handlers are to be searched.

[Flag]
: Pointer to a short which is to receive the value RXSUBCOM_OK or RXSUBCOM_NOTREG. In fact, this is the same as the return value from the function.

[UserWord]
: Pointer to an area of 8 bytes. The *userarea* of the subcommand handler is copied to the area pointed to by UserWord. This parameter might be NULL if the data of the *userarea* is not needed.

The returned value from RexxQuerySubcom() can be one of:

[RXSUBCOM_OK]
: The subcommand handler was found, and the required information has been returned in the Flag and UserWord variables.

[RXSUBCOM_NOTREG]
: The subcommand handler was not found. The Flag variable will also be set to this value, and the UserWord variable is not changed.

[RXSUBCOM_BADTYPE]
: One or more of the parameters had illegal values, and the operation was not carried through.

3. The External Function Handler Interface

This sections describes the external function handler interface, which extends the language by enabling external functions to be written in a language other than REXX.

3.1 What is an External Function Handler

An external function handler is a piece of code, that is called to handle external functions and subroutine calls in REXX. It must be either a subroutine in the application that started the interpreter, or a subroutine in a dynamic link library. In any case, when the interpreter needs to execute a function registered as an external function, it will call the external function handler, passing the function name as a parameter.
All external functions written in a language other than REXX must be registered with the interpreter before starting a REXX script.

An external function handler can handle one or more functions. The handler can determine the function actually called by examiining one of the parameters passed to the handler and act accordingly.

The prototype of a subcommand handler function is:

```
APIRET APIENTRY handler(
    PSZ name,
    ULONG argc,
    PRXSTRING argv,
    PSZ queuename,
    PRXSTRING returnstring
) ;
```

After a function is registered with this function defined as the handler, this function is called whenever the application calls the function. The value of the parameters are:

[name]
: The function called.

[argc]
: The number of parameters passed to the function. Argv will contain argc RXSTRINGs.

[queuename]
: The name of the currently define data queue.

[returnstring]
: Points to a RXSTRING which is to receive the return value from the function. Passing the return value as a string makes it possible to return non-numeric return codes. As a special case, you might set returnstring.strptr to NULL, instead of specifying a return string of the ASCII representation of zero.

The returnstring string will provide a 256 byte array which the programmer might use if the return data is not longer that that. If that space is not sufficient, the handler can provide another area itself. In that case, the handler should not de-allocate the default area, and the new area should be allocated in a standard fashion. if the external function does not return a value, it should set returnstring to an empty RXSTRING. This will enable the interpreter to raise error 44; *Function did not return data*, if the external function is called as a function. If the external function is invoked via a CALL command, the interpreter drops the special variable RESULT.

The handler returns zero if the function completed successfully. When the handler returns a non-zero value, the interpreter will raise error 40; *Invalid call to routine*.

3.2 The RexxRegisterFunctionExe() function

This function is used to register an external function handler with the interface. The external function handler must be a procedure located within the code of the application. After registration, the REXX interpreter can execute external functions as if they were built-ins.

The prototype for RexxRegisterFunctionExe() is:

```
APIRET APIENTRY RexxRegisterFunctionExe(
    PSZ FuncName,
    PFN EntryPoint
) ;
```

All the parameters are input, and their significance are:

[FuncName]
: Points to an ASCII NUL terminated character string which defines the name of the external function to be registered. This is the same name as the REXX interpreter uses with a function call or via the CALL command.

[EntryPoint]
: Points to the entrypoint of the external function handler routine for the function to be registered. See the section on External Function Handlers for more information.

The area pointed to by FuncName is copied to a private area in the interface, so the programmer may de-allocate or reuse the area used for this parameter after the call has returned.

The RexxRegisterFunctionExe() returns an unsigned long, which carries status information describing the outcome of the operation. The status will be one of the RXFUNC values:

[RXFUNC_OK]
: The handler was successfully registered.

[RXFUNC_DUP]
: The handler was successfully registered. There already existed another external function handler which was registered with RexxRegisterFunctionExe(), but this will be shadowed by the newly registered handler.

[RXFUNC_NOEMEM]
: The handler was not registered, due to lack of memory.

3.3 The RexxRegisterFunctionDll() function

This function is used to set up an external function handler that is located in a module in a dynamic link library. Some operating systems don't have dynamic linking, and thus cannot make use of this facility. The prototype of this function is:

```
APIRET APIENTRY RexxRegisterFunctionDll(
    PSZ ExternalName,
    PSZ LibraryName,
    PSZ InternalName
) ;
```

All the parameters are input, and their significance are:

[ExternalName]
: Points to an ASCII NUL terminated character string which defines the name of the external function to be registered. This is the same name as the REXX interpreter uses with a function call or via the CALL command.

[LibraryName]
: Points to an ASCII NUL terminated character string which defines the name of the dynamic library. This string may require a directory specification.

[InternalName]
: Points to an ASCII NUL terminated character string which defines the name of the entrypoint within the dynamic library. On systems where the case of function names in dynamic libraries is relevant, this name **must** be specified in the same case as the function name within the dynamic library.

The areas pointed to by all parameters are copied to a private area in the interface, so the programmer may de-allocate or reuse the area used for these parameters after the call has returned.

The RexxRegisterFunctionDll() returns an unsigned long, which carries status information describing the outcome of the operation. The status will be one of the RXFUNC values:

[RXFUNC_OK]
: The handler was successfully registered.

[RXFUNC_DUP]
: The handler was successfully registered. There already existed another external function handler which was registered with RexxRegisterFunctionDll(), but this will be shadowed by the newly registered handler.

[RXFUNC_NOEMEM]
: The handler was not registered, due to lack of memory.

3.4 The RexxDeregisterFunction() function

This function is used to remove a particular external function handler from the list of registered external function handlers. The prototype of the function is:

```
APIRET APIENTRY RexxDeregisterFunction(
    PSZ FuncName
) ;
```

The parameter is an input values:

[FuncName]
: Points to an ASCII NUL terminated character string which defines the name of the external function to be registered. This is the same name as the REXX interpreter uses with a function call or via the CALL command.

The list of defined function handlers is searched, and if an environment matching the one named by the parameter are found, it is deleted. This call is used to de-register function handlers registered with either RexxRegisterFunctionExe() or RexxRegisterFunctionDll().

The returned value from RexxDeregisterFunction() can be one of:

[RXFUNC_OK]
: The handler was successfully deleted.

[RXFUNC_NOTREG]
: The handler was not found.

Most systems that do have dynamic linking have no method for reclaiming the space used by dynamically linked routines. So, even if you were able to load a *dll*, there are no guarantees that you will be able to unload it.

3.5 The RexxQueryFunction() function

This function retrieves the status of an external function handler. The prototype of the function is:

```
APIRET APIENTRY RexxQueryFunction(
    PSZ FuncName
) ;
```

The significance of the parameters is:

[FuncName]
: Points to an ASCII NUL terminated character string which defines the name of the external function to be registered. This is the same name as the REXX interpreter uses with a function call or via the CALL command.

The returned value from RexxQueryFunction() can be one of:

[RXFUNC_OK]
: The external function handler was found.

[RXFUNC_NOTREG]
: The handler was not found.

4. Executing REXX Code

This sections describes the RexxStart() function, which allows the application to startup the interpreter and make it interpret pieces of REXX code.

4.1 The RexxStart() function

This function is used to invoke the REXX interpreter in order to execute a piece of REXX code, which may be located on disk, as a pre-tokenized macro, or as ASCII source code in memory.

```
APIRET APIENTRY RexxStart(
    LONG ArgCount,
    PRXSTRING ArgList,
    PSZ ProgramName,
    PRXSTRING Instore,
    PSZ EnvName,
    LONG CallType,
    PRXSYSEXIT Exits,
    PUSHORT ReturnCode,
    PRXSTRING Result
) ;
```

Of these parameters, ReturnCode and Result are output-only, while Instore is both input and output. The rest of the parameters are input-only. The significance of the parameters are:

[ArgCount]
: The number of parameter strings given to the procedure. This is the number of defined REXX-strings pointed to by the ArgList parameter.

[ArgList]
: Pointer to an array of REXX-strings, constituting the parameters to this call to REXX. The size of this array is given by the parameter ArgCount. If ArgCount is greater than one, the first and last parameters are ArgList[0] and ArgList[ArgCount-1]. If ArgCount is 0, the value of ArgList is irrelevant.

  If the strptr of one of the elements in the array pointed to by ArgList is NULL, that means that this parameter is empty (i.e. unspecified, as opposed to a string of zero size).

[ProgName]
: An ASCII NUL terminated string, specifying the name of the REXX script to be executed. The value of Instore will determine whether this value is interpreted as the name of a (on-disk) script, or a pre-tokenized macro. If it refers to a filename, the syntax of the contents of this parameter depends on the operating system.

[Instore]
: Parameter used for storing tokenized REXX scripts. This parameter might either be NULL, else it will be a pointer to two RXSTRING structures, the first holding the ASCII version of a REXX program, the other holding the tokenized version of that program. See below for more information about how to use Instore.

[EnvName]
: Pointer to ASCII NUL terminated string naming the environment which is to be the initial current environment when the script is started. If this parameter is set to NULL, the filetype is used as the initial environment name. What the filetype is, may depend on your operating system, but in general it is everything after the last period '.' in the filename.

[CallType]
: A value describing whether the REXX interpreter is to be invoked in command, function or subroutine mode. Actually, this has little significance. The main difference is that in command mode, only one parameter string can be passed, and in function mode, a value must be returned. In addition, the mode chosen will affect the output of the PARSE SOURCE instruction in REXX.

  Three symbolic values of integral type are defined, which can be used for this parameter: RXCOMMAND, RXFUNCTION and RXSUBROUTINE.

[SysExists]
: A pointer to an array of exit handlers to be used. If no exit handlers are to be defined, NULL may be specified. Each element in the array defines one exit handler, and the element immediately following the last definition must have a sysexit_code set to RXENDLST.

[ReturnCode]

Pointer to a SHORT integer where the return code is stored, provided that the returned value is numeric, and within the range -(2**15) to 2**15-1. I don't know what happens to ReturnCode if either of these conditions is not satisfied. It probably becomes undefined, which means that it is totally useless since the program has to inspect the return string in order to determine whether ReturnCode is valid.

[Result]

Points to a REXX string into which the result string is written. The caller may or may not let the strptr field be supplied. If supplied (i.e. it is non-NULL), that area will be used, else a new area will be allocated. If the supplied area is used, its size is supposed to be given by the strlength field. If the size if not sufficient, a new area will be allocated, by some system dependent channel (i.e. malloc()), and the caller must see to that it is properly de-allocated (using free()).

Note that the ArgCount parameter need not be the same as the ARG() built-in function would return. Differences will occur if the last entries in ArgList are *null strings*.

The Instore parameter needs some special attention. It is used to directly or indirectly specify where to fetch the code to execute. The following *algorithm* is used to determine what to execute:

If Instore is NULL, then ProgName names the filename of an on-disk REXX script which it to be read and executed.

Else, if Instore is not NULL, the script is somewhere in memory, and no reading from disk is performed. If both Instore[0].strptr and Instore[1].strptr are NULL, then the script to execute is a pre-loaded macro which must have been loaded with a call to either RexxAddMacro() or RexxLoadMacroSpace(); and ProgName is the name of the macro to execute.

Else, if Instore[1].strptr is non-NULL, then Instore[1] contains the pre-tokenized image of a REXX script, and it is used for the execution.

Else, if Instore[0].strptr is non-NULL, then Instore[0]} contains the ASCII image of a REXX script, just as if the script had been read directly from the disk (i.e. including linefeeds and such). This image is passed to the interpreter, which tokenizes it, and stores the tokenized script in the Instore[1] string, and then proceeds to execute that script. Upon return, the Instore[1] will be set, and can later be used to re-execute the script within the same process, without the overhead of tokenizing.

The user is responsible for de-allocating any storage used by Instore[1]. Note that after tokenizing, the source code in Instore[0] is strictly speaking not needed anymore. It will only be consulted if the user calls the SOURCELINE() built-in function. It is not an error to use SOURCELINE() if the source is not present, but nullstrings and zero will be returned.

Regina does not currently return any tokenized data in Instore[1] that can be used in a later call to RexxStart, outside of the current process. What Regina returns in Instore[1], is an index into an in-memory tokenized version of the source code. Once the process that parsed the source has stopped, the tokenized code is lost.

The valid return values from RexxStart() are:

[Negative]

indicates that a syntax error occurred during interpretation. In general, you can expect the error value to have the same absolute value as the REXX syntax error (but opposite signs, of course).

[Zero]

indicates that the interpreter finished executing the script without errors.

[Positive]

indicates probably that some problem occurred, that made it impossible to execute the script, e.g. a bad parameter value. However, I can't find any references in the documentation which states which values it is supposed to return.

During the course of an execution of RexxStart(), subcommand handlers and exit handlers might be called. These may call any function in the application interface, including another invocation of RexxStart().

Often, the application programmer is interested in providing support simplifying the specification of filenames, like an environment variable search path or a default file type. The REXX interface does support a default file type: .CMD, but the user may not set this to anything else. Therefore, it is generally up to the application programmer to handle search paths, and also default file types (unless .CMD is OK).

If the initial environment name (EvnName) is NULL, then the initial environment during interpretation will be set equal to the file type of the script to execute. If the script does not have a file
type, it is probably set to some interpreter specific value.

5. Variable Pool Interface

This section describes the variable pool part of the application interface, which allows the application programmer to set, retrieve and drop variables in the REXX interpreter from the application program. It also allows access to other information.

The C preprocessor symbol INCL_RXSHV must be defined if the definitions for the variable pool interface are to be made available when rexxsaa.h is included.

5.1 Symbolic or Direct

First, let us define two terms, *symbolic* variable name and *direct* variable name, which are used in connection with the variable pool.

A symbolic variable name is the name of a variable, but it needs normalization and tail substitution before it names the real variable. The name foo.bar is a symbolic variable name, and it is transformed by normalization, to FOO.BAR, and then by tail substitution to FOO.42 (assuming that the current value of BAR is 42).

Normalization is the process of uppercasing all characters in the symbolic name; and tail substitution is the process of substituting each distinct simple symbol in the tail for its value.

On the other hand, a direct variable refers directly to the name of the variable. In a sense, it is a symbolic variable that has already been normalized and tail substituted. For instance, foo.bar is not a valid direct variable name, since lower case letters are not allowed in the variable stem. The direct variable FOO.42 is the same as the variable above. For simple variables, the only difference between direct and symbolic variable names is that lower case letters are allowed in symbolic names

Note that the two direct variable names FOO.bar and FOO.BAR refer to different variables, since upper and lower case letters differ in the tail. In fact, the tail of a compound direct variable may contain any character, including ASCII NUL. The stem part of a variable, and all simple variables can not contain any lower case letters.

As a remark, what would the direct variable FOO. refer to: the stem FOO. or the compound variable having stem FOO. and a nullstring as tail? Well, I suppose the former, since it is the more useful. Thus, the latter is inaccessible as a direct variable.

5.2 The SHVBLOCK structure

All requests to manipulate the REXX variable pool are controlled by a structure which is called SHVBLOCK, having the definition:

```
typedef struct shvnode {
    struct shvnode *shvnext ;    /* ptr to next in blk in chain */
    RXSTRING shvname ;           /* name of variable */
    RXSTRING shvvalue ;          /* value of variable */
    ULONG shvnamelen ;   /* length of shvname.strptr */
    ULONG shvvaluelen ;  /* length of shvvalue.strptr */
```

```
    UCHAR shvcode ;     /* operation code */
    UCHAR shvret ;      /* return code */
} SHVBLOCK ;

typedef SHVBLOCK *PSHVBLOCK ;
```

The fields shvnext and shvcode are purely input, while shvret is purely output. The rest of the fields might be input or output, depending on the requested operation, and the value of the fields. The significance of each field is:

[shvnext]
: One call to RexxVariablePool() may sequentially process many requests. The shvnext field links one request to the next in line. The last request must have set shvnext to NULL. The requests are handled individually and thus, calling RexxVariablePool() with several requests is equivalent to making one call to RexxVariablePool() for each request.

[shvname]
: Contains the name of the variable to operate on, as a RXSTRING. This field is only relevant for some requests, and its use may differ.

[shvvalue]
: Contains the value of the variable to operate on as a RXSTRING. Like shvname, this might not be relevant for all types of requests.

[shvnamelen]
: The length of the array that shvname.strptr points to. This field holds the maximum possible number of characters in shvname.strptr. While shvname.strlength holds the number of characters that are actually in use (i.e. defined).

[shvvaluelen]
: The length of the array that shvvalue.strptr points to. Relates to shvvalue, like shvnamelen relates to shvname.

[shvcode]
: The code of operation; decides what type of request to perform. A list of all the available requests is given below.

[shvret]
: A return code describing the outcome of the request. This code is a bit special. The lower seven bits are flags which are set depending on whether some condition is met or not. Values above 127 are not used in this field.

There is a difference between shvnamelen and shvname.strlength. The former is the total length of the array of characters pointed to by shvname.strptr (if set). While the latter is the number of these characters that are actually in use. When a SHVBLOCK is used to return data from RexxVariablePool(), and a pre-allocated string space has been supplied, both these will be used; shvname.strlength will be set to the length of the data returned, while shvnamelen is never changed, only read to find the maximum number of characters that shvname can hold.

Even though shvnamelen is not really needed when shvname is used for input, it is wise to set it to its proper value (or at least set it to the same as shvname.strlength). The same applies for shvvalue and shvvaluelen.

The field shvcode can take one of the following symbolic values:

[RXSHV_DROPV]
: The variable named by the direct variable name shvname is dropped (i.e. becomes undefined). The fields shvvalue and shvvaluelen do not matter.

[RXSHV_EXIT]
: This is used to set the return value for an external function or exit handler.

[RXSHV_FETCH]
: The value of the variable named by the direct variable name shvname is retrieved and stored in shvvalue. If shvvalue.strptr is NULL, the interpreter will allocate sufficient space to store the value (but it is the responsibility of the application programmer to release that space). Else, the value will be stored in the area allocated for shvvalue, and shvvaluelen is taken to be the maximum size of that area.

[RXSHV_NEXTV]
: This code is used to retrieve the names and values of all variables at the current procedure level; i.e. excluding variables shadowed by PROCEDURE. The name and value of each variable are retrieved

simultaneously into shvname and shvvalue, respectively.
Successive requests for RXSHV_NEXTV will traverse the interpreter's internal data structure for storing variables, and return a new pair of variable name and value for each request. Each variable that is visible in the current scope, is returned once and only once, but the order is non-deterministic. When all available variables in the REXX interpreter have already been retrieved, subsequent RXSHV_NEXTV requests will set the flag RXSHV_LVAR in the shvret field. There are a few restrictions. The traversal will be reset whenever the interpreter resumes execution, so an incomplete traversal can not be continued in a later external function, exit handler, or subcommand handler. Also, any set, fetch or drop operation will reset the traversal. These restrictions have been added to ensure that the variable pool is static throughout one traversal.

[RXSHV_PRIV]
Retrieves some piece of information from the interpreter, other than a variable value, based on the value of the shvname field. The value is stored in shvvalue as for a *normal* fetch. A list of possible names is shown below.

[RXSHV_SET]
The variable named by the direct variable name shvname is set to the value given by shvvalue.

[RXSHV_SYFET]
Like RXSHV_FETCH, except that shvname is a symbolic variable name.

[RXSHV_SYDRO]
Like RXSHV_DROPV, except that shvname is a symbolic variable name.

[RXSHV_SYSET]
Like RXSHV_SET, except that shvname is a symbolic variable name.

One type of request that needs some special attention is the RXSHV_PRIV, which retrieves a kind of *meta-variable*. Depending on the value of shvname, it returns a value in shvvalue describing some aspect of the interpreter. For RXSHV_PRIV the possible values for shvname are:

[PARM]
Returns the ASCII representation of the number of parameters to the currently active REXX procedure. This may not be the same value as the built-in function ARG() returns, but is the number ArgCount in RexxStart(). The two might differ if a routine was called with trailing omitted parameters.

[PARM.n]
The n must be a positive integer; and the value returned will be the n'th parameter at the current procedure level. This is not completely equivalent to the information that the built-in function ARG() returns. For parameters where ARG() would return the state omitted, the returned value is a *null string*, while for parameters where ARG() would return the state *existing*, the return value will be the parameter string (which may be a *zero length string*.

[QUENAME]
The name of the currently active external data queue. This feature has not yet been implemented in Regina, which always return *default*.

[SOURCE]
Returns the same string that is used in the PARSE SOURCE clause in REXX, at the current procedure level of interpretation.

[VERSION]
Returns the same string that is used in the PARSE VERSION clause in REXX.

The value returned by a variable pool request is a bit uncommon. A return value is computed for each request, and stored in the shvret field. This is a one-byte field, of which the most significant bit is never set. A symbolic value RXSHV_OK is defined as the value zero, and the shvret field will be equal to this name if none if the flags listed below is set. The symbolic value for these flags are:

[RXSHV_BADF]
The shvcode of this request contained a bad function code.

[RXSHV_BADN]
The shvname field contained a string that is not valid in this context. What exactly is a valid value depends on whether the operation is a private, a symbolic variable, or direct variable operation.

[RXSHV_LVAR]
Set if and only if the request was RXSHV_NETXV, and all available variables have already been retrieved by earlier requests.

[RXSHV_MEMFL]

There was not enough memory to complete this request.

[RXSHV_NEWV]
Set if and only if the referenced variable did not previously have a value. It can be returned for any set, fetch or drop operation.

[RXSHV_TRUNC]
Set if the retrieved value was truncated when it was copied into either the shvname or shvvalue fields. See below.

These flags are directly suitable for logical OR, without shifting, e.g. to check for truncation and no variables left, you can do something like:

```
if (req->shvret & (RXSHV_TRUNC | RXSHV_LVAR))
    printf("Truncation or no vars left\n") ;
```

RXSHV_TRUNC can only occur when the interface is storing a retrieved value in a SHVBLOCK, and the pre-allocated space is present, but not sufficiently large. As described for RXSHV_FETCH, the interpreter will allocate enough space if shvvalue.strptr is NULL, and then RXSHV_TRUNC will never be set. Else the space supplied by shvvalue.strptr is used, and shvvaluelen is taken as the maximum length of shvvalue, and truncation will occur if the supplied space is too small.

Some implementations will consider SHV_MEMFL to be so severe as to skip the rest of the operations in a chain of requests. In order to write compatible software, you should never assume that requests following in a chain after a request that returned SHV_MEMFL have been performed.

RXSHV_BADN is returned if the supplied shvname contains a value that is not legal in this context. For the symbolic set, fetch and drop operations, that means a symbol that is a legal variable name; both upper and lower case letters are allowed. For the direct set, fetch and drop operations, that means a variable name after normalization and tail substitution is not a legal variable name. For RXSHV_PRIV, it must be one of the values listed above.

There is a small subtlety in the above description. TRL states that when a REXX assignment assigns a value to a stem variable, all possible variables having that stem are assigned a new value (independent of whether they had an explicit value before). So, strictly speaking, if a stem is set, then a RXSHV_NETV sequence should return an (almost) infinite sequence of compound variables for that stem. Of course, that is completely useless, so you can assume that only compound variables of that stem given an explicit value after the stem was assigned a value will be returned by RXSHV_NEXTV. However, because of that subtlety, the variables returned by RXSHV_NEXTV for compound variables might not be representative for the state of the variables.

e.g. what would a sequence of RXSHV_NEXT requests return after the following REXX code ?:

```
foo. = 'bar'
drop foo.bar
```

The second statement here, might not change the returned values! After the first statement, only the stem foo. would probably have been returned, and so also if all variables were fetched after the second statement.

5.3 Regina Notes for the Variable Pool

Due to the subtleties described at the end of the previous subsection, some notes on how Regina handles RXSHV_NEXTV requests for compound variables are in order. The following rules applies:

- Both the stem variable FOO. and the compound variable having FOO. as stem and a nullstring as tail, are returned with the name of FOO.. In this situation, a sequence of RXSHV_NEXTV requests may seem to return values for the same variable twice. This is unfortunate, but it seems to be the only way. In any case, you'll have to perform the RXSHV_SYFET in order to determine which is which.
- If a stem variable has not been assigned a value, its compound variables are only returned if they have been assigned an explicit value. i.e. compound variables for that stem that have either never been assigned a value, or have been dropped, will not be reported by RXSHV_NEXTV. There is nothing strange about this.
- If a stem variable has been assigned a value, then its compound variables will be reported in two cases: Firstly, the compound variables having explicitly been assigned a value afterwards. Secondly, the compound variables

which have been dropped afterwards, which are reported to have their initial value, and the flag RXSHV_NEWV is set in shvret.

It may sound a bit stupid that unset variables are listed when the request is to list all variables which have been set, but that is about the best I can do, if I am to stay within the standard definition and return a complete and exact status of the variable pool.

If the return code from RexxVariablePool() is less than 128, Regina is guaranteed to have tried to process all requests in the chain. If the return code is above 127, some requests may not have been processed. Actually, the number 127 (or 128) is a bit inconvenient, since it will be an problem for later expansion of the standard. A much better approach would be to have a preprocessor symbol (say, RXSHV_FATAL, and if the return code from the RexxVariablePool() function was larger than that, it would be a *direct* error code, and not a *composite* error code built from the shvret fields of the requests. The RXSHV_FATAL would then have to be the addition of all the atomic composite error codes.

(Warning: author mounting the soapbox.)

The *right* way to fix this, is to let the function RexxVariablePool() set another flag in shvret (e.g. named RXSHV_STEM) during RXSHV_NEXTV if and only if the value returned is a stem variable. That way, the application programmer would be able to differ between stem variables and compound variable with a null string tail.

To handle the other problem with compound variables and RXSHV_NEXTV, I would have liked to return a *null string* in shvvalue if and only if the variable is a compound variable having its initial value, and the stem of that compound variable has been assigned a value. Then, the value of the compound variable is equal to its name, and is available in the shvname field.

I'd also like to see that the shvret value contained other information concerning the variables, e.g. whether the variable was exposed at the current procedure level. Of course, Regina does not contain any of these extra, non-standard features.

(Author is dismounting the soapbox.)

When Regina is returning variables with RXSHV_NEXTV, the variables are returned in the order in which they occur in the open hashtable in the interpreter. i.e. the order in which variables belonging to different bins are returned is consistent, but the order in which variables hashed to the same bin are returned, is non-deterministic. Note that all compound variables belonging to the same stem are returned in one sequence.

5.4 The RexxVariablePool() function

This function is used to process a sequence of variable requests, and process them sequentially. The prototype of this function is:

```
APIRET APIENTRY ULONG RexxVariablePool(
    SHVBLOCK *Request
) ;
```

Its only parameter is a pointer to a SHVBLOCK structure, which may be the first of the linked list. The function performs the operation specified in each block. If an error should occur, the current request is terminated, and the function moves on to the next request in the chain.

The result value is a bit peculiar. If the returned value is less than 128, it is calculated by logically OR'ing the returned shvret field of all the requests in the chain. That way, you can easily check whether any of the requests was e.g. skipped because of lack of memory. To determine which request, you have to iterate through the list.

If the result value is higher than 127, it signifies an error. If any of these values are set, you can not assume that any of the requests have been processed. The following symbolic name gives its meaning.

[RXSHV_NOAVL]

Means that the interface is not available for this request. This might occur if the interface was not able to start the interpreter, or if an operation requested a variable when the interpreter is not currently executing any script (i.e. idle and waiting for a script to execute).

6. The System Exit Handler Interface

The exit handlers provide a mechanism for governing important aspects of the REXX interpreter from the application: It can trap situations like the interpreter writing out text, and then handle them itself, e.g. by displaying the text in a special window. You can regard system exits as a sort of *hooks*.

6.1 The System Exit Handler

Just like the subcommand handler, the system exit handler is a routine supplied by the application, and is called by the interpreter when certain situations occur. These situations are described in detail later. For the examples below, we will use the output from SAY as an example.

If a system exit handler is enabled for the SAY instruction, it will be called with a parameter describing the text that is to be written out. The system exit handler can choose to handle the situation (e.g. by writing the text itself), or it can ignore it and let the interpreter perform the output. The return code from the system exit tells the interpreter whether a system exit handled the situation or not.

A system exit handler must be a routine defined according to the prototype:

```
LONG APIENTRY my_exit_handler(
    LONG ExitNumber,
    LONG Subfunction,
    PEXIT ParmBlock
) ;
```

In this prototype, the type PEXIT is a pointer to a parameter block containing all the parameters necessary to handle the situation. The actual definition of this parameter block will vary, and is described in detail in the list of each system exit.

The exits are defined in a two-level hierarchy. The ExitNumber defines the main function for a system exit, while the Subfunction defines the subfunction within that main function. e.g. for SAY, the main function will be RXSIO (the system exit for standard I/O) and the subfunction will be RXSIOSAY. The RXSIO main function has other sub-functions for handling trace output, interactive trace input, and PULL input from standard input.

The value returned from the system exit handler must be one of the following symbolic values:

[RXEXIT_HANDLED]
: Signals that the system exit handler took care of the situation, and that the interpreter should not proceed to do the default action. For the SAY instruction, this means that the interpreter will not print out anything.

[RXEXIT_NOT_HANDLED]
: Signals that the system exit handler did not take care of the situation, and the interpreter will proceed to perform the default action. For the SAY instruction, this means that it must print out the argument to SAY.

[RXEXIT_RAISE_ERROR]
: Signals that the interpreter's default action for this situation should not be performed, but instead a SYNTAX condition should be raised. Don't get confused by the name, it is not the ERROR condition, but the SYNTAX condition is raised, using the syntax error *Failure in system service* (normally numbered 48).

In addition to returning information as the numeric return value, information may also be returned by setting variables in the parameter block. For instance, if the system exit is to handle interactive trace input, that is how it will supply the interpreter with the input string.

It is a good and disciplined practice to let your exit handlers start by verifying the ExitNumber and Subfunction codes, and immediately return RXEXIT_NOT_HANDLED if it does not recognize both of them. That way, your application will be upwards compatible with future interpreters which might have more sub-functions for any given main function.

6.2 List of System Exit Handlers

6.2.1 RXFNC — The External Function Exit Handler

The RXFNC system exit handler provides hooks for external functions. It has only one subfunction; RXFNCCAL, which allows an application program to intervene and handle any external function or subroutine.

Do not confuse this exit handler with the external function routines which allow you to define new REXX, semi-built-in functions. The exit handler is called for all invocations of external routines, and can be called for function names which you were unaware of.

The parameter ParmBlock for RXFNCCAL is defined as:

```
typedef struct {
    typedef struct {
      unsigned int rxfferr:1 ;
      unsigned int rxffnfnd1 ;
      unsigned int rxffsub: 1;
    } rxfnc_flags ;
    unsigned char *rxfnc_address ;
    unsigned short rxfnc_addressl ;
    unsigned char *rxfnc_que ;
    unsigned short rxfnc_quel ;
    unsigned short rxfnc_argc;
    RXSTRING *rxfnc_argv ;
    RXSTRING rxfnc_retc ;
} RXFNCCAL_PARM ;
```

The significance of each variable is:

[rxfnc_flags.rxfferr]
: Is an output parameter that is set on return in order to inform the interpreter that the function or subroutine was incorrectly called, and thus the SYNTAX condition should be raised.

[rxfnc_flags.rxffnfnd]
: Is an output parameter that tells the interpreter that the function was not found. Note the inconsistency: it is only effective if tthe exit handler returns RXEXIT_HANDLED, which looks like a logic contradiction to setting the not-found flag.

[rxfnc_flags.rxffsub]
: Is an input parameter that tells the exit handler whether it was called for a function or subroutine call. If set, the call being handled is a subroutine call and returning a value is optional; else it was called for a function, and must return a value in rxfnc_retc if RXEXIT_HANDLED is to be returned.

[rxfnc_name]
: Is a pointer to the name of the function or subroutine to be handled, stored as a character array. This is an input parameter, and its length is given by the rxfnc_namel parameter.

[rxfnc_namel]
: Holds the length of rxfnc_name. Note that the last character is the letter *ell*, not the number one.

[rxfnc_que]
: Points to a character array holding the name of the currently active queue. This is an input parameter. The length of this name is given by the rxfnc_quel field.

[rxfnc_que1]
: Holds the length of rxfnc_que. Note that the last character is the letter *ell*, not the number one.

[rxfnc_argc]
: Is the number of arguments passed to the function or subroutine. It defines the size of the array pointed to by the rxfnc_argv field.

[rxfnc_argv]
Points to an array holding the parameters for the routines. The size of this array is given by the rxfnc_argc field. If rxfnc_argc is zero, the value of rxfnc_argv is undefined.

[rxfnc_retc]
Holds an RXSTRING structure suitable for storing the return value of the handler. It is the responsibility of ht ehandler to allocate space for the contents of this string (i.e. the array pointed to by the rxfnc_retc.strptr).

6.2.2 RXCMD — The Subcommand Exit Handler

The main function code for this exit handler is given by the symbolic name RXCMD. It is called whenever the interpreter is about to call a subcommand, i.e. a command to an external environment. It has only one subfunction: RXCMDHST.

The ParmBlock parameter for this subfunction has the following definition:

```
typedef struct {
    typedef struct {
      unsigned int rxfcfail:1 ;
      unsigned int rxfcerr:1 ;
    } rxcmd_flags ;
    unsigned char *rxcmd_address ;
    unsigned short rxcmd_addressl ;
    unsigned char *rxcmd_dll ;
    unsigned short rxcmd_dll_len ;
    RXSTRING rxcmd_command ;
    RXSTRING rxcmd_retc ;
} RXCMDHST_PARM ;
```

The significance of each variable is:

[rxcmd_flags.rxfcfail]
If this flag is set, the interpreter will raise a FAILURE condition at the return of the exit handler.

[rxcmd_flags.rxfcerr]
Like the previous, but the ERROR condition is raised instead.

[rxcmd_address]
Points to a character array containing the name of the environment to which the command normally would be sent.

[rxcmd_addressl]
Holds the length of rxcmd_address. Note that the last character is the letter *ell*, not the number one.

[rxcmd_dll]
Defines the name for the DLL which is to handle the command. I'm not sure what this entry is used for. It is not currently in use for Regina.

[rxcmd_dll_len]
Holds the length of rxcmd_dll. If this length is set to zero, the subcommand handler for this environment is not a DLL, but an EXE handler.

[rxcmd_command]
Holds the command string to be executed, including command name and parameters.

[rxcmd_retc]
Set by the exit handler to the string which is to be considered the return code from the command. It is assigned to the special variable RC at return from the exit handler. The user is responsible for allocating space for this variable. I have no clear idea what happens if rxcmd_retc.strptr is set to NULL; it might set RC to zero, to the null string, or even drop it.

It seems that this exit handler is capable of raising both the ERROR and the FAILURE conditions simultaneously. I don't know whether that is legal, or whether only the FAILURE condition is raised, since the ERROR condition is a sort of *subset* of FAILURE.

Note that the return fields of the parameter block are only relevant if the value RXEXIT_HANDLED was returned. This applies to the rxcmd_flags and rxcmd_retc fields of the structure.

6.2.3 RXMSQ — The External Data Queue Exit Handler

The external data queue exit handler is used as a hook for operations manipulating the external data queue (or the stack). Unfortunately, the stack is a borderline case of what is relevant to the REXX SAA API. Operations like putting something on, retrieving a string from, obtaining the size, etc. of the stack is not part of the SAA API. However, some of this functionality is seemingly here; but not all. For instance for the RXMSQPLL subfunction, SAA API is called by the interpreter before the interpreter calls whatever system-specific call is available for retrieving a string from the stack.
Thus the SAA API can be used by an application to provide the interpreter with a fake stack, but it is not a suitable means for the application itself to manipulate the *real* stack.
The RXMSG exit has four subfunctions:

[RXMSQPLL]
This is called before a line is retrieved from the stack and the application may itself provide the interpreter with an alternative line. On entry, the third parameter points to a structure having the following definition:

```
typedef struct {
    RXSTRING rxmsq_retc;
} RXMSQPLL_PARM;
```

The rxmsq_retc field holds the string to be retrieved from the stack. Note that it is an output parameter, so its value on entry is undefined.

[RXMSQPSH]
This is called before the interpreter puts a line on the stack, and it may grab the line itself, and thus prevent the interpreter from putting the line on the stack. Note that this exit handles both pushing and queuing. The third parameter is:

```
typedef struct {
    struct {
        unsigned rxfmlifo: 1;
    } rxmsq_flags;
    RXSTRING rxmsq_value;
} RXMSQPSH_PARM;
```

Here the field rxmsq_value holds the string to be put on the stack. Whether the string is to be pushed or queued is determined by the boolean value rxmsq_flags.rxmlfifo, which is TRUE if the string is to be pushed.
All values are input values. What happens if you change them is not defined in the SAA API. Some implementations may let you modify the contents of rxmsq_value and return RXEXIT_NOT_HANDLED and the string push by the interpreter contains the modified string. However, you should not rely on this since it is highly incompatible. You may not de-allocate rxmsq_value.

[RXMSQSIZ]
this is called before the interpreter tries to determine the size of the stack, and it may present an alternative size to the interpreter. The third parameter is:

```
typedef struct {
    ULONG rxmsq_size;
} RXMSQSIZ_PARM;
```

The field rxmsq_size can be set to the number the application wants the QUEUED() function to return. Note that this parameter is undefined on entry, so it cannot be used to retrieve the number of lines on the stack.

[RXSQNAM]
This is called before the interpreter tries to retrieve the name of the current stack, and it may present the interpreter with an alternative name. Note that this functionality is part of SAA but not TRL; it supports the **Get** option of the RXQUEUE() built-in function. Note that there are no other exits supporting the other options of RXQUEUE(). The third parameter for this exit is:

```
typedef struct {
    RXSTRING rxmsq_name;
} RXMSQNAM_PARM;
```

As with RXSQMSIZ, the field rxmsq_name can be set to the name which the application wants to return to the interpreter as the name of the current stack. Note that this is an output-only parameter; its value on input is undefined, and in particular is not the name of the real stack.

Note that this area is troublesome. In TRL, external data queues are not defined as part of the language, while in SAA it is. Thus, TRL-compliant interpreters are likely to implement stacks in various ways that may not be compatible with the SAA.

6.2.4 RXSIO — The Standard I/O Exit Handler

The main code for this exit handler has the symbolic value RXSIO. There are four sub-functions:

[RXSIODTR]
: Called whenever the interpreter needs to read a line from the user during interactive tracing. Note the difference between this subfunction and RXSIOTRD.

[RXSIOSAY]
: Called whenever the interpreter tries to write something to standard output in a SAY instruction, even a SAY instruction without a parameter.

[RXSIOTRC]
: Called whenever the interpreter tries to write out debugging information, e.g. during tracing, as a trace back, or as a syntax error message.

[RXSIOTRD]
: Called whenever the interpreter need to read from the standard input stream during a PULL or PARSE PULL instruction. Note that it will not be called if there is sufficient data on the stack to satisfy the operation.

Note that these function are only called for the exact situations that are listed above. e.g. the RXSIOSAY is not called during a call to the REXX built-in function LINEOUT() that writes to the default output stream. TRL says that SAY is identical to calling LINEOUT() for the standard output stream, but SAA API still manages to see the difference between stem variables and compound variables with a ‘‘*zero-length-string*’’ tail. Please bear with this inconsistency.

Depending on the subfunction, the ParmBlock parameter will have four only slightly different definitions. It is kind of frustrating that the ParmBlock takes so many different datatypes, but it can be handled easily using unions, see a later section. The definitions are:

```
typedef struct {
    RXSTRING rxsiodtr_retc ;  /* the interactive trace input */
} RXSIODTR_PARM ;

typedef struct {
    RXSTRING rxsio_string ;   /* the SAY line to write out */
} RXSIOSAY_PARM ;

typedef struct {
    RXSTRING rxsio_string ;   /* the debug line to write out */
} RXSIOTRC_PARM ;

typedef struct {
    RXSTRING rxsiotrd_retc ;  /* the line to read in */
} RXSIOTRD_PARM ;
```

In all of these, the RXSTRING structure either holds the value to be written out (for RXSIOSAY and RXSIOTRC), or the value to be used instead of reading standard input stream (for RXSIOTRD and RXSIODTR). Note that the values set by RXSIOTRD and RXSIODTR are ignored if the exit handler does not return the value RXEXIT_HANDLED.

Any end-of-line marker are stripped off the strings in this context. If the exit handler writes out the string during RXSIOSAY or RXSIOTRC, it must supply any end-of-line action itself. Similarly, the interpreter does not expect a end-of-line marker in the data returned from RXSIODTR and RXSIOTRD.

The space used to store the return data for the RXSIODTR and RXSIOTRD sub-functions, must be provided by the exit handler itself, and the space is not de-allocated by the interpreter. The space can be reused by the application at any later time. The space allocated to hold the data given by the RXSIOSAY and RXSIOTRC sub-functions, will be allocated by the interpreter, and must neither be de-allocated by the exit handler, nor used after the exit handler has terminated.

6.2.5 RXHLT — The Halt Condition Exit Handler

Note: Because the RXHLT exit handler is called after every REXX instruction, enabling this exit slows REXX program execution.

The main code for this exit handler has the symbolic value RXHLT. There are two sub-functions:

[RXHLTTST]

Called whenever the interpreter polls externally raised HALT conditions; ie after every REXX instruction.

The definition of the ParmBlock is:

```
typedef struct {
    unsigned rxfhhlt : 1 ;
} RXHLTTST_PARM ;
```

The rxfhhlt parameter is set to the state of the HALT condition in the interpreter; either TRUE or FALSE.

[RXHLTCLR]

Called to acknowledge processing of the HALT condition when the interpreter has recognized and raised a HALT condition.

6.2.6 RXTRC — The Trace Status Exit Handler

6.2.7 RXINI — The Initialization Exit Handler

RXTER and this exit handler are a bit different from the others. RXINI provides the application programmer with a method of getting control before the execution of the script starts. Its main purpose is to enable manipulation of the variable pool in order to set up certain variables before the script starts, or set the trace mode.

It has only one subfunction, RXINIEXT, called once during each call to RexxStart(): just before the first REXX statement is interpreted. Variable manipulations performed during this exit will have effect when the script starts.

As there is no information to be communicated during this exit, the value of ParmBlock is undefined. It makes no difference whether you return RXEXIT_HANDLED or RXEXIT_NOT_HANDLED, since there is no situation to handle.

6.2.8 RXTER — The Termination Exit Handler

This exit resembles RXINI. Its sole subfunction is RXTEREXT, which is called once, just after the last statement of the REXX script has been interpreted. The state of all variables are intact during this call; so it can be used to retrieve the values of the variables at the exit of a script. (In fact, that is the whole purpose of this exit handler.)

Like RXINI, there is no information to be communicated during the exit, so ParamBlock is undefined in this call. And also like RXINI, it is more of a hook than an exit handler, so it does not matter whether you return RXEXIT_HANDLED or RXEXIT_NOT_HANDLED.

Implementation Limits

This chapter lists the implementation limits required by the REXX standard. All implementations are supposed to support at least these limits.

1. Why Use Limits?

Why use implementation limits at all? Often, a program (ab)uses a feature in a language to an extent that the implementor did not foresee. Suppose an implementor decides that variable names can not be longer than 64 bytes. Sooner or later, a programmer gets the idea of using very long variable names to encode special information in the name; maybe as the output of a machine generated program. The result will be a program that works only for some interpreters or only for some problems.

By introducing implementation limits, REXX tells the implementors to what extent a implementation is required to support certain features, and simultaneously it tells the programmers how much functionality they can assume is present.

Note that these limited are required minimums for what an implementation must allow. An interpreter is not supposed to enforce these limits unless there is a good reason to.

2. What Limits to Choose?

A limit must not be perceived as an absolute limit, the implementor is free to increase the limit. To some extent, the implementor may also decrease the limit, in which case this must be properly documented as a non-standard feature. Also, the reason for this should be noted in the documentation.

Many interpreters are likely to have “memory” as an implementation limit, meaning that they will allow any size as long as there is enough memory left. Actually, this is equivalent to no limit, since running out of memory is an error with limit enforcing interpreters as well. Some interpreters let the user set the limits, often controlled through the `OPTIONS` instruction.

For computers, limit choices are likely to be powers of two, like 256, 1024, 8192, etc. However, the REXX language takes the side of the user, and defines the limits in units which looks as more “sensible” to computer non-experts: most of the limits in REXX are numbers like 250, 500, 1000, etc.

3. Required Limits

These are the implementation minimums defined by REXX:

`[Binary strings]`
: Must be able to hold at least 50 characters after packing. That means that the unpacked size might be at least 400 characters, plus embedded white space.

`[Elapse time clock]`
: Must be able to run for at least 10**10-1 seconds, which is approximately 31.6 years. In general, this is really a big overkill, since virtually no program will run for a such a period. Actually, few computers will be operational for such a period.

[Hexadecimal strings]
Must be able to hold at least 50 characters after packing. This means that the unpacked size might be at least 100 characters, plus embedded white space.

[Literal strings]
Must be able to hold at least 100 characters. Note that a double occurrence of the quote character (the same character used to delimit the string) in a literal string counts as a single character. In particular, it does not count as two, nor does it start a new string.

[Nesting of comments]
Must be possible to in at least 10 levels. What happens then is not really defined. Maybe one of the syntax errors is issued, but none is obvious for this use. Another, more dangerous way of handling this situation would be to ignore new start-of-comments designators when on level 10. This could, under certain circumstances, lead to running of code that is actually commented out. However, most interpreter are likely to support nesting of comments to an arbitrary level.

[The Number of Parameters]
In calls must be supported up to at least 10 parameters. Most implementations support somewhat more than that, but quite a few enforce some sort of upper limit. For the built-in function, this may be a problem only for `MIN()` and `MAX()`.

[Significant digits]
Must be supported to at least 9 decimal digits. Also, if an implementation supports floating point numbers, it should allow exponents up to 9 decimal digits. An implementation is allowed to operate with different limits for the number of significant digits and the numbers of digits in exponents.

[Subroutine levels]
May be nested to a total of 100 levels, which counts both internal and external functions, but probably not built-in functions. You may actually trip in this limit if you are using recursive solution for large problems. Also, some tail-recursive approaches may crash in this limit.

[Symbol (name) length]
Can be at least 50 characters. This is the name of the symbol, not the length of the value if it names a variable. Nor is it the name of the variable after tail substitution. In other words, it is the symbol as it occurs in the source code. Note that this applies not only to simple symbols, but also compound symbols and constant symbols. Consequently, you can not write numbers of more than 50 digits in the source code, even if `NUMERIC DIGITS` is set high.

[Variable name length]
Of at least 50 characters. This is the name of a variable (which may or may not be set) after tail substitution.

4. Older (Obsolete) Limits

First edition of TRL1 contained some additional limits, which have been relaxed in the second edition in order to make implementation possible for a large set of computers. These limits are:

[Clock granularity]
Was defined to be at least of a millisecond.

Far from all computers provide this granularity, so the requirement have been relaxed. The current requirement is a granularity of at least one second, although a millisecond granularity is advised.

5. What the Standard does not Say

An implementation might enforce a certain limit even though one is not specified in the standard. This section tries to list most of the places where this might be the case:

[The stack]
(Also called: the external data queue) is not formally defined as a concept of the language itself, but a concept to which the REXX language has an interface. Several limits might apply to the stack, in particular the maximum length of a line in the stack and the maximum number of lines the stack can hold at once.

There might also be also be other limits related to the stack, like a maximum number of buffers or a maximum number of different stack. These concepts are not referred to by REXX, but the programmer ought to be aware of them.

[Files]
May have several limits not specified by the definition of REXX, e.g. the number of files simultaneously open, the maximum size of a file, and the length and syntax of file names. Some of these limits are enforced by the operating system rather than an implementation. The programmer should be particularly aware of the maximum number of simultaneously open files, since REXX does not have a standard construct for closing files.

[Expression nesting]
Can in some interpreters only be performed to a certain level. No explicit minimum limit has been put forth, so take care in complex expressions, in particular machine generated expressions.

[Environment name length]
May have some restrictions, depending on your operating system. There is not defined any limit, but there exists an error message for use with too long environment names.

[Clause length]
May have an upper limit. There is defined an error message “Clause too long” which is supposed to be issued if a clause exceeds a particular implementation dependent size. Note that a “clause” does not mean a “line” in this context; a line can contain multiple clauses.

[Source line length]
Might have an upper limit. This is not the same as a “clause” (see above). Typically, the source line limit will be much larger than the clause limit. The source line limit ought to be as large as the string limit.

[Stack operations]
Might be limited by several limits; first there is the number of strings in the stack, then there is the maximum length of each string, and at last there might be restrictions on the character set allowed in strings in the stack. Typically, the stack will be able to hold any character. It will either have “memory” as the limit for the number of string and the length of each string, or it might have a fixed amount of memory set aside for stack strings. Some implementations also set a maximum length of stack strings, often 2*8 or 2*16.

6. What an Implementation is Allowed to “Ignore”

In order to make the REXX language implementable on as many machines as possible, the REXX standard allow implementation to ignore certain features. The existence of these features are recommended, but not required. These features are:

[Floating point numbers]
Are not required; integers will suffice. If floating points are not supported, numbers can have not fractional or exponential part. And the normal division will not be available, i.e. the operator “ /” will not be present. Use integer division instead.

[File operations]
Are defined in REXX, but an implementation seems to be allowed to differ in just about any file operation feature.

7. Limits in Regina

Regina tries not to enforce any limits. Wherever possible, "memory" is the limit, at the cost of some CPU whenever internal data structures must be expanded if their initial size were too small. Note that Regina will only increase the internal areas, not decrease them afterwards. The rationale is that if you happen to need a large internal area once, you may need it later in the same program too.

In particular, Regina has the following limits:

Binary strings	source line size
Clock granularity	0.001-1 second (note 3)
Elapse time clock	until ca. 2038 (note 1)
Hexadecimal strings	source line size
Literal string length	source line size
Nesting of comments	memory
Parameters	memory
Significant digits	memory (note 2)
Subroutine levels	memory
Symbol length	source line size
Variable name length	memory (note 2)

Notes:

1) Regina uses the Unix-derived call `time()` for the elapse time (and time in general). This is a function which returns the number of seconds since January 1st 1970. According to the ANSI C standard, in which Regina is written, this is a number which will at least hold the number 2**31-1. Therefore, these machines will be able to work until about 2038, and Regina will satisfy the requirement of the elapse time clock until 2006. By then, computers will hopefully be 64 bit.

Unfortunately, the `time()` C function call only returns whole seconds, so Regina is forced to use other (less standardized) calls to get a finer granularity. However, most of what is said about `time()` applies for these too.

2) The actual upper limit for these are the maximum length of a string, which is at least 2**32. So for all practical purposes, the limit is "memory".

3) The clock granularity is a bit of a problem to define. All systems can be trusted to have a granularity of about 1 second. Except from that, it's very difficult to say anything more specific for certain. Most systems allows alternative ways to retrieve the time, giving a more accurate result. Wherever these alternatives are available, Regina will try to use them. If everything else fails, Regina will use 1 second granularity.

For most machines, the granularity are in the range of a few milliseconds. Some typical examples are: 20 ms for Sun3, 4 ms for Decstations 3100, and 10 ms for SGI Indigo. Since this is a hardware restriction, this is the best measure anyone can get for these machines.

Definitions

In order to make the definitions more readable, but still have a rigid definition of the terms, some extra comments have been added to some of the definitions. These comments are enclosed in square brackets.

Argument is an *expression* supplied to a *function* or *subroutine*, and it provides data on which the call can work on.

Assignment is a *clause* in which second *token* is the equal sign. [Note that the statements "`a==b`" and "`3=4`" are an (invalid) assignment, not an expression. The type of the first token is irrelevant; if the second token is the equal sign, then the clause is assumed to be an assignment.]

Blanks are characters which *glyphs* are empty space, either vertically or horizontally. A blank is not a *token* (but may sometimes be embedded in tokens), but acts as *token separators*. [Exactly which characters are considered blanks will differ between operating systems and implementations, but the <space> character is always a blank. The <tab> character is also often considered a blank. Other characters considered blank might be the end-of-line <eol>), vertical tab (<vt>), and formfeed (<ff>). See specific documentation for each interpreter for more information.]

Buffer

Caller routine

Character is a piece of information about a mapping from a storage unit (normally a byte) and a *glyph*. Often used as "the meaning of the glyph mapped to a particular storage unit". [The glyph "A" is the same in EBCDIC and ASCII, but the character "A" (i.e. the mapping from glyph to storage unit) differs.]

Character string is an finite, ordered, and possibly empty set of *characters*.

Clause is a non-empty collection of *tokens* in a REXX script. The tokens making up a clause are all the consecutive tokens delimited by two consecutive *clause delimiters*. [Clauses are further divided into *null clauses*, *instructions*, *assignments*, and *commands*.]

Clause delimiter is a non-empty sequence of elements of a subset of *tokens*, normally the linefeed and the semicolon. Also the start and end of a REXX *script* are considered clause delimiters. Also colon is a clause separator, but it is only valid after a label.

Command

Compound variable is a *variable* which name has at least one "`.`" character that isn't positioned at the end of the name.

Current environment is a particular *environment* to which *commands* is routed if no explicit environment is specified for their routing.

Current procedure level is the *procedure level* in effect at a certain point during execution.

Daemon

Decimal digit

Device driver

Digit is a single character having a numeric value associate with its glyph.

Empty string

Environment is a interface to which REXX can route *commands* and afterwards retrieve status information like *return values*.

Evaluation is the process applied to an *expression* in order to derive a *character string*.

Exposing is the binding of a *variable* in the *current procedure level* to the variable having the same name in the *caller routine*. This binding will be in effect for as long as the current procedure level is active.

Exponential form is a way of writing particularly large or small *numbers* in a fashion that makes them more readable. The number is divided into a mantissa and an exponent of base 10.

Expression is a non-empty sequence of *tokens*, for which there exists syntactic restrictions on which tokens can be members, and the order in which the tokens can occur. [Typically, an expression may consist of literal strings or symbols, connected by concatenation and operators.]

External data queue see "stack".

External subroutine is a *script* of REXX code, which is executed as a response to a *subroutine* or *function* call that is neither internal nor built-in.

FIFO

Glyph is an atomic element of text, having a meaning and an appearance; like a letter, a digit, a punctuation mark, etc.

Hex is used as a general abbreviation for term *hexadecimal* when used in compound words like hex digit and hex string.

Hexadecimal digit is a *digit* in the number system having a base of 16. The first ten digits are identical with the *decimal digits* (0-9), while for the last six digits, the first six letters of the Latin alphabet (A-F) are used.

Hexadecimal string is a *character string* that consists only of the *hexadecimal digits*, and with optional *whitespace* to divide the hexadecimal digits into groups. Leading or trailing whitespace is illegal. All groups except the first must consist of an even number of digits. If the first group have an odd number of digits, an extra leading zero is implied under some circumstances.

Instruction is a *clause* that is recognized by the fact that the first *token* is a special *keyword*, and that the clause is not an *assignment* or label. Instructions typically are well-defined REXX language components, such as loops and function calls.

Interactive trace is a *trace* mode, where the *interpreter* halts execution between each *clause*, and offer the user the possibility to specify arbitrary REXX *statements* to be executed before the execution continues.

Label

LIFO

Literal name is a name which will always be interpreted as a constant, i.e. that no variable substitution will take place.

Literal string is a *token* in a REXX *script*, that basically is surrounded by quotation marks, in order to make a *character string* containing the same *characters* as the literal string.

Keyword is a element from finite set of symbols.

Main level

Main program

Name space is a collection of named *variables*. In general, the expression is used when referring to the set of variables available to the *program* at some point during interpretation.

Nullstring is a *character string* having the length zero, i.e. an empty character string. [Note the difference from the undefined string.]

Operating system

Parameters

Parsing

Procedure level

Program is a collection of REXX code, which may be zero or more *scripts*, or other repositories of REXX code. However, a program must contain a all the code to be executed.

Queue see “external data queue” or “stack”.

Routine is a unit during run-time, which is a procedural level. Certain settings are saved across *routines*. One *routine* (the caller *routine*) can be temporarily suspended while another *routine* is executed (the called *routine*). With such nesting, the called *routine* must be terminated before execution of the caller *routine* can be resumed. Normally, the CALL instruction or a function call is used to do this. Note that the main level of a REXX script is also a *routine*.

Script is a single file containing REXX code.

Space separated

Stack

Statement is a *clause* having in general some action, i.e. a clause other than a *null clause*. [Assignments, commands and instructions are statements.]

Stem collection

Stem variable

Strictly order

Subkeyword is a *keyword*, but the prefix “sub” stresses the fact that a *symbol* is a keyword only in certain contexts [e.g. inside a particular instruction].

Subroutine is a *routine* which has been invoked from another REXX *routine*; i.e. it can not be the “main” program of a REXX script.

Symbol

Symbol table

Tail substitution

Term

Token

Token separator

Uninitialized

Variable name

Variable symbol

Whitespace One or several consecutive *blank* characters.

hex literal

norm. hex string

bin {digit,string,literal}

norm. bin string

packed char string

Character strings is the only type of data available in Rexx, but to some extent there are ‘subtypes’ of character strings; character strings which contents has certain format. These special formats is discussed below.

Bibliography

[KIESEL]

Peter C. Kiesel, *REXX - Advanced Techniques for Programmers*. McGraw-Hill, 1993, ISBN 0-07-034600-3

[CALLAWAY]

Merill Callaway, *The ARexx Cookbook*. 511-A Girard Blvd. SE, Albuquerque, NM 87106: Whitestone,1992, ISBN 0-9632773-0-8

[TRL2]

M. F. Cowlishaw, *The REXX Language- Second Edition*. Englewood Cliffs: Prentice-Hall, 1990, ISBN 0-13-780651-5

[TRL1]

M. F. Cowlishaw, *The REXX Language - First Edition*. Englewood Cliffs: Prentice-Hall, 1985, ISBN 0-13-780735-X

[SYMPOS3]

Proceedings of the REXX Symposium forDdevelopers andUusers. Stanford: Stanford Linear Accelerator Center, 1992

[TRH:PRICE]

Stephen G. Price, *SAA Portability*, chapter 37, pp 477-498. In Goldberg ans Smith III [TRH], 1992

[TRH]

Gabriel Goldberg and Smith III, Philip H., *The REXX Handbook*. McGraw-Hill, 1992, ISBN 0-07-023682-8

[DANEY]

Charles Daney, *Programming in REXX*. McGraw-Hill, 1992, ISBN 0-07-015305-1

[BMARKS]

Brian Marks, *Advanced REXX programming*. McGraw-Hill, 1992

[ZAMARA]

Chris Zamara and Nick Sullivan, *Using ARexx on the Amiga*. Abacus, 1991, ISBN 1-55755-114-6

[REXXSAA]

W. David Ashley, *SAA Procedure Language REXX Reference*. 5 Timberline Dr., Trophy Club, Tx 76262: Pedagogic Software, 1991

[MCGH:DICT]

Sybil P. Parker, *McGrw-Hill Dictionary of Computers*. McGraw-Hill, 1984, ISBN 0-07-045415-9

[PJPLAUGER]

P. J. Plauger, *The Standard C Library*. Englewood Cliffs: Prentice Hall, 1992, ISBN 0-13-131509-9

[KR]

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language - Second Edition*. Englewood Cliffs: Prentice Hall, 1988, ISBN 0-13-110362-8

[ANSIC]

Programming languages - C. , Technical Report ISO/IEC 9899:1990, ISO, Case postale 56, CH-1211 Geneve 20, Switzerland, 1990

[OX:CDICT]

Edward L. Glaser and I. C. Pyle and Valerie Illingsworth, *Oxford Reference Dictionary of Computing - Third Edition.* Oxford University Press, 1990, ISBN 0-19-286131-X

[ANSI]

Programming languages - REXX. , ANSI X3.274-1996, 11 West 42nd Street, New York, New York 10036