The ZOC Rexx Reference

(Using Regina Rexx in ZOC)

Version 3.7

Markus Schmidt

Juli, 2018

Based on "*The Regina REXX Interpreter*" which is available at the Regina home page at: http://regina-rexx.sourceforge.net

Original Authors:

Mark Hessling <mark@rexx.org>
Florian Große-Coosmann <florian@grosse-coosmann.de>

Authors/Copyrights:

Table of Contents

1 Preface

This document is based on *The Regina REXX Interpreter* by Anders Christensen and Mark Hessling. In accordance with the document's license I am the author of this document, but in reality I have done little to earn this title. I am seeing myself merely as an editor.

In fact the text of this book is mostly identical with the original version, although some parts have been moved or removed in order to make it more consistent with the intended use. This intention is to provide an introduction to Rexx for users of our terminal product ZOC.

This product uses the Regina REXX interpreter as a scripting solution and it's users will have a different background and will approach REXX differently than the intended audience of the original book.

The full Regina Rexx interpreter is a very powerful tool, intended to serve the same basic purpose as other shell scripting languages like Perl, Python, etc. Some parts of this text will still reflect this. However, I would like to ask the readers to bear with possible inconsistencies and shortcomings. Writing text is not a strength of mine and English is not my native language, but I hope that overall this book it will still be of good value as an introduction, overview and reference about the Rexx language

Last but not least I would like to thank the original authors for their excellent work and for providing the book under this license.

Markus Schmidt

Note: The original book and fully featured REXX interpreter are available at http://regina-rexx.sourceforge.net and both are highly recommend for anyone who is interested in using REXX beyond the scope of our ZOC product.

2 Introduction to ZOC's Regina Rexx

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

2.1 Purpose of this document

The purpose of this document is to provide a fairly complete reference of the Regina REXX language as used with the ZOC terminal product.

For starters, a quick and easy introduction to REXX has been provided in the ZOC help file, which you should read first. Also there is a wealth of REXX tutorials available on the internet..

This book is intended as a reference manual to provide all the details which the intro in the ZOC help file sacrificed for simplicity and to fill the gaps.

2.2 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 added to ZOC as a dynamically loadable library, which ZOC loads at runtime. This library consists all the regular features from Regina, although many do not apply in the context of using Rexx from within ZOC. Description of features which do not apply to using Rexx in ZOC has been removed from this book.

2.3 ZOC REXX Extensions

In addition to the original Regina REXX language elements described here, ZOC extends the language with commands to perform tasks related to terminal emulation. From the perspective or REXX those are a 3rd party library and they are described in chapter 5. (The documentation for the ZOC specific commands can also be found in the help menu of the ZOC program under *ZOC Rexx commands*).

2.4 Executing Rexx programs with ZOC

Rexx programs are generally executed by Regina from the ZOC menu or through the ZOC command line or associated with Host directory entries.

The Rexx programs are usually stored in files with the .ZRX extensions are loaded into memory and passed to the Rexx interpreter for execution.

2.4.1 External Rexx programs

A Rexx program can execute other Rexx programs through the CALL command.

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

First of all we process the environment variable **REGINA_MACROS**. If no file is found we proceed with the current directory and then with the environment variable **PATH**. The semantics of the use of **REGINA_MACROS** and **PATH** are the same, and the search in the current directory is omitted for the superuser on Unix systems for security reasons. The current directory must be specified explicitly by the superuser.

When processing an environment variable, the content is split into the different paths and each path is processed separately. Note that the search algorithm to this point is ignored if the program name contains a file path specification. eg. if "CALL .\MYPROG" is called, then no searching of **REGINA_MACROS** or **PATH** is done; only the concatenation of suffixes is carried out.

For each file name and path element, a concatenated file name is created. If a known file extension is part of the file name only this file is searched, otherwise the file name is extended by the extensions "" (empty string), ".rexx", ".rex", ".cmd", and ".rx" in this order. The file name case is ignored on systems that ignore the character case for normal file operations like DOS, Windows, and OS/2.

The first matching file terminates the whole algorithm and the found file is returned.

The environment variable **REGINA_SUFFIXES** extends the list of known suffixes as specified above, and is inserted after the *empty* extension in the process. **REGINA_SUFFIXES** has to contain a space or comma separated list of extensions, a dot in front of each entry is allowed, e.g. ".macro,.mac,.regina" or "macro mac regina"
Note that it is planned to extend the list of known suffixes by ".rxc" in version 3.4 to allow for seamless integration of pre-compiled macros.

Example: Locating an external Rexx program for execution

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

```
Call myextfunc arg1, arg2
```

Assume also that the file **myextfunc.cmd** exists in the directory /opt/rexx/, and that **PATH**=/usr/bin:/opt/rexx, **REGINA_MACROS** is **not** set, and **REGINA_SUFFIXES**=.macro.

The files that Regina will search for in order are:

```
./myextfunc
./myextfunc.macro
./myextfunc.rexx
./myextfunc.rex
./myextfunc.cmd
./myextfunc.rx

/usr/bin/myextfunc
/usr/bin/myextfunc.macro
/usr/bin/myextfunc.rexx
/usr/bin/myextfunc.rex
/usr/bin/myextfunc.cmd
/usr/bin/myextfunc.rx

/opt/rexx/myextfunc
/opt/rexx/myextfunc.macro
/opt/rexx/myextfunc.rexx
/opt/rexx/myextfunc.rex
/opt/rexx/myextfunc.cmd      /* found!! terminate search*/
```

3 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 as described in the first part of the chapter.

3.1 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 {20}.

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}.

3.2 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, these */
/* are several clauses while the comment above is a     */
/* single comment.                                      */

-- These lines also consist of multiple comments, and thus
-- multiple clauses. This form of comment was introduced
-- in Regina 3.4
```

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 is expected. 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 not enabled). 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.

3.3 Commands

3.3.1 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.

3.4 Basic REXX 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 of 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 preceeding 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 explained in a chapter on tracing, to be written later.

In addition to this, variants may also exist. 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.

3.4.1 The ARG Instruction

```
ARG [ template ] [, [ 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.

3.4.2 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 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.

Breakage Alert!!

Prior to Version 3.1 of Regina, the following syntactical use of the CALL instruction was valid:

```
CALL routine '(' [ parameter ] [, [ parameter ] ... ] ')' ;
```

e.g.

```
call myfunc('abcDEF',,"")
```

This syntax is not allowed by ANSI and use of this syntax will now result in Error 37.1. There exists an option introduced in Regina 3.3 which reenables a similar behaviour, although parameters with individual parentheses are allowed since 3.1. The option is called CALLS_AS_FUNCS and should
be enabled using the environment variable called REGINA_OPTIONS. See the description of the instruction OPTIONS for further details.

Breakage Alert!!

For the CALL instruction, watch out for interference with line continuation. If there are trailing commas, it might be interpreted as line continuation. Appending a semicolon where appropriate is a common solution to make the desired bahaviour obvious. If a CALL instruction uses 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; Regina restores the current OPTIONS. Note that external subroutines don't inherit the current OPTIONS as

internal subroutines do. See the section OPTIONS for a detailed explanation. 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.

3.4.3 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.

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 f 6
end
call f 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 5 and 5, while in the second loop, the numbers 4 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`.

3.4.4 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.

3.4.5 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.

3.4.6 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 not 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}.

3.4.7 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.

3.4.8 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.

3.4.9 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 digits 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.

3.4.10 The PARSE Instruction

```
PARSE [ option ] [ CASELESS ] type [ template ] ;
    option = { UPPER | LOWER }
    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 specified by *template*. *template* can be a number of patterns seperated by commas. A complete description of parsing is given in chapter [not yet written].

If the *option* **UPPER** is specified, the input source strings are uppercased (based on locale) before being split into the variables specified by template.

If the *option* **LOWER** is specified, the input source strings are lowercased (based on locale) before being split into the variables specified by template.

If **CASELESS** is specified, any character strings in *template* will be matched against the source strings irrespective of case (based on locale).

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 now compliant to the ANSI Standard.
Thus PARSE VERSION will return 5.00.

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].

3.4.11 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.

3.4.12 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.

3.4.13 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.

3.4.14 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
```

3.4.15 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.

3.4.16 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.

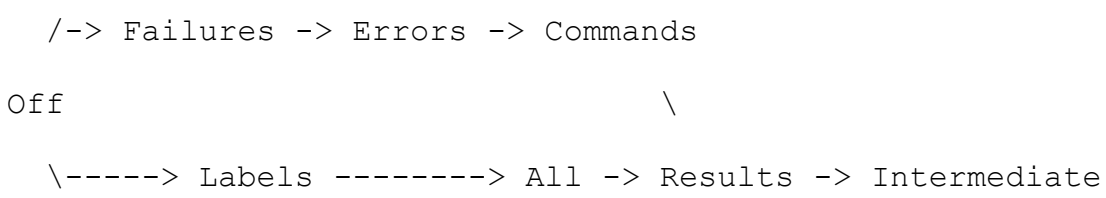
[O]
: (Off) Turns off all tracing.

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

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.

3.4.17 The UPPER Instruction

```
UPPER symbol [ symbol [ symbol [...] ] ] ;
```

The UPPER instruction is used to translate the contents of one or more variables to uppercase. The variables are translated in sequence from left to right.

Each symbol is separated by one or more blanks.

While it is more convenient and probably faster than individual calls to TRANSLATE, UPPER is not part of the ANSI standard and is not common in other interpreters so should be avoided. It is provided to ease porting of programs from CMS.

Only simple and compound symbols can be specified. Specification of a stem variable results in an error.

3.5 Advanced Instructions

3.5.1 The ADDRESS Instruction

```
ADDRESS = [ environment  [ command ] [ redirection ] ] ;
     [ [ VALUE ] expr [ redirection ] ] ;

redirection : = WITH input_redir [output_redir] [error_redir]
     WITH input_redir [error_redir] [output_redir]
     WITH output_redir [input_redir] [error_redir]
     WITH output_redir [error_redir] [input_redir]
     WITH error_redir [input_redir] [output_redir]
     WITH error_redir [output_redir] [input_redir]

input_redir : = INPUT NORMAL
     INPUT io

output_redir : = OUTPUT NORMAL
     OUTPUT [ APPEND | REPLACE ] io

error_redir : = ERROR NORMAL
     ERROR [ APPEND | REPLACE ] io

io : = { STREAM | STEM | LIFO | FIFO } symbol
     { STREAM | LIFO | FIFO } string
```

We will discuss redirection later.

The ADDRESS instruction controls where commands to an 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. Starting with Regina 3.0 the special variables .RC and .RS are set too, according to the ANSI standard.

In other words: All *normal* commands are ADDRESS statements with a suppressed keyword and environment.

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. eg.

```
SYSTEM='PATH'
ADDRESS SYSTEM "echo Hello"
```

is equivalent to a plain

```
ADDRESS SYSTEM "echo Hello"
```
or
```
ADDRESS "SYSTEM" "echo Hello"
```

for the external *echo* command.
A symbol specified as an environment name isn't case-sensitive, whereas a string must match the case. Built-in environments are always uppercased.

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. 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. Regina has an infinite list and never pushes out any entry. Possible values are listed below. If you supply a *command* with the `ADDRESS` statement, the *environment* is interpreted as a temporary change for just this command.

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. This is Regina's behavior. Every environment exists only once. The redirection command below always changes the behavior of one -- the given -- environment. You can imagine a set of playing cards in your hand. The operation is to draw one card by name and put it to the front.

If you do not specify any subkeywords or parameters to `ADDRESS`, 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

Let's look at some examples, they can sometimes be combined with a redirection:

```
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 the current environment.

The third example sets SYSTEM as the 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 swaps the two uppermost entries on the stack; and SYSTEM ends up at the top pushing the environment specfied in *newenv* below it.

The sixth clause is equivalent to the fourth example, but is not allowed by ANSI. Since Regina 3.0 this style is deprecated and can't be used if OPTIONS STRICT_ANSI is in effect. Again, avoid this kind of ADDRESS statement style, and use the VALUE version instead.

Example: The VALUE subkeyword

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

```
ADDRESS ENV

ADDRESS VALUE ENV
```

The first of these sets the current default environment to ENV, while the second sets it to the value of the symbol ENV.

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 the current environment; and if it is an illegal name, the interpreter will complain only when the environment is actually used. Other implementations may not allow you to select an invalid environment name at all.

Regina allows every name as an environment name. Regina gives an error message about wrong names only when the name is used. The error string looks somewhat strange if Regina is used as a separate program, since the extension of the environment name space is only useful when running as part of a program which extends the standard names.

Regina uses three kinds of environments. Some have alias names. The environment names are:

SYSTEM
    alias OS2ENVIRONMENT
    alias ENVIRONMENT

This is the default environment which is selected at startup. The standard operating system command line interpreter will be loaded to execute the commands. You can use the built-in commands of the command line interpreter, often called shell, or any other program which the command line interpreter can find and load.

COMMAND
    alias CMD
    alias PATH

This environment loads the named program directly. You may supply a path if this is needed for the current operating system to load the program, otherwise Regina uses the standard operating system search rules for programs. This is done by searching through the items of the PATH system-environment variable in most operation systems. You can't use built-in shell functionality like system redirections like you can with SYSTEM. Regina's redirections are more powerful and work in either environment.

REXX
    alias REGINA

This environment uses a new instance of the Regina interpreter program to execute a program. The program has to be a REXX script. This environment has several advantages. The output of a script can be redirected, the process is independent and a risk of a crash is minimized when playing with external libraries, finally, Regina itself tries to find the correct REXX interpreter by itself and does everything to create a new incarnation of Regina.

The definition of REXX says nothing 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 is no NONE environment in standard REXX, i.e. an environment that ignores commands, but some interpreters implement the TRACE setting ??? to accomplish this. Regina uses the environment SYSTEM as the preselected environment as mentioned above. More implementation specific details can be found in the section implementation specific documentation for Regina.

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

ADDRESS Redirections

ANSI defines redirections for the ADDRESS statement. This feature has been missing from Regina until version 3.0; although you have had the chance to redirect input and output by using **LIFO**> and >**FIFO** modifiers on command strings.

These command modifiers still exist and have a higher precedence than the ANSI defined

redirections. Note, that **LIFO** and **FIFO** can be used by the newer redirection system. But, first of all, some examples show the usage of ADDRESS redirections.

```
ADDRESS SYSTEM "sort" WITH INPUT STEM names. OUTPUT STEM
            names.

ADDRESS SYSTEM "myprog" WITH INPUT STEM somefood. OUTPUT
            STREAM 'prg.out' ERROR STEM oops.

ADDRESS PATH WITH INPUT FIFO '' OUTPUT NORMAL

ADDRESS SYSTEM WITH INPUT FIFO '' OUTPUT FIFO '' ERROR NORMAL

ADDRESS SYSTEM "fgrep 'bongo'" WITH INPUT STREAM 'feeder'
```

The first command instructs the default command line interpreter to call the program called *sort*. The input for the command is read from the stem *names.* (note the trailing period) and the output is sent back to the same stem variable after the command terminates. Thus, bothering about the implementation of a fast sort algorithm for a stem is as simple as calling a program which can actually do the sort.

A program called *myprog* is called in the second case. The input is fetched from the stem *somefood.* (again note the trailing period), and the standard output of the program is redirected to the stream called prg.out. Any generated error messages via the standard error stream are redirected to the stem called *oops.*

In the third example, the redirection behavior of the environment PATH is changed for all future uses. The input for all commands addressed to this environment is fetched from the standard stack in FIFO order. After each call the stack will be flushed. The output is sent to the default output stream, which is the current console in most cases. The behavior for error messages is not changed.

The fourth example allows pipes between commands in the environment; SYSTEM for all future uses. The input is fetched from the default stack and sent to the default stack after each command. The stack itself is flushed in between. Each executed program will write to something which is the input to the next called command. The error redirection is set or set back to the initial behavior of writing to the standard error stream.

The fifth example relates to the fourth. The default stack has to be filled with something initially. This is done by the redirection to the stream “feeder” while writing the output of the *fgrep* command to the default FIFO as declared in example four. After this, a single line with a simple *sort* command will sort the output of *fgrep* and place it in the default stack. You can fetch the final output of your pipe cascade by reading the stack contents. This statement overwrites some of the rules of the fourth example temporarily.

You can see the powerful possibilities of the redirection command. The disadvantage is the loss of a direct overview of what happens after a permanent redirection command has executed.

Its now the time to show you all rules and semantics of the redirection.

Rules for the redirection by the keyword `WITH` of the `ADDRESS` statement:

- Every environment has its own default *redirection set*.
- Every *redirection set* consists of three independent *redirection elements*; standard input (`INPUT`), standard output (`OUTPUT`) and standard error (`ERROR`). Users with some experiences with Unix, DOS & Windows or OS/2 may remember the redirection commands of the command line interpreter which can redirect each of the streams, too. This is nearly the same.
- Each *redirection element* starts with the program-startup streams given to REXX when invoking the interpreter. These can be reset to the startup default by specifying the argument `NORMAL` for each *redirection element*.
- The sequence of the *redirection elements* is irrelevant.
- You can specify each *redirection element* only once per statement.
- Redirections can be intermixed. This means you can let both the `OUTPUT` and the `ERROR` redirection point to the same "thing". The data from the different channels will be put to the assigned "thing" as they arrive. ANSI's point of view isn't very clear at this point. They state to keep the output different for files and put them together after the called program finished while the data shall be mixed at once when using stems.
  Regina always mixes the fetched data at once if possible.
- Redirections from and to the same source/destination try to keep the data consistent. If the `INPUT`/`OUTPUT` pair or the `INPUT`/`ERROR` pair points to the same destination, the content of the input or output channel is buffered so that writing to the output won't overwrite the input.
- A *redirection element* is entered by its name (e.g. `INPUT`), a redirection processor (e.g. `STREAM`) and a destination symbol (e.g. OUT_FN) following the rules to the redirection processor. This means that you have to enter a dot after a symbol name for a stem, or any symbol for the rest of the processors, in which case the content of the symbol is used as for normal variables.
- Both `OUTPUT` and `ERROR` streams can replace or append the data to the destination. Simply append either `APPEND` or `REPLACE` immediately after the `OUTPUT` or `ERROR` keywords. `REPLACE` is the default.
- The destination is checked or cleared prior to the execution of the command.
- ANSI defines two redirection processors: `STEM` and `STREAM`. The processors `LIFO` and `FIFO` are allowed extensions to the standard.
- The processor `STEM` uses the content of the symbol *destination.0* to access the count of the currently accessible lines. *destination* is the given destination name, of course. *destination.0* must be filled with a whole, non-negative number in terms of the DATATYPE built-in function. Each of *n* lines can be addressed by appending the whole numbers one to *n* to the stem. Example: `STEM` foo. is given, FOO.0 contains 3. This indicates three content lines. They are the contents of the symbols FOO.1 and FOO.2 and FOO.3 .
- The processor `STREAM` uses the content of the symbol *destination* to use a stream as known in the STREAM built-in function. The usage is nearly equivalent to the commands LINEIN *destination* or LINEOUT *destination* for accessing the contents of the file. An empty variable (content set to the empty string) as the content of the *destination* is allowed and indicates the default input, output or error streams given to the REXX program. This is equivalent to the NORMAL keyword.
- The processor `LIFO` uses the content of the symbol *destination* as a queue name. New lines are pushed in last-in, first-out order to the queue. An empty *destination* string is allowed and describes the default queue. Lines are fetched from the queue if this processor is used for the `INPUT` stream.
- The processor `FIFO` uses the content of the symbol *destination* as a queue name. New lines are

pushed in first-in, first-out order to the queue. An empty *destination* string is allowed and describes the default queue. Lines are fetched from the queue if this processor is used for the INPUT stream.

- On INPUT, all the data in the input stream is read up to either the end of the input data or until the called process terminates. The latter one may be determined after feeding up the input stream of the called process with unused data. Thus, there is no way to say if data is used or not. This isn't a problem with STEMs. But all file related sequential access objects including LIFOs and FIFOs may have lost data between two calls. Imagine an input file (STREAM) with three lines:

  ```
  One line
  DELIMITER
  Second line
  ```

  and furthermore two processes **p1** and **p2** called WITH INPUT STREAM **f** with **f** containing the three lines above. **p1** reads lines up until a line containing DELIMITER and **p2** processes the rest. It is very likely that the second process won't fetch any line because the stream may be processed by REXX, and REXX may has put one or more lines ahead into the feeder pipe to the process. This might or might not happen. It is implementation dependent and Regina shows this behavior. The input object is checked for existence and if it is properly set up before the command is started.
  In short: INPUT may or may not use the entire input.
- Both OUTPUT and ERROR objects are checked for being properly set up just before the command starts. REPLACE is implemented as a deletion just before the command starts. Note that ANSI doesn't force STEM lines to be dropped in case of a replacement. A big stem with thousands of lines will still exist after a replacement operation if the called command doesn't produce any output. Just destination.0 is set to 0.

The redirection of commands is a mystery to many people and it will continue be. You can thank all the people who designed stacks, queues, pipelines and all the little helper utilities of a witch's kitchen of process management.

3.5.2 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.

3.5.3 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.

3.5.4 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.

3.5.5 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.

3.5.6 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.

3.5.7 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.

3.6 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.

3.6.1 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 decreasing 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.

3.6.2 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.

3.6.3 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 sensitive, 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.
>>=	Strictly greater than or equal.
<<=	Strictly less than or equal.

3.6.4 Concatenation Operators

The concatenation operators combine two strings to form one, by appending the second 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 space.

Examples:

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

3.6.5 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.

4 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.

4.1 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.

4.1.1 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. At the right of each function syntax is an indication of where the function is defined.

(ANSI)	ANSI Standard for REXX 1996
(EXT-ANSI)	Extended REXX
(SAA)	System Application Architecture - IBM
(OS/2)	IBM OS/2 REXX
(CMS)	REXX on CMS
(AREXX)	AREXX on Amiga
(REGINA)	Additional function provided by Regina

Definitions of the AREXX built-in functions have been taken verbatim from
http://dfduck.homeip.net/dfd/ados/arexx/main.shtml

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.

4.1.2 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.

4.1.3 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.

4.1.4 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. Regina also provides further information in errors in built-in functions, as defined by the ANSI standard. This additional information is provided as sub-error messages and usually provide a more detailed explanation of the error.

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.

4.1.5 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 no 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. It is also dependent on whether the user specifies the **-a** command line switch.

- `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. Error messages may also be returned in the user's native language.

- `QUEUED()` refers to the system specific concept of a "stack", which is either internal or external to the implementation. 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.

The built-in functions maked as **AREXX** are available by default on Amiga and AROS systems, but the **AREXX_BIFS** OPTION is required on other system to make these functions available.

As you can see, even REXX interpreters that are within the standard can differ quite a lot in the built-in functions. 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 ...

4.1.6 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 space bar 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.

4.2 Regina Built-in Functions

Below follows an in depth description of all the functions in the library of built-in functions. Note that all functions in this section are available on all ports of Regina. Each function is designated as being part of the ANSI standard, or from other implementations. Following sections describe those built-in functions that are available on specific ports of Regina, or when Regina is built with certain switches.

ABBREV(long, short [,length]) - (ANSI)

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) - (ANSI)

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([option]) - (ANSI)

Returns the current default environment to which commands are sent or optionally specific details about the targets of command input/output and errors. The value is set with the `ADDRESS` clause, for more information, see documentation on that clause.

If *option* is not specified the default option is “N”.
Option can be:

[N]

(Normal) Returns the current default environment.

[I]

(Input) Returns the target details for input as three words: *position type resource*.

[O]

(Output) Returns the target details for output as three words: *position type resource*.

[E]

(Error) Returns the target details for errors as three words: *position type resource*.

position will be one of: **INPUT** (for option "I"), **APPEND** or **REPLACE**
type will be one of: **STEM**, **STREAM**, **FIFO**, **LIFO**, **NORMAL**
resource will be the name of the stem, stream or queue or blank

`ADDRESS()`	`SYSTEM  /* Maybe */`
`ADDRESS('N')`	`SYSTEM  /* Maybe */`
`Defaults:`	
`ADDRESS('I')`	`INPUT NORMAL`
`ADDRESS('O')`	`REPLACE NORMAL`
`ADDRESS('E')`	`REPLACE NORMAL`
`After: ADDRESS SYSTEM WITH INPUT FIFO 'MYQUEUE' OUTPUT STEM mystem. ERROR APPEND STREAM 'my.err'`	
`ADDRESS('I')`	`INPUT FIFO MYQUEUE`
`ADDRESS('O')`	`REPLACE STEM MYSTEM.`
`ADDRESS('E')`	`APPEND STREAM my.err`

ARG([argno [,option]]) - (ANSI)

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. On all platforms, Regina will receive one parameter string at the main procedural level, unless Regina is started with the -a switch, when multiple parameter strings are passed in.

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 doesn't exist*/
ARG(2,'E')	1
ARG(2,'O')	0
ARG(3,'E')	0 /*Third parameter omitted */
ARG(9,'O')	1

B2C(binstring) - (AREXX)

Converts a string of binary digits(0,1)into the corresponding(packed)character representation. The conversion is the same as though the argument string had been specified as a literal binary string(e.g. '1010'B). Blanks are permitted in the string,but only at byte boundaries. This function is particularly useful for creating strings that are to be used as bit masks.

B2C('00110011')	'3'
B2C('01100001')	'A'

B2X(binstring) - (ANSI)

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'`

BEEP(frequency [,duration]) - (OS/2)

Sounds the machine's bell. The *frequency* and *duration* (in milliseconds) of the tone are specified. If no *duration* value is specified, it defaults to 1. Not all operating systems can sound their bells with the given specifications.

`BEEP(50,1000)`	

BITAND(string1 [,[string2] [,padchar]]) - (ANSI)

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`

BITCHG(string, bit) - (AREXX)

Changes the state of the specified *bit* in the argument *string*. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string.

BITCHG('0313'x,4)	'0303'x

BITCLR(string, bit) - (AREXX)

Clears (sets to zero) the specified *bit* in the argument *string*. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string.

BITCLR('0313'x,4)	'0303'x

BITCOMP(string1, string2, bit [,pad]) - (AREXX)

Compares the argument strings bit-by-bit,starting at bit number 0. The returned value is the bit number of the first bit in which the strings differ,or -1 if the strings are identical.

BITCOMP('7F'x,'FF'x)	'7'
BITCOMP('FF'x,'FF'x)	'-1'

BITOR(string1 [, [string2] [,padchar]]) - (ANSI)

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

BITOR('12x)	'12'x
BITOR('15'x,'24'x)	'35'x
BITOR('15'x,'2456'x)	'3556'x
BITOR('15'x,'2456'x,'F0'x)	'35F6'x
BITOR('1111'x,,'4D'x)	'5D5D'x
BITOR('pQrS',,'20'x)	'pqrs' /* ASCII */

BITSET(string, bit) - (AREXX)

Sets the specified *bit* in the argument *string* to 1. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string.

`BITSET('0313'x,2)`	`'0317'x`

BITTST(string, bit) - (AREXX)

The boolean return indicates the state of the specified bit in the argument string.
Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte to the string.

`BITTST('0313'x,4)`	`'1'`

BITXOR(string1[, [string2] [,padchar]]) - (ANSI)

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`

BUFTYPE() - (CMS)

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
```

C2B(string) - (AREXX)

Converts the supplied *string* into the equivalent string of binary digits.

C2B('abc')	'011000010110001001100011'

C2D(string [,length]) - (ANSI)

Returns a 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) - (ANSI)

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'

CD(directory) - (REGINA)

CHDIR(directory) - (REGINA)

Changes the current process's directory to the *directory* specified. A more portable, though non-standard alternative is to use the DIRECTORY BIF.

CHDIR('/tmp/aa')	/* new directory now /tmp/aa */

CENTER(string, length [, padchar]) - (ANSI)

CENTRE(string, length [, padchar]) - (ANSI)

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) - (ANSI)

The purpose of this function 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]]) - (ANSI)

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 know many characters were read other 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 reads the stream raw. Some operating systems use special characters to differentiate 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 identically 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]]) - (ANSI)

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]) - (ANSI)

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 */

CLOSE(file) - (AREXX)

Closes the *file* specified by the given logical name. The returned value is a boolean success flag, and will be 1 unless the specified file was not open.

CLOSE('input')	'1'

COMPARE(string1, string2 [,padchar]) - (ANSI)

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'

COMPRESS(string [,list]) - (AREXX)

If the *list* argument is omitted,the function removes leading,trailing,or embedded blank characters from the *string* argument. If the optional *list* is supplied, it specifies the characters to be removed from the *string*.

COMPRESS(' why not ')	'whynot'
COMPRESS('++12-34-+','+-')	'1234'

CONDITION([option]) - (ANSI)

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.

[E]

(Extra) The error code (a single number) and a sub-error code if available (two numbers and a period; eg 40.5) that was generated by the condition.

[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) - (ANSI)

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) - (ANSI)

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

CRYPT(string, salt) - (REGINA)

Encrypts the given *string* using the supplied *salt* and returns the encrypted string. Only the first two

characters of *salt* are used. Not all operating systems support encryption, and on these platforms, the string is returned unchanged. It is also important to note that the encrypted string is not portable between platforms.

`CRYPT('a string', '1x')`	`'1xYwPPWI1zRJs' /* maybe */`

DATATYPE(string [,option]) - (ANSI)

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]]]) - (ANSI)

This function returns information relating to the current local 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. (ANSI)

[C]
(Century) The number of days in this century from January 1st −00 until today, inclusive. The return value will be a positive integer. (Regina Extension)

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

[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. (ANSI)

[I]
(ISO) Returns the date according the format specified by International Standards

Organization Standard ISO 8601:2004. The format will be "`yyyy-mm-dd`", and each part is padded with leading zero where appropriate. (Regina Extension)

[M]

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

[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. (ANSI)

[O]

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

[S]

(Standard/Sorted) Returns the date according the format specified by International Standards Organization Standard ISO 2014-1976 (E). The format will be "`yyyymmdd`", and each part is padded with leading zero where appropriate. (ANSI)

[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. (ANSI)

[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. (ANSI)

[T]

(*time_t*) Returns the current date/time in UNIX *time_t* format. *time_t* is the number of seconds since January 1st 1970. (Regina Extension)

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()` returns its result are affected by the settings of `NUMERIC`. Also note that if there is more than one call to `DATE()` (or `TIME()`) in a single clause of REXX code, all of them will use the same basis data for calculating the date (or 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'
DATE('T')	694620000
DATE('I')	'1992-01-06'

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:
BDEOUNST.
The default value for *option_in* is **N**.
When a date is converted to format **T**, the returned value is the input date with a time of 00:00:00.

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]) - (ANSI)

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]) (ANSI)

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*/`

DESBUF() - (CMS)

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()`	`0`

DIGITS() - (ANSI)

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 */`

DIRECTORY([new directory]) - (OS/2)

Returns the current directory for the running process, and optionally changes directory to the specified *new directory*. If the *new directory* exists, and the change to *new directory* succeeds, the *new directory* is returned. If the *new directory* does not exist or an error occurred changing to that *new directory*, the empty string is returned.

`DIRECTORY()`	`'/tmp' /* Maybe */`
`DIRECTORY('c:\temp')`	`'c:\temp' /* Maybe */`

D2C(integer [,length]) - (ANSI)

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)	Error 40.13
D2C(-10,3)	'fffff5'x

D2X(integer [,length]) - (ANSI)

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)	Error 40.13
D2X(-10,5)	'ffff5'x

DROPBUF([number]) - (CMS)

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 */`

EOF(file) - (AREXX)

Checks the specified logical *file* name and returns the boolean value 1(True) if the end-of-file has been reached, and 0(False)otherwise.

`EOF('infile')`	`'1' /* maybe */`

ERRORTEXT(errno [, lang]) - (ANSI)

Returns the REXX error message associated with error number *errno*. If the *lang* character is specified, it will determine the native language in which the error message is returned. The default value for *lang* is "`N`".

Possible options are:

[N]
(Normal) The error text is returned in the default native language.

[S]
(Standard English) The error text is returned in English.

For more information on how Regina supports different native languages, see **Native Language Support.**
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'

errno can also be specified as an *errno* followed by a sub error number, with a period between. The resulting string will be the text of the sub-error number with placemarkers indicating where substitution values would normally be placed.

ERRORTEXT(40.24)	<bif> argument 1 must be a binary string; found "<value>"

Regina also supports messages in several native languages. See the section on **Native Language Support** for details on how this is configured. With **DE** as the native language in effect:

ERRORTEXT(40.24)	Routine <bif>, Argument 1 muß eine Binätzeichenkette sein; "<value>"
ERRORTEXT(40.24,'S')	<bif> argument 1 must be a binary string; found "<value>"

EXISTS(filename) - (AREXX)

Tests whether the specified name of the given *filename* exists. The *filename* string may include any portion of a full file path specification. Note that the argument is not a logical file name used in other ARexx file functions. A more portable equivalent of this is to use the 'QUERY EXISTS' command of the STREAM BIF.

EXISTS('c:\temp\infile.txt')	'1' /* maybe */

EXPORT(address, [string], [length] [,pad]) - (AREXX)

Copies data from the (optional) string into a previously-allocated memory area, which must be specified as a 4-byte *address*. The *length* parameter specifies the maximum number of characters to be copied; the default is the length of the string. If the specified *length* is longer than the string, the remaining area is filled with the *pad* character or nulls('00'x). The returned value is the number of characters copied.
Caution is advised in using this function. Any area of memory can be overwritten,possibly causing a system crash.
See also STORAGE() and IMPORT().
Note that the *address* specified is subject to a machine's endianess.

EXPORT('0004 0000'x,'The answer')	'10'

FILESPEC(option, filespec) - (OS/2)

Returns the specified portion of a passed *filespec*, depending on the *option* passed.
Possible options are:

[Drive]
The file's drive. On platforms that don't have the concept of a drive letter, returns blank.
[Name]
The file's name. This is the string following the last path delimiter (if there is one).
[Path]
The file's path. This is the string up to, but not including the last path delimiter.
Only the first letter of *option* is required.

FILESPEC('Drive','C:\config.sys')	'C'
FILESPEC('Name','C:\config.sys')	'config.sys'
FILESPEC('Path','C:\config.sys')	'\'
FILESPEC('Drive','/usr/bin/regina')	''
FILESPEC('Name','/usr/bin/regina')	'regina'
FILESPEC('Path','/usr/bin/regina')	'/usr/bin'

FIND(string, phrase) - (CMS)

Searches *string* for the first occurrence of the sequence of blank-delimited words *phrase*, and return the word number of the first word of *phrase* in *string*. Multiple blanks between words are treated as a single blank for the comparison. Returns 0 if *phrase* not found. Deprecated: see WORDPOS().

FIND('now is the time','is the time')	2
FIND('now is the time','is the')	2
FIND('now is the time','is time')	0

FORK() - (REGINA)

This function spawns a new process as a child of the current process at the current point in the program where FORK is called. The program then continues from this point as two separate processes; the parent and the child. FORK returns 0 to the child process, and the process id of the child process spawned to the parent (always non-zero). A negative return value indicates an error while attempting to create the new process. FORK is not available on all platforms. If FORK is not supported, it will always return '1'. It is safe to assume that a return value of '1' means that FORK is not supported. All platforms AFAIK, will never return '1' as a child process id; that number is usually reserved for the first process that starts on a machine.

FORK()	'0' /* To child */
	'3456' /* maybe to parent */

FORM() - (ANSI)

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]]]]]) - (ANSI)

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'

FREESPACE(address, length) - (AREXX)

Returns a block of memory of the given *length* to the interpreter's internal pool. The address argument must be a 4-byte string obtained by a prior call to GETSPACE(),the internal allocator. It is not always necessary to release internally-allocated memory,since it will be released to the system when the program terminates. However,if a very large block has been allocated,returning it to the pool may avoid memory space problems. The return value is a boolean success flag.
See also GETSPACE()

FREESPACE('00042000'x,32)	'1'

FUZZ() - (ANSI)

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 */

GETENV(environmentvar) - (REGINA)

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' )
```

GETPID() - (REGINA)

Returns the process id of the currently running process.

GETPID()	'234' /* Maybe */

GETSPACE(length) - (AREXX)

Allocates a block of memory of the specified length from the interpreter's internal pool. The returned value is the 4-byte address of the allocated block, which is not cleared or otherwise initialized. Internal memory is automatically returned to the system when the Rexx program terminates,so this function
should not be used to allocate memory for use by external programs.
See also FREESPACE()

GETSPACE(32)	'0003BF40' /* maybe */

GETTID() - (REGINA)

Returns the thread id of the currently running process.

GETTID()	'2' /* Maybe */

HASH(string) - (AREXX)

Returns the hash attribute of a string as a decimal number,and updates the internal hash value of the string.

HASH('1')	'49'

IMPORT(address [,length]) - (AREXX)

Creates a string by copying data from the specified 4-byte *address*. If the *length* parameter is not supplied,the copy terminates when a null byte is found.
See also EXPORT()
Note that the *address* specified is subject to a machine's endianess.

IMPORT('0004 0000'x,10)	'The answer' /* maybe */

INDEX(haystack, needle [,start]) - (CMS)

Returns the character position of the string *needle* in *haystack*. If *needle* is not found, 0 is returned. By default the search starts at the first character of haystack (*start* is 1). This can be overridden by giving a different *start*, which must be a positive, whole number. See POS function for an ANSI function that does the same thing.

INDEX('abcdef','cd')	'3'
INDEX('abcdef','xd')	'0'
INDEX('abcdef','bc',3)	'0'
INDEX('abcabc','bc',3)	'5'
INDEX('abcabc','bc',6)	'0'

INSERT(string1, string2 [,position [,length [,padchar]]]) - (ANSI)

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'`

JUSTIFY(string, length [,pad]) - (CMS)

Formats blank-delimited words in *string*, by adding *pad* characters between words to justify to both margins. That is, to width *length* (*length* must be non-negative). The default *pad* character is a blank.
string is first normalized as though SPACE(*string*) had been executed (that is, multiple blanks are converted to single blanks, and leading and trailing blanks are removed). If *length* is less than the width of the normalized string, the string is then truncated on the right and any trailing blank is removed. Extra *pad* characters are then added evenly from the left to right to provide the required length, and the blanks between words are replaced with the *pad* character.

`JUSTIFY('The blue sky',14)`	`'The  blue   sky'`
`JUSTIFY('The blue sky',8)`	`'The blue'`
`JUSTIFY('The blue sky',9)`	`'The  blue'`
`JUSTIFY('The blue sky',9,'+')`	`'The++blue'`

LASTPOS(needle, haystack [,start]) - (ANSI)

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]) - (ANSI)

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) - (ANSI)

Returns the number of characters in *string*.

LENGTH('')	'0'
LENGTH('Foo')	'3'
LENGTH('Foo bar')	'7'
LENGTH(' foo bar ')	'10'

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

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. Afterward, 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]]) - (ANSI)

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] [,option]) - (ANSI)

Returns 1 if there is at least one complete line remaining in the named file *stream* or 0 if no complete lines remain in the file. 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 ANSI Standard has extended this function from TRL2. It allows an *option*:

[C]
: (Count) Returns the actual number of complete lines remaining in the stream, irrespective of how expensive this operation is.

[N]
: (Normal) Returns 1 if there is at least one complete line remaining in the file or 0 if no lines remain. This is the default. To maintain backwards compatibility with older releases of Regina, the OPTION; NOFAST_LINES_BIF_DEFAULT can be used to make the default option behave as though LINES(streamid,'C') was specified.

LINES will only return 0 or 1 for all transient streams, as the interpreter can not reposition in these files, and can therefore not count the number of remaining lines.

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','C')	'2' /* Maybe */
LINES('/tmp/file')	'1' /* Maybe */

LOWER(string [,start [,length [,pad]]]) - (REGINA)

Translates the substring of *string* that starts at *start*, and has the length *length* to lower case. *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.

If a specific locale is set (via the -l switch), then the string is set to the correct lowercase values based on that locale.

LOWER('One Fine Day')	'one fine day'
LOWER('FRED', 2)	'Fred'
LOWER('FRED', 3, 1)	'FreD'
LOWER('FRED',1, 10, '*')	'fred******'

MAKEBUF() - (CMS)

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 */

MAX(number1 [,number2] ...) - (ANSI)

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] ...) - (ANSI)

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

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

OPEN(file, filename, ['Append'|'Read'|'Write']) - (AREXX)

Opens a file for the specified operation. The *file* argument defines the logical name by which the file will be referenced. The *filename* is the external name of the file, and may include any portions of a full file path.
The function returns a boolean value that indicates whether the operation was successful. There is no limit to the number of files that can be open simultaneously, and all open files are closed automatically when the program exits.
See also CLOSE(), READ(), WRITE()

OPEN('myfile','c:\temp\aa.txt','R')	'1'
OPEN('infile','/tmp/fred.txt')	'1'

OVERLAY(string1, string2 [,[start] [,[length] [,padchar]]]) - (ANSI)

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',4)`	`'oldNEWlue'`
`OVERLAY('NEW','old-value',4,5)`	`'oldNEW  e'`
`OVERLAY('NEW','old-value',4,5,'*')`	`'oldNEW**e'`
`OVERLAY('NEW','old-value',4,2)`	`'oldNEalue'`
`OVERLAY('NEW','old-value',9)`	`'old-valuNEW'`
`OVERLAY('NEW','old-value',12)`	`'old-value  NEW'`
`OVERLAY('NEW','old-value',12,,'*')`	`'old-value**NEW'`
`OVERLAY('NEW','old-value',12,5,'*')`	`'old-value**NEW**'`

POOLID() - (REGINA)

Returns the current call level for the current procedure.

`POOLID()`	`'1' /* top level */`
`POOLID()`	`'6' /* 6th level call nesting */`

POPEN(command [,stem.]) - (REGINA)

Runs the operating system *command*. If the optional *stem*. is supplied all output from the *command* is placed in the specified stem variable as a REXX array. Note that only the command's stdout can be captured.
This command is now deprecated. ADDRESS WITH can do the same thing, and can also capture the command's stderr.

`POPEN('ls -l', 'lists.')`	`/* LISTS. stem has list */`
`ADDRESS SYSTEM 'ls -l' WITH OUTPUT STEM LISTS.`	`/* same as above */`

POS(needle, haystack [,start]) - (ANSI)

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')	4
POS('to','to be or not to be',10)	14
POS('is','to be or not to be')	0
POS('to','to be or not to be',18)	0

PUTENV(environmentvar=[value]) - (REGINA)

Sets the value of the named system environment variable or deletes it. The existing value is returned if the environment variable has a value or if this environment variable is not defined, a nullstring is returned.
If no value is supplied, the system environment variable is deleted. This is the only mechanism available to delete a system environment variable.

PUTENV('FRED=hello')	'' /* If unset */
PUTENV('FRED=')	'hello' /* variable deleted */

QUALIFY([streamid]) - (ANSI)

Returns a name for the *streamid*. The two names are currently associated with the same resource and the result of this function may be more persistently associated with that resource.

QUALIFY('../mypath/fred.the')	'/home/mark/mypath/fred.the'

QUEUED() - (ANSI)

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) - (ANSI)

RANDOM([min] [,[max] [,seed]]) - (ANSI)

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 positive 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.

If *seed* is specified; (it must be a positive whole number) 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 */`

RANDU([seed]) - (AREXX)

Returns a uniformly-distributed pseudo random number between 0 and 1. The number of digits of precision in the result is always equal to the current Numeric Digits setting. With the choice of suitable scaling and translation values, RANDU()can be used to generate pseudo random numbers on an arbitrary interval.
The optional *seed* argument is used to initialize the internal state of the random number generator.
See also RANDOM()

RANDU()	'0.371902021'
RANDU(45)	'0.873' /*numeric digits 3*/

READCH(file, length) - (AREXX)

Reads the specified number of characters from the given logical file and returns them. The length of the returned string is the actual number of characters read,and may be less than the requested length if, for example, the end-of-file was reached.
See also READLN()

READCH('infile',10)	'a string o'

READLN(file) - (AREXX)

Reads characters from the given logical file into a string until a "newline" character is found. The returned string does not include the "newline".
See also READCH()

READLN('infile')	'a string of chars'

REVERSE(string) - (ANSI)

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]) - (ANSI)

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'

RXFUNCADD(externalname, library, internalname) - (SAA)

Registers the *internalname* in *library* as an external function callable from with the current program by referencing *externalname*. *library* is a REXX external function package in the format of shared library or dynamic link library (DLL). *library* and internalname are case-sensitive. *library* is the **base** name of the shared library or dynamic link library. On platforms that support DLLs, the full

name of the external function package is *library*.**dll**. On Unix environments, the full name of the shared library is **lib***library*.**a** (AIX), **lib***library*.**sl** (HPUX) or **lib***library*.**so** (most other Unixes). External function packages are searched for in the location where shared libraries or DLLs are normally found by the operating system. DLLs are normally located in directories specified in the **PATH** or **LIBPATH** environment variables. Shared libraries are normally searched for in **LD_LIBRARY_PATH** or **LIBPATH** environment variables.

This function returns 0 if the function is registered successfully.

`RXFUNCADD('SQLLoadFuncs','rexxsql','SQLLoadFuncs')`	`0`

RXFUNCDROP(externalname) - (SAA)

Removes the specified externalname from the list of external functions available to be called. This function returns 0 if the function was successfully dropped.

`RXFUNCDROP('SQLLoadFuncs')`	`0`

RXFUNCERRMSG() - (REGINA)

Returns the error message associated with the last call to RXFUNCADD. This function is generally used immediately after a failed call to RXFUNCADD to determine why it failed.

`RXFUNCERRMSG()`	`'rexxsql.dll not found' /* Maybe */`

RXFUNCQUERY(externalname) - (SAA)

Returns 0 if the *externalname* is already registered, or 1 if the *externalname* is not registered.

`RXFUNCQUEURY('SQLLoadFuncs')`	`1 /* Maybe */`

RXQUEUE(command [,queue|timeout]) - (OS/2)

This function interfaces to the Regina internal or external queue mechanism. If OPTIONS INTERNAL_QUEUES is set, all operations on queues are internal to the interpreter.

[C]
: (Create) Request the interpreter or rxstack to create a new named *queue*. If the *queue* name already exists, a new unique queue name is generated. The name of the queue that was created (either the specified queue or the system-generated queue) is returned. All queue names are case-insensitive; i.e. the queue name FRED and fred are the same.

[D]
: (Delete) Deletes the specified *queue*. The default queue; SESSION becomes the current queue.

[G]
: (Get) Returns the current *queue* name.

[S]
: (Set) Sets the current queue name to that *queue* specified. The previously current queue is

returned. It is valid to set a queue name to a queue that has not been created.

[T]

(Timeout) Sets the *timeout* period (in milliseconds) to wait for something to appear on the current queue (as set by RXQUEUE('S', queue)). By default, when a line is read from a queue will a PULL command, it either returns immediately with the top line in the stack, or it will wait for a line to be entered by the user via the process' stdin. If 0 is specified, Regina will wait forever for a line to be ready on the stack.
An error will result if an attempt is made to set a timeout on an internal queue; timeouts only make sense on external queues (ie those with a '@' in them that use the rxstack process).

`RXQUEUE('Create')`	`'S0738280'`
`RXQUEUE('Create','fred')`	`'FRED'`
`RXQUEUE('Create', 'fred')`	`'S88381'`
`RXQUEUE('Get')`	`'S88381'`
`RXQUEUE('Delete','fred')`	`'SESSION'`
`RXQUEUE('Set','fred')`	`'SESSION'`
`RXQUEUE('Timeout',10)`	`'0'`

SEEK(file, offset, ['Begin'|'Current'|'End') - (AREXX)

Moves to a new position in the given logical file, specified as an *offset* from an anchor position. The default anchor is Current. The returned value is the new position relative to the start of the file.

`SEEK('infile',10,'B')`	`'10'`
`SEEK('infile',0,'E')`	`'356' /* file length */`

SHOW(option, [name], [pad]) - (AREXX)

Returns the names in the resource list specified by the *option* argument, or tests to see whether an entry with the specified *name* is available. The currently implemented options keywords are Clip, Files, Libraries, and Ports, which are described below.
Clip. Examines the names in the Clip List.
Files. Examines the names of the currently open logical file names.
Libraries. Examines the names in the Library List,which are either function libraries or function hosts.
Ports. Examine the names in the system Ports List.
If the *name* argument is omitted, the function returns a string with the resource names separated by a blank space or the *pad* character,if one was supplied. If the *name* argument is given, the returned boolean value indicates whether the *name* was found in the resource list. The *name* entries are case-sensitive.
Only the **Files** *option* is valid on all platforms. All other values for *option* are only applicable to the Amiga and AROS ports.

SIGN(number) - (ANSI)

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'

SLEEP(seconds) - (CMS)

Pauses for the supplied number of seconds.

SLEEP(5)	/* sleeps for 5 seconds */

SOURCELINE([lineno]) - (ANSI)

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 a 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]]) - (ANSI)

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* are 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'

STATE(streamid) - (CMS)

Returns 0 if the *streamid* exists, or 1 if it does not. Use STREAM(streamid, 'C', 'QUERY EXISTS') for portability.

STORAGE([address], [string], [length], [pad]) - (AREXX)

Calling STORAGE() with no arguments returns the available system memory. If the address argument is given, it must be a 4-byte string,and the function copies data from the optional *string* into the indicated memory area. The *length* parameter specifies the maximum number of bytes to be copied,and defaults to the length of the string. If the specified length is longer than the string,the remaining area is filled with the *pad* character or nulls('00'x.)
The returned value is the previous contents of the memory area. This can be used in a subsequent call to restore the original contents.
Caution is advised in using this function. Any area of memory can be overwritten,possibly causing a system crash.

STORAGE()	'248400'
STORAGE('0004 0000'x,'The answer')	'question' /* maybe */

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

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. This consists of at least 8 words:

Device Number	Under DOS, Win32, OS/2, this represents the disk number, with 0 being Drive A.
Inode Number	Under DOS, Win32, OS/2, this is zero.
Permissions	User/Group/Other permissions mask. Consists of 3 octal numbers with 4 representing read, 2 representing write, and 1 representing execute. Therefore a value of 750 is read/write/execute for user, read/execute for group, and no permissions for other.

Number Links Under DOS, Win32, OS/2, this will always be 1.

User Name The owner of the stream. Under DOS, Win32, OS/2, this will always be “USER”.

Group Name The group owner of the stream. Under DOS, Win32, OS/2, this will always be “GROUP”.

Size Size of stream in bytes.

Stream Type One or more of the following:
RegularFile a normal file.
Directory a directory.
BlockSpecial a block special file.
FIFO usually a pipe.
SymbolicLink a symbolic link. If the stream is a symbolic link, the the details returned are details about the link, not the file the link points to.
Socket a socket
SpecialName a named special file.
CharacterSpecial a character special file.

[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 writable 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.

SEEK READ CHAR returns the current read position of the open stream expressed in characters.

SEEK READ LINE returns the current read position of the open stream expressed in lines.

SEEK WRITE CHAR returns the current write position of the open stream expressed in characters.

SEEK WRITE LINE returns the current write position of the open stream expressed in lines.

SEEK SYS returns the current read position of the open stream as the operating 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 modification of the stream. The

format of the string returned is YYYY-MM-DD HH:MM:SS.

You can use **POSITION** in place of **SEEK** in the above options.

[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.

[SEEK *position* READ|WRITE [CHAR|LINE]]

Positions the file's read or write pointer in the file to the specified *position*. **SEEK** is a synonym for **POSITION.**

position A position can be of the following forms. [*relative*]*offset*.

relative can be one of:

= The file pointer is moved to the specified *offset* relative to the start of the file. This is the default.

< The file pointer is moved to the specified *offset* relative to the end of the file.

\- The file pointer is moved backwards relative to the current position.

\+ The file pointer is moved forwards relative to the current position.

offset is a positive whole number.

READ	The read file pointer will be positioned.
WRITE	The write file pointer is positioned.
CHAR	The *offset* specified in *position* above is in terms of characters.
LINE	The *offset* specified in *position* above is in terms of lines.

Assume a file; '/home/mark/myfile' last changed March 30th 2002 at 15:07:56, with 100 lines, each line 10 characters long, and the following command executed in sequence.

STREAM('myfile','C','QUERY EXISTS')	'/home/mark/myfile'
STREAM('myfile','C','QUERY SIZE')	1100
STREAM('myfile','C','QUERY TIMESTAMP')	2002-03-30 15:07:56
STREAM('myfile','C','QUERY DATETIME')	03-30-02 15:07:56
STREAM('myfile','D')	
STREAM('myfile','S')	UNKNOWN
STREAM('myfile','C','QUERY SEEK READ')	
STREAM('myfile','C','OPEN READ')	READY:
STREAM('myfile','D')	
STREAM('myfile','S')	READY
STREAM('myfile','C','QUERY SEEK READ')	1
STREAM('myfile','C','CLOSE')	UNKNOWN
STREAM('myfile','C','STATUS')	
STREAM('myfile','C','FSTAT')	773 35006 064 1 mark mark 1100 RegularFile
STREAM('myfile','C','READABLE')	1
STREAM('myfile','C','WRITABLE')	1
STREAM('myfile','C','EXECUTABLE')	0
STREAM('myfile','C','??')	

STRIP(string [,[option] [,char]]) - (ANSI)

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]]) - (ANSI)

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*]) - (ANSI)

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) - (ANSI)

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]]]) - (ANSI)

Returns a string containing information about the local 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. (ANSI)

[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. (ANSI)

[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. (ANSI)

[J]

(No idea) Returns the number of seconds of CPU time the currently running process has currently used. The output will have no leading zeros, and will have 6 decimal places. (Regina Extension)

[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. (ANSI)

[M]

(Minutes) Returns the number of complete minutes since midnight, in a format having no

leading spaces or zeros. (ANSI)

[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. (ANSI)

[O]

(Offset) Returns the number of microseconds between UTC time and local time. This option was added with the ANSI Standard. (ANSI)

[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. (ANSI)

[S]

(Seconds) Returns the number of complete seconds since midnight, in a format having no leading spaces or zeros. (ANSI)

[T]

(*time_t*) Returns the current date/time in UNIX *time_t* format. *time_t* is the number of seconds since January 1st 1970. (Regina Extension)

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 on March 30th 2002, 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'`
`TIME('O')`	`36000000000 /* East Coast Aus */`
`TIME('J)`	`5.342000 /* Maybe */`

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: **CHLMNST.** The default value for *option_in* is **N**.
When a time is converted to format **T**, the returned value is the input time for the current date.

`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]) - (ANSI)

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 [,[tableout] [,[tablein] [,padchar]]]) - (ANSI)

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')`	`'  '`
`TRANSLATE('FooBar','abfor',,'#')`	`'######'`

TRIM(string) - (AREXX)

Removes trailing blanks from the string argument. A more portable option is to use the Trailing option of the STRIP BIF.

`TRIM(' abc ')`	`' abc'`

TRUNC(number [,length]) - (ANSI)

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'

UNAME([option]) - (REGINA)

Returns details about the current platform. This function is basically a wrapper for the Unix command; uname. Valid values for *option* are:

[A]
(All) The default. Returns a string with the all following option values. Equivalent to: UNAME('S') UNAME('N') UNAME('R') UNAME('V') UNAME('M').

[S]
(System) The name of the operating system.

[N]
(Nodename) The name of the machine.

[R]
(Release) The release of the operating system.

[V]
(Version) The version of the operating system.

[M]
(Machine) The machine's hardware type.

Example running Linux Redhat 6.1 on 'boojum', Athalon K7

UNAME('S')	Linux
UNAME('N')	boojum
UNAME('R')	2.2.12.-20
UNAME('V')	#1 Mon Sep 27 10:40:35 EDT 1999
UNAME('M')	i686

Example running Windows NT 4.0 on 'VM_NT', Intel Pentium

UNAME('S')	WINNT
UNAME('N')	VM_NT
UNAME('R')	0
UNAME('V')	4
UNAME('M')	i586

UNIXERROR(errorno) - (REGINA)

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)*	

UPPER(string [,start [,length [,pad]]]) – (AREXX/REGINA)

Translates the substring of *string* that starts at *start*, and has the length *length* to upper case. *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.

If a specific locale is set (via the -l switch), then the string is set to the correct uppercase values based on that locale.

While this BIF is an AREXX BIF, it is not necessary to have OPTIONS ANSI_BIFS set to use it. The optional arguments are Regina extensions.

UPPER('One Fine Day')	'ONE FINE DAY'
UPPER('fred', 2)	'fRED'
UPPER('fred', 1, 1)	'Fred'
UPPER('fred',1, 10, '*')	'FRED******'

USERID() - (REGINA)

Returns the name of the current user. A meaningful name will only be returned on those platforms that support multiple users, otherwise an empty string is returned.

USERID()	'mark' /* Maybe */

VALUE(symbol [,[value], [pool]]) - (ANSI)

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.

The valid values of *pool* in Regina are one of **ENVIRONMENT**, **SYSTEM**, **OS2ENVIRONMENT**, or *pool* can be a number representing the call level of the current procedure, with the first level being 1. It is therefore possible to get and set the value of a variable in a higher call level procedure from the current one without the need to **EXPOSE** the variable. This and the **POOLID**() **BIF** which returns the current call level are Regina extensions.

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]]) - (ANSI)

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 not 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')`	`'0'`
`VERIFY('foobar','barfo','M')`	`'1'`
`VERIFY('foobar','fob','N')`	`'5'`
`VERIFY('foobar','barf','N',3)`	`'3'`
`VERIFY('foobar','barf','N',4)`	`'0'`

WORD(string, wordno) - (ANSI)

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) - (ANSI)

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) - (ANSI)

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]) - (ANSI)

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) - (ANSI)

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

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

WRITECH(file, string) - (AREXX)

Writes the string argument to the given logical file. The returned value is the actual number of characters written.

WRITECH('outfile','Testing')	'7'

WRITELN(file, string) - (AREXX)

Writes the string argument to the given logical file with a "newline" appended. The returned value is the actual number of characters written, including the“newline” character(s).

WRITELN('outfile','Testing')	'8' /* Unix */
WRITELN('outfile','Testing')	'9' /* DOS */

XRANGE([start] [,end]) - (ANSI)

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) - (ANSI)

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.

`X2B('')`	`''`
`X2B('466f6f 426172')`	`'010001100110111101101111010000100110000101110010'`
`X2B('46  6f  6f')`	`'010001100110111101101111'`

X2C(hexstring) - (ANSI)

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]) - (ANSI)

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'

4.3 Implementation specific documentation for Regina

4.3.1 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

4.3.2 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]`

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]

This is the default value if you do not specify an option. 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.

4.3.3 REXX VMS Interface Functions

```
F$CVSI

F$CVTIME

F$CVUI

F$DIRECTORY

F$ELEMENT

F$EXTRACT

F$FAO

F$FILE_ATTRIBUTES

F$GETDVI

F$GETJPI

F$GETQUI

F$GETSYI

F$IDENTIFIER

F$INTEGER

F$LENGTH

F$LOCATE
```

```
F$LOGICAL

F$MESSAGE

F$MODE

F$PARSE

F$PID

F$PRIVILEGE

F$PROCESS

F$SEARCH

F$SETPRV

F$STRING

F$TIME

F$TRNLNM

F$TYPE

F$USER
```

5 ZOC REXX Extensions

5.1 ZOC-REXX Commands/Functions Overview

The help text below lists the ZOC extensions to the REXX scripting language. REXX in itself is a full featured programming language with native programming elements like variables, loops, etc. In the examples below, the native language elements are show in uppercase like CALL, SAY, etc. and you will find an overview here. ZOC then offers terminal emulation specific extensions to this language.

The names of ZOC extensions always begin with Zoc They are like built in procedures or functions and for many of them it is necessary to provide one or more arguments.

Generally there are two types of ZOC-commands:
Commands that return a value (these are also called Functions) and commands that don't.

ZOC-commands that do not return a value are called with the procedure call syntax:

```
CALL <cmd-name> <arguments>
```

Functions are called with the function call syntax: <result-var>= <cmd-name>(<arguments>)

However, it is possible to call functions (commands which return values) with the procedure style call if you are not interested in the result code; in other words,

```
CALL ZocDownload "ZMODEM", "\FILES\DOWNLOAD"
```

and

```
error= ZocDownload("ZMODEM", "\FILES\DOWNLOAD")
```

are both legal.

In the list below, functions are indicated by the use of brackets.

5.2 ZocAsk([<title> [, <preset>]])

Show a text input window and read text from user. If the second argument (preset) is provided, the entry field will be preset with this value.

Example:

```
answer= ZocAsk("What is the best terminal?", "ZOC")
IF answer="ZOC" THEN ...
```

See also: ZocDialog, ZocAskPassword, ZocAskFilename, ZocAskFoldername, ZocRequest, ZocRequestList

5.3 ZocAskPassword([<title>])

Same as the ZocAsk command, except that it is intended to enter passwords, i.e. the entry field shows typed characters as dots and you cannot preset the field with a default value.

Example:

```
pw= ZocAskPassword("What's your password?")
IF pw=="secret" THEN ...
```

5.4 ZocAskFilename(<title> [, <preselected file>])

Display a file selection window and return the filename. If the file dialog is cancelled, the string `##CANCEL##` is returned.

Example:

```
file= ZocAskFilename("Select file to upload", "*.ZIP")
IF file\="##CANCEL##" THEN DO
    CALL ZocUpload "ZMODEM", file
END
```

See also: ZocAsk, ZocDialog, ZocFilename, ZocAskFilenames, ZocAskFoldername, ZocListFiles

5.5 ZocAskFilenames(<title> [, <preselected file> [, <delimiter>]])

Display a window that allows selection of multiple files and return the filenames separated by a space character. If the file dialog is cancelled, the string `##CANCEL##` is returned.

The items can then be extracted from the list by using the `ZocString("WORD", idx)` function. If you expect filenames to contain space characters, you need to supply a different delimiter and use the `ZocString("PART", idx, "|")` function instead.

Example:

```
files= ZocAskFilenames("Select file to process", "*.ZIP",
"|")

howmany= ZocString("PARTCOUNT", files, "|")
DO i=1 TO howmany
    name= ZocString("PART", files, i, "|")
    SAY i||". NAME= "||name
END
```

See also: ZocFilename, ZocAskFilename, ZocAskFoldername, ZocListFiles, ZocMessageBox, ZocRequest, ZocRequestList

5.6 ZocAskFoldername(<title> [, <preselected folder>])

Display a folder selection dialog and return the name of the selected folder. If the dialog is cancelled, the string `##CANCEL##` is returned.

Example:

```
folder= ZocAskFoldername("Select Folder")
IF folder\="##CANCEL##" THEN DO
    SAY folder
END
```

See also: ZocFilename, ZocAskFilename, ZocAskFilenames

5.7 ZocBeep [<n>]

Beep n times.

Example:

```
CALL ZocBeep 2
```

5.8 ZocClipboard <subcommand> [, <writestring>]

Performs a clipboard function for one of the following subcommands:

READ	Returns the current content of the clipboard
WRITE	Writes the string from the 2nd parameter to the clipboard

Example:

```
clip= ZocClipboard("READ")
newclip= clip||ZocCtrlString("^M^J Zoc was here!")
CALL ZocClipboard "WRITE", newclip)
```

5.9 ZocClearScreen

Clears the screen and resets the emulation to its initial state.

Example:

```
CALL ZocClearScreen
```

5.10 ZocCommand <subcommand>

Performs a function for one of the following subcommands:

CLS	Clear screen.
CLEARSCROLLBACK	Clear scrollback buffer.
CANCELCONNECT	Cancel a connect request that is currently in progress.
LOADGLOBALCOLORS	Reload the global color table from file (default file name or 2nd parameter).
SAVEPROGRAMSETTINGS	Permanently stores changes, which were made via the ZocSetProgramOption command.
SAVESESSIONPROFILE	Stores changes made via the ZocSetSessionOption to the current session profile file (see also ZocSaveSessionProfile)
SETMARKEDAREA	Marks an area on screen. After the subcommand provide the word BLOCK or STREAM, followed by coordinates x1,y1 and x2,y2
SENDBREAK	Sends a modem break signal (Serial/Modem connections only).

Example:

```
CALL ZocCommand "SAVESESSIONPROFILE"
CALL ZocCommand "SETMARKEDAREA","BLOCK",0,0,1,79
```

See also: ZocMenuEvent

5.11 ZocConnect [<address>]

Connect to a host via telnet, modem, ssh etc. or read the address to connect to from the user if the parameter is omitted.

The connection method will be the one that is active in the current session profile. Alternately a connection method can be selected in the script via ZocSetDevice.

If the address is an SSH host, you can pass the username and password to the host in the form *CALL ZocConnect "user:pass@ssh.somedomain.com"*

Example:

```
CALL ZocSetDevice "Secure Shell"
CALL ZocConnect "harry:alohomora@192.168.1.1:10022"
```

Example:

```
CALL ZocSetDevice "Telnet"
CALL ZocConnect "server.hogwarts.edu"

Call ZocTimeout 20
x= ZocWait("Login:")
IF x=640 THEN SIGNAL waitfailed /* login prompt not received
*/
CALL ZocSend "harry^M"
x= ZocWait("Password:")
IF x=640 THEN SIGNAL waitfailed /* password prompt not
received */
CALL ZocSend "secret^M"

/* At this point we are logged in */

waitfailed:
EXIT
```

See also: ZocConnectHostdirEntry, ZocSetDevice, ZocDisconnect, ZocGetInfo("ONLINE")

5.12 ZocConnectHostdirEntry <name>

Makes a connection based on the details of an entry in the ZOC host directory (the host directory entry should not have a Login REXX file assigned to it).

Example:

```
CALL ZocConnectHostdirEntry "Webhost 03"
```

See also: ZocConnect, ZocDisconnect, ZocGetInfo("ONLINE")

5.13 ZocCtrlString(<text>)

This function converts a string containing control codes into a string where the control codes are converted into their respective values.

Example:

```
crlf= ZocCtrlString("^M^J") /* results in two byte string
hex"0D0A" */
```

See also: ZocCtrlString

5.14 ZocDelay [<sec>]

Wait a given time in seconds or wait 0.2 seconds if the parameter is omitted.

Example:

```
CALL ZocDelay 4.8
```

5.15 ZocDeviceControl <string>

This rather arcane command performs an operation that is specific to the current connection type (e.g. to return the signal states of the COM during a Serial/Direct type connection).

Possible control commands for each communication method are described in ZOC Devices.

Example:

```
state= ZocDeviceControl("GETRS232SIGNALS")
```

See also: ZocSetDevice

5.16 ZocDialog <subcommand> [, <parameter>]

Performs a dialog related function:

LOAD	Shows a user defined dialog window. The 2nd parameter is a combination of the the name of the dialog together with the name of the file containing the dialog template (the file name is either a fully qualified file name or the name of a file located in the same folder as

	the currently running script file).
SHOW	Shows a user defined dialog window. There can be an optional 2nd parameter as described above for the LOAD command. In this case, the SHOW command will include the LOAD operation.
GET	Returns the value of one of the dialog elementes (e.g. the text which the user had typed into an entry field or the state of a checkbox or radiobutton).
SET	Set the value of one of the dialog elementes, e.g. the text which the will initially be shown in an entry field or the description of an item. With a checkbox or radiobutton, the values ##ON## or ##OFF## will also change the initial state.

For details about how dialog templates are built, see ZocDialog Templates.

Example:

```
dlgrc= ZocDialog("SHOW", "MAIN@test.dlg")
IF dlgrc=="##OK##" THEN DO
  name= ZocDialog("GET", "ED1")
  SAY "Hallo "||name
  SAY ZocDialog("GET", "CB1")
  SAY ZocDialog("GET", "P1")
  SAY ZocDialog("GET", "P2")
END
EXIT
```

See also: ZocAsk, ZocAskPassword, ZocMessageBox, ZocRequest, ZocRequestList

5.17 ZocDisconnect

Disconnects the current connection. Same as `ZocCommand "DISCONNECT"`.

Example:

```
CALL ZocDisconnect
```

See also: ZocConnect, ZocConnectHostdirEntry

5.18 ZocDownload(<protocol>[:<options>], <file or dir>)

Download one or more files using a file transfer protocol.

The first parameter is the name of a file transfer protocol (as listed in ZOC's Options→Session Profile→File Transfer dialog).

The exact nature of the second parameter varies depending on the transfer type (see note below).

For a discussion of the protocol options, please see the ZocUpload command further down in this list.

Depending on success or failiure, the function returns the string `##OK##` or `##ERROR##`

Example:

```
CALL ZocSetSessionOption "TransferAutoStart=no"
ret= ZocDownload("ZMODEM", "C:\ZOC\INFILES")
IF ret=="##ERROR##" THEN DO
    CALL ZocBeep 5
    SAY "Download failed."
END
```

Note: The second parameter varies depending on the file transfer type:
XMODEM: Local destination filename.
YMODEM: Local destination folder.
ZMODEM: Local destination folder.
SCP: Remote source file, e.g. `/var/log/somefile.txt`.

Note: If you have Auto Transfer enabled in the Options→Session Profile→File Transfer options, and if the remote host starts the transfer before ZOC-REXX processes the ZocDownload command, then two download windows will come up. So you need to make sure you are issuing ZocDownload() before the host starts or make sure that Auto Transfer option is disabled.

Note: If the file has a name that is set for download to the alternate path (in Options→Session Profile→File Handling), the directory parameter is ignored.

See also: ZocUpload

5.19 ZocDoString(<commandstring>)

Pass an Action Code to ZOC for processing.

You can obtain such strings by temporarily mapping something to a key via Options→Keyboard Mapping Profiles and then copying the result from the key's mapping assistant.

Example:

```
CALL ZocDoString "^EXEC=notepad.exe"
```

See also: ZocMenuEvent, ZocShell, ZocSendEmulationKey

5.20 ZocEventSemaphore(<subcommand>[, <signal-id>])

This function can be used to synchronize and exchange signals between multiple REXX scripts (it has no use within a single script). To facilitate this, the function offers a signaling mechanism with 16 signal slots for which other scripts can wait.

Possible **<subcommands>** are:

RESET	Sets the state of this semaphore to not-fired and clears the signal counter.
FIRE	Sets state to signaled, increases the counter and releases all waiting scripts.
TEST	Returns the number of received signals (i.e. FIRE commands) since the last reset.
WAIT	Wait for a signal. If the semaphore was already fired, the command will return immediately, resetting the signal (see above) returning the number of signals which happened since the last reset (max 255). If the semaphore has not yet been signaled (fired), the function will block and wait for a signal or it will return after a timeout (set via ZocZimeout returning a code of 640 (the behavior of this function is very similar ZocWait.
<signal-id>	An optional number [1..15] to access/use a different semaphore (default is 0).

Example:

```
Call ZocEventSemaphore "RESET"
/* do some work */
...
/* wait until another script fires the signal */
ret= ZocEventSemaphore("WAIT")
IF ret=640 THEN DO
   /* signal was received */
END
```

See also: ZocTimeout, ZocWait, ZocGlobal

5.21 ZocFilename(<command>[, <options>])

This group of commands offers filename operations.

COMBINE <path>[, <path2>], <file>	Combines filename parts to a full filename. If <file> is already a fully qualified name, <path> and optional <path2> are ignored.

EXISTS <filename>	Returns ##YES## bzw. ##NO##, depending on the existence
GETFILE <filename>	Returns the filename part of a full file descriptor.
GETPATH <filename>	Returns the directory part of a full file descriptor.
GETSIZE <filename>	Returns the size of a file.
GETVOLUMELABEL <driveletter>	Returns the drive label for a drive, e.g. "C:" (Windows only).
ISFOLDER <pathname>	Returns ##YES## or ##NO##, depending on if the given path refers to an existing folder.
RESOLVE <string>	Resolves one of ZOC's special file/path placeholders like %ZOCFILES% or %USERPROFILE%.
WRITEACCESS <filename>	Returns ##YES## or ##NO##, depending on if a file can be written.

Example:

```
workdir= ZocGetInfo("WORKDIR")
datadir= ZocFilename("RESOLVE", "%ZOCFILES%")

fullfile= ZocAskFilename("Choose File", workdir)
file= ZocFilename("GETFILE", fullfile)
path= ZocFilename("GETPATH", fullfile)
```

```
file2= file||".tmp"
target= ZocFilename("COMBINE", path, file2)
IF ZocFilename("EXISTS", target)=="##YES##" THEN DO
    CALL ZocMessageBox "Can't overwrite file "||file2
    EXIT
END
```

5.22 ZocFileCopy(<source filename>, <destination>)

Copy a file to a new destination which can either be a file name or folder name.

Wildcards like * or ? are not supported int the source file (see ZocListFiles).

Example:

```
CALL ZocFileCopy "Z:\SALES.DAT", "Z:\SALES.BAK"

ok= ZocFileCopy("C:\DATA\USERFILE.TMP", "C:\BACKUP")
IF ok="##OK##" THEN EXIT
```

See also: ZocFilename, ZocFileDelete, ZocFileRename, ZocListFiles, ZocShell

5.23 ZocFileDelete(<filename>)

Deletes a file. The filename may not contain wildcards (* or ?, see ZocListFiles).

The function returns ##OK## or ##ERROR##

Example:

```
ok= ZocFileDelete("C:\DATA\USERFILE.TMP")
IF ok="##OK##" THEN EXIT

filename= ZocFilename("COMBINE", "%ZOCFILES%", "rexx.log")
CALL ZocFileDelete filename
```

See also: ZocFilename, ZocFileCopy, ZocFileRename, ZocListFiles, ZocShell

5.24 ZocFileRename(<oldname>, <newname>)

Renames a file. The renamed file will always remain in the same folder as the original file. Filenames may not contain wildcards * or ?, see ZocListFiles.

The function returns ##OK## or ##ERROR##

Example:

```
ret= ZocFileRename("C:\DATA\USERFILE.TMP",
"C:\DATA\USERFILE.TXT")
```

See also: ZocFilename, ZocFileCopy, ZocFileDelete, ZocListFiles, ZocShell

5.25 ZocGetHostEntry(<name>, <key>)

Retrieves the key-value pair for a ZOC host directory entry (see ZocSetHostEntry or ZocSetSessionOption commands for more information about such key-value pairs).

Example:

```
pair= ZocGetHostEntry("ZOC Support BBS", "connectto")
PARSE VALUE pair WITH key'="'val'"'
CALL ZocConnect val
```

5.26 ZocGetInfo(<what>)

Depending on the parameter, this function can return various bits of information about the current environment and ZOC session.

COMPUTERNAME	The name of the computer on which ZOC is running.
CONNECTEDTO	The host name, ip, or phone number to which ZOC is connected.
CURRENTDEVICE	The name of the currently active communication method, e.g. Telnet.
CURRENTEMULATION	The name of the currently active emulation, e.g. Xterm.
CURRENTLOGFILENAME	Current file name for logging (without path).
CURRENTSCRIPTNAME	Filename and path of the main script which is currently executed (this will not return sub-scripts which are executed via CALL from inside another script).
CURRENTSESSIONPROFILENAME	The filename and full path of the current session profile.
CURSOR-X	The x-position of the cursor (starting with zero).
CURSOR-Y	The y-position of the cursor (starting with zero).
DESKTOPSIZE	The net size of the Windows/Mac OS X desktop in pixels (excluding taskbar, dock, etc.)
DOWNLOADDIR	The default drive and directory for downloads.

EXEDIR	The drive and directory in which ZOC has been installed.
LASTDOWNLOADEDFILE	The name and path of the last downloaded file.
ONLINE	Information if ZOC is currently connected to a host: `##YES##, ##NO##, ##UNKNOWN##`
OSYS	Returns a string which indicates the operating system and OS version under which ZOC is running, e.g. `Windows XP`.
OWNIP	Returns the computer&apos.s IPV4 address in the local LAN or WLAN.
PROCESSID	ZOC's system process id.
SCREENHEIGHT	Height (number of lines) of the terminal.
SCREENWIDTH	Width (number of characters) of the terminal.
TRANSFER	Information if a file transfer is currently active: `##YES##, ##NO##`
UPLOADDIR	The default drive and directory for uploads.
USERID	The ID of the currently logged in user.
VERSION	The current ZOC version, e.g. `6.24`
VERSIONEX	The current ZOC version including beta version (if any), e.g. `6.24b`
WINPOS	A string indicating the position and size of the program window on the screen.
WORKDIR	ZOC's working directory (containing host directory, options, etc).

Example:

```
ver= ZocGetInfo("VERSIONEX")
SAY "ZOC "||ver

CALL ZocTimeout 30
timeout= ZocWait("Password")
IF timeout=640 | ZocGetInfo("ONLINE")<>"##YES##" THEN DO
    SIGNAL PANIC /* disconnected! */
END
```

5.27 ZocGetProgramOption(<key>)

Retrieves the key-value pair for a ZOC program option (see ZocGetSessionOption and ZocSetProgramOption for more info about key-value paírs).

Example:

```
pair= ZocGetProgramOption("DisconEndProg")
PARSE VALUE pair WITH key"="value
```

```
IF value=="yes" THEN DO
    SAY "ZOC will terminate after this session."
END
```

See also: ZocGetSessionOption, ZocSetSessionOption, ZocSetProgramOption

5.28 ZocGetScreen(<x>,<y>,<len>) or ZocGetScreen("<alias>")

The ZocGetScreen() function can be used to return characters that are currently displayed in ZOC's terminal window. It returns <len> characters beginning from position <x>,<y> (zero based). If it reaches the right margin, it continues on the next line without adding a CR or LF.

There are also a few short versions for common combinations of x,y and len:
ALL: The whole screen (position 0/0, length SCREENWIDTH*SCREENHEIGHT)
LEFTOFCURSOR: The text left of the cursor (position 0/CURSOR-Y, length CURSOR-X)
CURRENTLINE: The whole line where the cursor is (position 0/CURSOR-Y, length SCREENWIDTH

Example:

```
curline= ZocGetScreen("LEFTOFCURSOR")
SAY "^M"
SAY "The text before the cursor was: "||curline
```

Example:

```
width= ZocGetInfo("SCREENWIDTH")
posy= ZocGetInfo("CURSOR-Y")
screenpart= ZocGetScreen(0,0, posy*width)
IF POS("#", screenline)=0 THEN DO
    SAY "There is no hash character on screen above the
cursor."
END
```

Example:

```
cx= ZocGetInfo("SCREENWIDTH")
cy= ZocGetInfo("SCREENHEIGHT")

-- loop through all lines on screen
DO y= 0 TO cy-1
    line= ZocGetScreen(0,y, cx)
    -- check each line for some text
    IF POS("**SUCCESS**", line)>=1 THEN DO
        Call ZocMessageBox "Found **SUCCESS* on screen in line
"||y
    END
END
```

See also: ZocGetInfo("CURSOR-X"), ZocGetInfo("CURSOR-Y"), ZocGetInfo("SCREENWIDTH"), ZocGetInfo("SCREENHEIGHT")

5.29 ZocGetSessionOption(<key>)

Retrieves the key-value pair for a ZOC option from the session profile (see the ZocSetSessionOption command for more details).

Example:

```
pair= ZocGetSessionOption("Beep")
PARSE VALUE pair WITH key"="value
IF value=="no" THEN DO
    SAY "The beep option is turned off."
END
```

Example:

```
pair= ZocGetSessionOption("EnqString")
PARSE VALUE pair WITH key'="'value'"'
SAY "The configured answer to enq is: "||value
```

See also: ZocSetSessionOption, ZocGetProgramOption, ZocSetProgramOption

5.30 ZocGlobal(<operation>, [<options>])

This group of functions allows permanent storage of values in a file based data pool. This can be used to remember values across various script runs (the operations are atomic and they are protected against concurrent access).

Possible operations are:

SELECT <name>	Select a different global pool. The name will be used to create a file name in the form `<name>Global.ini` in the user data folder or it can point to a different location, e.g. `C:\data\mypool` (which will then also belcome `mypoolGlobal.ini`).
INIT	Clear all values in the global space.
GET <name>	Returns the value with the given name.
PUT <name>, <value>	Stores a value in the storage pool under a given name.
SET <name>, <value>	Same as PUT

Example:

```
CALL ZocGlobal "SELECT", "MyValStore"
x= ZocGlobal("GET", "LASTX")
...
CALL ZocGlobal "SET", "LASTX", x
```

5.31 ZocKeyboard(<command> [, <timeout>])

This function allows a REXX script to read keystrokes from the terminal window. It supports the subcommands *LOCK*, *UNLOCK* and *GETNEXTKEY*

LOCK	Lock the keyboard and prevent user input.
UNLOCK	Unlock the keyboard from a previous *LOCK* state.
GETNEXTKEY	Wait for the next keystroke and return it. You can also specify a timeout in seconds and if successful, GETNEXTKEY will return a string in the form *char*|*scancode*|*shift*|*ctrl*|*alt*. char: two byte hex number representing the ascii code of the key. scancode: The physical scan code from the keyboard (the scan code can be used to identify functional keys such home, del, f1, f2, etc.). shift, ctrl and alt are either 0 or 1 indicating if they were held down when the primary key was pressed.

The example below shows how the subcommands are used and how the possible result can be split into its parts and displayed in a user friendly form.

Example:

```
CALL ZocKeyboard "UNLOCK"
ret= ZocKeyboard("GETNEXTKEY", 30)
PARSE VALUE ret WITH hexkey"|"scan"|"shift"|"ctrl"|"alt
key= X2C(hexkey)
SAY "You pressed hex/key: "hexkey"/"key
SAY "Scan code: "scan
SAY "Shift/Ctrl/Alt states: "shift"/"ctrl"/"alt
```

5.32 ZocLastLine()

This is a function and returns the last line of text that was received from the point the last ZocWait/ZocWaitMux/ZocWaitLine command was issued to the point when it successfully returned

Example:

```
CALL ZocSend "ATZ^M"
timeout= ZocWaitMux("OK", "ERROR")
IF timeout\=640 & THEN DO
    IF ZocLastLine()="OK" THEN SIGNAL error
    CALL ZocConnect "555 3456"
END
```

Note: In many cases, using ZocReceiveBuf instead will be the more flexible choice. ZocGetScreen("LEFTOFLINE") often also provides a similar result.

Note: Also see the examples in the description for ZocWaitLine

5.33 ZocListFiles(<path\mask> [, <separator>])

The ZocListFiles function will retrieve a list of filenames for a directory.

The first parameter is a directory name and wildcard mask (e.g. `"c:\data\*.*"`).

The function will return a string which contains the number of files and the file names separated by space characters, e.g. `"3 download.zip sales.txt foobar.fil"`

This allows easy access to the parts of the string via REXX's WORD function (see example below). If you expect filenames to contain space characters you can provide a different list separator as the second parameter. E.g. a separator of "|" will return the string `"3 download.zip|sales.txt|foobar.pdf"`. In this case you can use `ZocString("PART", purelist, i, "|")` to extract the file names.

Note: The number of filenames returned is limited to 128 and the maximum length of the total string returned is 4096.

Example:

```
files= ZocListFiles("C:\TEMP\*")

howmany= WORD(files, 1)
SAY "Number of Files:" howmany

purelist= SUBSTR(files, LENGTH(howmany)+2)
DO i=1 TO howmany
    SAY "File " i "=" WORD(purelist, i)
END
```

See also: ZocFilename, ZocGetFilename, ZocGetFilenames, ZocGetFolderName, ZocString

5.34 ZocLoadKeyboardProfile [<zkyfile>]

Loads and activates a keyboard profile (*.zky file).

Example:

```
CALL ZocLoadKeyboardProfile "Alternate.zky"
```

5.35 ZocLoadSessionProfile <optsfile>

Loads and activates a session profile file (*.zoc).

Example:

```
CALL ZocLoadSessionProfile "Zoc4Linux.zoc"
```

5.36 ZocLoadTranslationProfile [<ztrfile>]

Loads and activates a character translation profile (*.ztr).

Example:

```
CALL ZocLoadTranslationProfile "7bitgerman.ztr"
```

5.37 ZocMath(<function>, <arg>[, <arg2>])

ZocMath calculates the math function based on the arguments. Valid functions are sin, cos, tan, asin, acos, sqrt, todeg, torad, bitand, bitor, bitxor.

Example:

```
angle = 270
anglerad = ZocMath("torad", angle)
sinresult = ZocMath("sin", anglerad)
lowbits = ZocMath("bitand", 175, 15)
hibits = ZocMath("bitand", X2D(A5), X2D(F0))
```

5.38 ZocMenuEvent(<menu text> [, <file>])

Perform a function from the ZOC menu. The <menu text> is a text that matches an entry in the ZOC menu. The <file> parameter is an optional file name which some menu events accept rather than prompting the user for a file.

Example:

```
CALL ZocMenuEvent "Paste (no line breaks)"
CALL ZocMenuEvent "Edit REXX Script", "test.zrx"
```

5.39 ZocMessageBox(<text> [, <mode>])

Display a message box with the given text (a ^M in the text creates a line break).

Normally an informational message window with an OK button (mode 0) is shown. Mode 1 shows an error message with an OK button. Mode 2 shows a message with a YES and NO button.

The return value is either ##OK## or ##YES## or ##NO##.

Example:

```
CALL ZocMessageBox "Connect Failed!", 1
ret= ZocMessageBox("The operation failed^M^MTry again?", 2)
IF ret=="##YES##" THEN DO
   ...
END
```

See also: ZocAsk, ZocAskPassword, ZocRequest, ZocRequestList

5.40 ZocNotify <text> [, <duration>]

Display a small floating message at the center of the window. If duration is given, controls the time in milliseconds which the window will stay on screen.

Example:

```
CALL ZocNotify "Hello World!", 1500
```

5.41 ZocPlaySound <file>

Plays a .WAV file.

Example:

```
CALL ZocPlaySound "ka-ching.wav"
```

5.42 ZocReceiveBuf(<buffer size>)

This function makes ZOC collect parts of a session in a memory buffer and returns the previous buffer's contents (if any) as a string.

Initially the buffer has a size of zero, which means that no data is collected. To start data collection you need to call the function with a parameter indicating the size of the next receive buffer. After that, incoming data is added to the buffer until either the buffer is full or until the function is called again. Calling ZocReceiveBuf again will retrieve the buffer's content, reset the content and set a new size for the buffer.

A sequence of calls to ZocReceiveBuf in order to retrieve text from a database will look like this:

Example:

```
/* make a receive buffer of 256 bytes */
CALL ZocTimeout 60

CALL ZocReceiveBuf 256
CALL ZocSend "read abstract^M"
CALL ZocWait "Command>"

/* get the result from the read command and */
/* make a larger buffer to hold the result of */
/* a subsequent command.  */
abst= ZocReceiveBuf(4096)
CALL ZocSend "read contents^M"
CALL ZocWait "Command>"
```

```
/* get the content and discontinue buffering */
cont= ZocReceiveBuf(0)

/* At this point, both variables (abst and cont) will start
with the word "read" and end with the character ">",
containing
whatever data was received between the command and next
prompt.  */
```

Example:

```
/* read the remote environment variables and extract the
TERM= value */

Call ZocReceiveBuf 2048
Call ZocSend "set^M"

/* you will need to wait for the your own prompt here */
Call ZocWait "PROMPT: ~username$"
data= ZocReceiveBuf(0)

/* google for "REXX PARSE COMMAND" to get more details
  on the PARSE command which is used to extract the data */
PARSE VALUE data WITH ."TERM="term .

SAY "The remote term setting is " term
```

Note: If executed via DDE, the ZocReceiveBuf command must be sent as a DdeRequest (rather than DdeExecute)

Note: See also ZocWaitForSeq and ZocString("LINE" ...)

5.43 ZocRegistry(<subcommand>[, <options>])

This group of commands allows access to the Windows registry.

OPEN <basekey>, <name>	Returns a <hkey> handle for access to a part of the registry. Basekey can be *HKEY_CURRENT_USER* or *HKEY_LOCAL_MACHINE*. The value can be used to read/write this part of the registry.
WRITE <hkey>, <value>, <data>	Writes <data> to the part of the registry which is associated with <hkey>. If <data> is provided in the form *"DWORD:nnnn"* the decimal value nnnn will be stored as REG_DWORD, otherwise data will be stored as REG_SZ (string).
READ <hkey>, <value>	Read a value from the registry part <hkey>. If the registry

	value is in REG_DWORD format, the command will return *"DWORD:n"* otherwise the string from the registry will be returned.
ENUM <hkey>, <n>	Returns the <n>th value name from <hkey> or ##ERROR## if no such value exists.
CLOSE <hkey>	Ends access to <hkey>.

Example:

```
hk= ZocRegistry("OPEN", "HKEY_CURRENT_USER", "Software")
IF hk=="##ERROR##" THEN EXIT
CALL ZocRegistry "WRITE", hk, "Test01", "Hello World"
CALL ZocRegistry "WRITE", hk, "Test02", "DWORD:1"
homepath= ZocRegistry("READ", hk, "%ZOC%")
SAY "ZOC installed in "||homepath
i= 0
DO  FOREVER
    x= ZocRegistry("ENUM", hk, i)
    IF x=="##ERROR##" THEN LEAVE
    i= i+1
    SAY "Value named "||x
END

CALL ZocRegistry "CLOSE", hk
EXIT
```

5.44 ZocRequest(<title>, <opt1> [, <opt2> [, <opt3>]])

Displays a dialog window with options and returns a string containing the selected option.

Example:

```
answer= ZocRequest("What do you want?", "Milk", "Honey")
IF answer=="Milk" THEN DO
    ...
END
```

See also: ZocAsk, ZocAskPassword, ZocMessageBox, ZocRequestList

5.45 ZocRequestList(<title>, <opt1> [, ...]])

Displays a dialog window with a list of options and returns the index of the selected option (or -1 for Cancel). If only one option is passed to the function, it is considered as a list of choices separated by vertical bars.

Example:

```
answer= ZocRequestList("Please select!", "Beer", "Wine",
"Whiskey", "Liquor")
IF answer=3 THEN DO
    ...
END

answer= ZocRequestList("Please select!", "Beer|Wine|Whiskey|
Liquor")
IF answer=3 THEN DO
    ...
END
```

See also: ZocAsk, ZocAskPassword, ZocMessageBox, ZocRequest

5.46 ZocRespond <text1> [, <text2>]

Send *text2* whenever *text2* is received while REXX is processing a ZocDelay or ZocWait command.

A maximum of 64 Respond commands can be active simultaneously. <text1> must not contain carriage returns or line feeds.

If *text2* is omitted or empty the response command for <text1> is cleared. If *text1* is empty ("") all responses are cleared.

Example:

```
/* Wait for 'Command' and auto-skip all possibly prompts in
between */
CALL ZocRespond "Enter", "^M"
CALL ZocRespond "More", "^M"
timeout= ZocWait("Command")
/* Clear responders */
CALL ZocRespond "Enter"
CALL ZocRespond "More"
```

The above example waits until the text `Command` is received. While waiting, all `Enter` and `More` prompts are answered automatically by sending Enter. After the Wait is satisfied, the respond commands are cancelled.

5.47 ZocSaveSessionProfile [<optsfile>]

Save the current session profile to file. If the <optsfile> parameter is omitted, ZOC will ask the user

for a filename.

Example:

```
CALL ZocSetSessionOption "JumpScroll=3"
CALL ZocSaveSessionProfile "Fastscroll.zoc"
```

See also: ZocCommand(“SAVESESSIONPROFILE”).

5.48 ZocSend <text>

Sends the given text to the remote host.

Internally the text sending is processed as a series keystrokes rather than sending it directly through the low level communication channel. The send speed is based on the text sending option int Options→Session Profile→Text Sending. If you need a faster, more direct version of this command, please use ZocSendRaw

Also, if the text contains control codes (e.g. `^M` for Enter), are replaced with their respective values. For nearly all emulations these control codes are based on the Control Codes Table Exceptions are the TN3270 and TN5250 emulations, where `^M` is interpreted as Newline/FieldExit, `^I` as Tab and `^Z` as Transmit/Enter

Example:

```
/* send JOE USER<enter>*/
CALL ZocSend "JOE USER^M"
```

Example:

```
/* Unix login Sequence */
CALL ZocWait "login:"
CALL ZocSend "harry^M"
CALL ZocWait "password:"
CALL ZocSend "alohomora^M"
```

Example:

```
/* 3270/5250 example */
Call ZocSetCursorPos 12,5
CALL ZocSend "Freddie"
CALL ZocSendEmulationKey "NewLine"
CALL ZocSend "Elm Street"
CALL ZocSendEmulationKey "Enter"

/* 3270/5250 same as above */
CALL ZocSend "Freddie^MElm Street^Z"
```

See also: ZocSendRaw, ZocSendEmulationKey.

5.49 ZocSendEmulationKey <keyname>

Send the code that represents a special key in the current terminal emulation, e.g. send F17 from a VT220 emulation or Attn unter TN3270.

The key names are described in the Key Names Appendix.

Example:

```
CALL ZocSendEmulationKey "f17" /* Send F17 based on current
emulation */
```

5.50 ZocSendRaw <datastring>

This command sends the data from datastring in untranslated form. The command does not translate control sequences like ^M. If you need to send such codes, you will have to use the REXX string functions like X2C(<hexcode>) or ZocCtrlString to create a corresponding character values (e.g. X2C(0D) for Enter).

ZocSendRaw may be useful if you want to send binary data to a host. For example, if you want to send 42 01 00 05 41 43 (hex) through the communication channel, you can do this as in the 2nd part of the example below.

Example:

```
CALL ZocSendRaw "Login"||X2C(0d) /* Login<enter> */

/* Four times the same result: */
CALL ZocSendRaw X2C(420100054143)
CALL ZocSendRaw "B"||X2C(01)||X2C(00)||X2C(05)||"AC"
CALL ZocSendRaw ZocCtrlString("B^A^@^EAC")
CALL ZocSend "B^A^@^EAC"
```

See also: ZocSend, ZocSendEmulationKey, ZocCtrlString

5.51 ZocSessionTab(<subcommand>, <parameters>)

This function allows a REXX script to access or manipulate session tabs. The subcommand defines the action, the parameters depend on the subcommand.

CLOSEATEXIT

Close the current session tab when the script exits.

Example:

```
CALL ZocSessionTab "CLOSEATEXIT"
```

CLOSETAB

Close the session tab with the given index (zero for the leftmost tab or -1 for the current tab).

Example:

```
CALL ZocSessionTab "CLOSETAB", 2
```

GETCOUNT

Returns the number of session tabs.

Example:

```
howmany= ZocSessionTab("GETCOUNT")
```

GETCURRENTINDEX

Returns the index of the tab in which this script is running.

Example:

```
myidx= ZocSessionTab("GETCURRENTINDEX")
```

GETINDEXBYNAME, <name>

Returns the index of the first tab that has a given title or -1 if none was found.

Example:

```
srvidx= ZocSessionTab("GETINDEXBYNAME", "My Server")
```

GETNAME, <index>

Return the name of the tab with a given index (zero for the leftmost tab) from the 1st parameter. An index of -1 refers to the session in which the script is running.

Example:

```
name= ZocSessionTab("GETNAME", -1)
```

ISCONNECTED, <index>

Returns ##YES## or ##NO## depending on if the tab with the given index has an active connection.

Example:

```
name= ZocSessionTab("ISCONNECTED", 2)
```

MENUEVENT, <index>, <menu>

Performs a command from the ZOC menu in a different session. The <index> parameter indicates the session (see the description of the <index> parameter for GETNAME), the <menu> parameter is the same as for ZocMenuEvent.

Example:

```
CALL ZocSessionTab "MENUEVENT", idx, "Disconnect"
```

RUNSCRIPT, <index>, <title>

Starts a script in a different session. The <index> parameter indicates the session (see description of the <index> parameter for GETNAME). Please note that this will not work for sessions which already have a script running (including the session/script which issues the ZocSessionTab("RUNSCRIPT", ...) command.

Example:

```
CALL ZocSessionTab "RUNSCRIPT", newidx, "configure.zrx"
```

SEND, <index>, <text>

Sends text to the session with the given index (similar to using the ZocSend command).
The <index> parameter indicates the session (for details see description of the <index> parameter for GETNAME).

Example:

```
CALL ZocSessionTab "SEND", 2, "exit^M"
```

SETCOLOR, <index>, <color>

Sets the color of the tab with a given index. The <index> parameter indicates the session (for details see the description of the <index> parameter for GETNAME). The color is a number between 0 and 7.

Example:

```
CALL ZocSessionTab "SETCOLOR", -1, 4
```

SETBLINKING, <index>, <blinkflag>

Activates or deactivates the blinking for the tab with a given index. The <index> parameter indicates the session (for details see the description of the <index> parameter for GETNAME). The <blinkflag> is either 1 or 0.

Example:

```
CALL ZocSessionTab "SETBLINKING", -1, 1
```

SETNAME, <index>, <title>

Sets the name of the tab with a given index. The <index> parameter indicates the session (for details see the description of the <index> parameter for GETNAME).

Example:

```
CALL ZocSessionTab "SETNAME", -1, "This Session"
```

SWITCHTO, <index>

Activate the tab with a given index. The <index> parameter indicates the session(for details see the description of the <index> parameter for GETNAME).

Example:

```
CALL ZocSessionTab "SWITCHTO", 2
```

Example:

```
/* ZocSessionTab sample: send a text to all tabs */

text= ZocAsk("Command to send to all tabs:")
IF text="##CANCEL##" THEN DO
  n= ZocSessionTabs("GETCOUNT")
  SAY n
  DO i=0 TO n-1
    name= ZocSessionTabs("GETNAME", i)
    CALL ZocSessionTabs "SEND", i, text||"^M"
    SAY "Sent "||text||" to "||name
  END
END
```

5.52 ZocSetAuditLogname <filename>

Set the file name for the audit log (a logfile that can not be turned off by the user, also see the setting in the ADMIN.INI file in the program folder) or turn off audit logging with "" (empty string) as a filename.

5.53 ZocSetAutoAccept 1|0

For those connection types that support it (e.g. Telnet), make the communication method accept incoming connections.

Example:

```
CALL ZocSetCursorPos 1,15
CALL ZocSetAutoAccept 1 /* accept calls */
```

5.54 ZocSetCursorPos <row>, <column>

This command moves the cursor to the given position on a TN3270 screen (in other emulations the command is ignored). The positions are 1-based, i.e. the top/left position on screen is 1/1.

Example:

```
CALL ZocSetDevice "Telnet"
CALL ZocSendEmulationKey "Enter"
```

5.55 ZocSetDevice <name> [, <commparm-string>]

Change the connection type (communication device). The name must be one of the names from the list in Options→Session Profile→Connection Type.

The optional commparm-string contains options, which can further tweak the operation of that connection (e.g. setting Telnet-specific options, see ZocSetDeviceOpts for information about how to obtain a commparm-string). If the comm parameters are omitted, ZOC uses the options which are set for the connection type in the currently active session profile.

Example:

```
CALL ZocSetDevice "Telnet"
CALL ZocConnect "bbs.channel1.com"
```

Example:

```
CALL ZocSetDevice "SERIAL/MODEM", "[1]COM3:57600-8N1|9|350"
```

5.56 ZocSetDeviceOpts <parameter-string>

This is a rather arcane command, which sets the options for a communication method (device) directly from REXX. However, since the options strings for the device are not standardized, in order to find a specific parameter string, you will need to set the options manually in the session profile dialog and then query the current connection type's parameter string by pressing Shift+Ctrl+F10 in ZOC's main window.

Let us assume you want to start a modem session on COM3 with 57600 bps, RTS/CTS and Valid-CD active and a break time of 350ms.

1. Go to Options→Session Profile→Connection Type, select Serial/Modem and the set these options.

2. Close the session profile window using 'Save'
3. Press Shift+Ctrl+F10
4. The status output of that connection type will show the current device-parameter [1]COM3:57600-8N1|9|350 which you can use as a parameter to the ZocSetDeviceOpts command.

Example:

```
/* Set serial options to:
   COM3, 57600-8N1, RTS/CTS, Valid-CD, 350ms */
CALL ZocSetDeviceOpts "[1]COM3:57600-8N1|9|350"
```

Example:

```
/* Select telnet and set options for
   "Start session with local echo" */
CALL ZocSetDevice "TELNET"
CALL ZocSetDeviceOpts "[3]12"
```

5.57 ZocSetEmulation <emulationname> [, <emuparm-string>]

The ZocSetEmulation command allows a script to activate a different terminal emulation. The parameter can be one of the names shown in the Emulation section of the session profile dialog, e.g. `VT220`, `TN3270`, etc.
The optional emuparm parameter contains optional settings that configure emulation dependent options (otherwise the settings from the current session profile are used).

Example:

```
CALL ZocSetEmulation "Xterm"
```

5.58 ZocSetHostEntry "name", "<key>=<value>"

Sets a value for a host directory entry from a key-value pair. To find actual names for these key-value pairs, please check the file HostDirectory.zocini (stored in the ZOC data folder) using an editor. The file contains all the key-value pairs that make up your host directory.

If instead of a key-value pair, you pass the string ##NEW##, a new host directory entry with that name will be created (if it does not yet exist). You can then configure it with subsequent calls that provide key-value pairs.

See the ZocSetSessionOption command for more background about key-value pair handling in general.

Example:

```
CALL ZocSetHostEntry "MyBBS", "emulation=1"

pair= ZocGetHostEntry("ZOC Support BBS", "calls")
PARSE VALUE pair WITH key"="value
value= value+1
CALL ZocSetHostEntry "ZOC Support BBS", "calls="||value
```

Example:

```
name= "My Router"
Call ZocSetHostEntry name, "##NEW##"
Call ZocSetHostEntry name, 'connectto="192.168.1.1"'
Call ZocSetHostEntry name, 'username="root"'
Call ZocSetHostEntry name, "deviceid=9"
Call ZocSetHostEntry name, "emulationid=3"
```

See also: ZocSetSessionOption, ZocGetHostEntry

5.59 ZocSetLogfileName <filename>

Set new name for logging.

Example:

```
CALL ZocSetLogfileName "Today.LOG"
```

5.60 ZocSetLogging 0|1 [,1]

Suspend/resume logging. If a second parameter with value `1` is given, ZocSetLogging will suppress the notification window.

Example:

```
CALL ZocSetLogging 1
```

5.61 ZocSetMode <key>, <value>

This commands allows you to set operation modes for some of the script commands. The following modes are supported:

SAY	A value of *`RAW`* sets the SAY command to *not* convert control codes like `^M` into their respective character values. A value of *`COOKED`* switches back to standard behavior.

	Example: `CALL ZocSetMode "SAY", "RAW"`
RESPOND	A value of *`RAW`* sets the respond command to *not* convert control codes like `^M` into their respective values for both sides of the command. **Example:** `CALL ZocSetMode "RESPOND", "RAW"`

5.62 ZocSetProgramOption "<key>=<value>"

This command modifies a ZOC option from the Options→Program Settings dialog. (similarly ZocSetSessionOption modifies the Options→Session Profile). The function basically works the same as ZocSetSessionOption but the underlying file which contains the key-value pairs is Standard.zfg You can load the file into an editor and you will see that the key names are mostly self explanatory. If you have problems finding a specific key or value, you can start the rexx script recorder, change the options, stop the recording and look at the resulting script.

Example:

```
CALL ZocSetProgramOption "SafAskClrCapt=yes"
CALL ZocSetProgramOption 'ScriptPath="ZocREXX"'  /* mind the
quotes */
CALL ZocSetProgramOption 'ScriptPath="'||pathvar||'"'  /*
mind the quotes */
```

See also: ZocCommand("SAVEPROGRAMSETTINGS"), ZocGetSessionOption, ZocSetSessionOption, ZocGetProgramOption

5.63 ZocSetSessionOption "<key>=<value>"

Sets a ZOC option from the Options→Session Profiles window, based on a key-value pair. To find out more about the key-value pairs, please have a look at the contents of the Standard.zoc file (or any other *.zoc session profile). The file contains all the key-value pairs, which make up a session profile.

If you are not sure what the value of a certain key means, just set the option you want to change in the options window, then click Save and check the options file for the new key-value pair. Or you can also start the REXX script recorder, then make the changes to the session profile and then stop recording and look at the resulting script.

Example:

```
CALL ZocSetSessionOption "JumpScroll=3"
CALL ZocSetSessionOption "ShowChat=no"
CALL ZocSetSessionOption 'MdmIni="ATZ^M"'  /* mind the quotes
*/
CALL ZocSetSessionOption 'TransAutoRemove="'||valvar||'"'  /*
mind the quotes */
```

Note: ZocSetSessionOption/ZocGetSessionOption will only work for options from the Options→Session Profile dialog. To change Options→Program Settings, use ZocSetProgramOption.

See also: ZocCommand("SAVESESSIONPROFILE"), ZocSaveSessionProfile, ZocGetSessionOption, ZocGetProgramOption, ZocSetProgramOption

5.64 ZocSetTimer <hh:mm:ss>

Set connection timer to given time. If called with an empty parameter, the function will return the current duration of the session timer in seconds. Calling the function with a parameter string `"STOP"` will hold the timer and `"RESUME"` will continue with a held timer.

Example:

```
CALL ZocSetTimer "00:00:20"
```

5.65 ZocSetUnattended 0|1

Enable or disable ZOC's unattended mode (same as command line parameter /U).

Example:

```
CALL ZocSetUnattended 1
```

5.66 ZocShell <command>, [<viewmode>]

Execute a command or program in a shell window via cmd.exe /c <command> (Windows) or /bin/bash -c "<command>" (Mac OS X). This is similar to using to REXX's native `ADDRESS CMD "<command>"` and essentially allows the execution of anything that you can execute in the shell/terminal window of the operating system.

Windows only: The optional viewmode parameter controls how the black shell window is shown: 0= normal, 1= hidden, 2= minimized, 3= maximized.

In many cases ZocShell is identical to using the native REXX shell interface via `ADDRESS CMD`

"<command>"

Example:

```
CALL ZocShell "DEL FILE.TMP"
CALL ZocShell "touch /tmp/file.lck", 1
```

See also: ZocShellExec, ZocShellOpen, ZocFileDelete, ZocFileRename

5.67 ZocShellExec <command>[, <viewmode>]

Execute a program directly (i.e. without passing it through the shell's command interpreter, thus avoiding the overhead of running the shell). Under Windows this only works for .com and .exe files (i.e. it will not work for .CMD scripts and internal shell commands like DEL, REN etc.).

The optional viewmode parameter controls how the application window is shown:
0= normal, 1= hidden, 2= minimized, 3= maximized.

ZocShellExec also returns the exit code of the application to the REXX script.

Example:

```
CALL ZocShellExec 'notepad.exe "somefile.txt"'
```

See also: ZocShell, ZocShellOpen

5.68 ZocShellOpen <filename>

This function is somewhat equivalent of a double click on a file, because it passes a filename to the operating system with a request to open it. The operating system will then start the program which is associated with the type of that file (e.g. a PDF viewer for PDF files or Notepad for TXT files).

Alternately, an URL can be passed as a parameter instead of a filename.

Example:

```
CALL ZocShellOpen 'C:.pdf'
```

Example:

```
CALL ZocShellOpen 'http://www.emtec.com/'
```

See also: ZocShell, ZocShellExec

5.69 ZocString(<subcommand>, <inputstring>, <p1> [, <p2>])

This is a function to manipulate a string and return a modified copy. The subcommand and the parameters <p1> and <p2> control the modification.

LINE	Return the <p1>th element of <inputstring> which is delimited by a line feed (hex 0A) and stripped of leading or trailing carriage return characters (hex 0D) (simply speaking, this refers to the <p1>th line). This is useful to parse multiline results from a ZocReceiveBuf call, e.g. `name= ZocString("LINE", data, 4)` will return the 4th line from the data variable.
LINECOUNT	Returns the number of elements in <inputstring> which are accessible as LINE.
MIME-ENCODE	Converts <inputstring> to base-64/MIME.
MIME-DECODE	Converts <inputstring> from base-64/MIME.
PART	Return the <p1>th part of <inputstring> which is delimited by <p2>, e.g. `name= ZocString("PART", "Anne|Charly|Joe", 2, "|")` will return "Charly".
PARTCOUNT	Return the number of <p1>-delimited parts in <inputstring>, e.g. `count= ZocString("PARTCOUNT", "Anne|Charly|Joe", "|")` will return 3.
REPLACE	Return a copy of <inputstring> where all occurrences of <p1> are replaced by <p2>, e.g. `betterstr= ZocString("REPLACE", str, "HyperTerminal", "ZOC")`
REMOVE	Return a copy of <inputstring> where all occurrences of <p1> are removed, e.g. `betterstr= ZocString("REMOVE", str, "HyperTerminal")`
REMOVECHARS	Return a copy of <inputstring> from which all characters of <p1> were removed, e.g. `str= ZocString("REMOVECHARS", str, "0123456789"||X2C(09))` removes all digits and tab characters from STR.
TAB	Return the <p1>th element of <inputstring> which is delimited by a tab-character, e.g. `name= ZocString("TAB", tabbed_data, 2))`
TABCOUNT	Return the number of elements of <inputstring> accessible as TAB.
WORD	Return the <p1>th element of <inputstring> which is delimited by a space, e.g. `name= ZocString("WORD", "The quick brown fox", 3)` will return "brown". Same as REXX's native WORD() function.
WORDCOUNT	Return the number of elements of <inputstring> accessible as WORD.

Example:

```
CALL ZocReceiveBuf 1024
CALL ZocSend "ps^M"
CALL ZocWait "$"
data= ZocReceiveBuf(0)

/* display result list line by
   line but ignoring the first*/
howmany= ZocString("LINECOUNT", data)
DO i=2 TO howmany
    SAY ZocString("LINE", data, i)
END
```

See also: ZocCtrlString

5.70 ZocSuppressOutput 0|1

Enables or disables suppressing of screen output. This command allows you to send/receive characters without any screen activity. Logging to the capture buffer and to file is also suppressed.

Output suppression is automatically reset to normal when the script ends or upon disconnecting from a host.

5.71 ZocSynctime <time>

This command lets you define the sync-time period (the default time is 250ms).

Background: Because REXX programs are running parallel to ZOC, the main window may receive and process more incoming data while REXX is performing its own work. This means that in some situations it is possible that text, which you expect to be received and which are going to wait for, has actually already scrolled by.
A typical example for this is a loop that attempts to process all incoming lines of text.

Example:

```
DO FOREVER
    timeout= ZocWaitLine()
    IF timeout\=640 THEN DO
        data= ZocLastLine()
        /* process data in some way */
    END
END
```

In this example ZOC could receive more text while the REXX program still processes the data from ZocLastLine and before it is ready to loop and issue the next ZocWaitLine.

To address this problem, ZOC's processing of incoming traffic is suspended for a short time period (the sync-time) whenever a Wait-command has been satisfied, thus giving the REXX program time to process the result and to get ready to wait for more data.

After a Wait-command (ZocWait, ZocWaitMux, etc.), ZOC will resume processing of incoming data either if the sync-time has elapsed or if the REXX program issues another Wait-command or if another command is processed, which needs to interact with the main window (ZocWrite, ZocSend, etc. but also SAY, TRACE, because those output to the main window).

Important: Instead of increasing the sync-time, in loops like the above, you should consider to merely collect and store the data for later processing (the ZocWaitLine command has an example of how to do this properly). An even better alternative is the use ZocReceiveBuf to catch all the data in one packet.

See also: ZocWait, ZocWaitIdle, ZocWaitLine, ZocWaitMux, ZocReceiveBuf, ZocLastLine

5.72 ZocTerminate [<return-code>]

Causes ZOC to end the program and close after the REXX program has ended. Usually an `EXIT` command will follow ZocTerminate.
If you supply the return-code parameter, the ZOC process will return this value to the operating system or to the calling program.

Example:

```
CALL ZocTerminate
EXIT
```

5.73 ZocTimeout <sec>

Set max. time to wait for a ZocWait/ZocWaitMux/ZocWaitLine/ZocEventSemaphore command.

Example:

```
/* Make subsequent ZocWait commands expire after 30 seconds
*/
CALL ZocTimeout 30

/* Wait until the host sends 'ready' of until the timeout
expires */
timeouttimeout= ZocWait("ready")
IF timeout=640 THEN SAY "ready-prompt not received within 30
seconds"
```

See also: ZocWait, ZocWaitIdle, ZocWaitLine, ZocWaitMux, ZocEventSemaphore, ZocSyncTime

5.74 ZocUpload <protocol>[:<options>],<path/filename>

Start to upload a file using the given file transfer protocol.

If the filename contains no path, it is taken from the standard upload directory, otherwise, if the path is relative, it is accessed relative to ZOC's program folder or relative to the upload folder.

For file transfer protocols which support multiple files (Ymodem, Zmodem), the file parameter may contain wildcards. Multiple filenames may be provided as one string in which the file names are separated by vertical bars `*.pdf|somefile.txt`

The protocol name is `ASCII`, `BINARY` or the name of the file transfer protocol from the Options→Session Profile→File Transfer dialog, e.g. `Zmodem` or `Kermit`.

Depending on the success of the transfer, ZocUpload will return `##OK##` or `##ERROR##`.

Example:

```
CALL ZocUpload "ZMODEM", "id_dsa.pub"
```

uploads id_dsa.pub from the local upload folder to the remote host using the Zmodem protocol.

Example:

```
success= ZocUpload("XMODEM", "updates.zip")
```

uploads the file updates.zip from the upload folder via Xmodem protocol and obtains a success indicator (`##OK##` or `##ERROR##`).

Example:

```
CALL ZocUpload "BINARY", "CNC-CONTROL.DAT"
```

sends the contents of the file directly without any translation or protocol.

Example:

```
CALL ZocUpload "ASCII", "commands.txt"
```

uploads commands.txt using the text sending options which are currently configured in the session profile.

Example:

```
CALL ZocUpload "ASCII:CRONLY+10", "\FAR\AWAY\LIST.TXT"
```

uploads LIST.TXT via text/ascii transfer using CR-only as the end of line marker and a character delay of 10ms.

Example:

```
CALL ZocUpload "ASCII:1+3", "HERE\SOME.DATA"
```

uploads the file SOME.DATA via ascii transfer with CR/LF translation and a character delay of 3ms.

Transfer Options
For all protocols (except IND$FILE, ASCII and BINARY, see below), you can set the optional options by copying a string from a session profile which configures that protocol. To do this, set your desired protocol options in ZOC's Options→ Session Profile→ File Transfer dialog and query the current parameter string by pressing Shift+Ctrl+F10 in ZOC's main window.

The optional options are set by using a string that configures that protocol. Valid strings can be obtained, if you set your protocol options in ZOC's Options→Session Profile→File Transfer dialog, press Save and then edit the `Options\Standard.zoc` file on your hard disk. The [OPTS_TRANSFER] section will contain an `ActiveTransfer=n` entry (e.g. n=2 for Zmodem) and a list of entries named `TransferOpts#nn` each of which corresponds to one file transfer protocol (`TransferOpts#02` being the Zmodem settings).

Example:
If you want to perform a file transfer via Xmodem using the CRC option and 1KB data blocks, you

1. Go to ZOC's Options→ Session Profile→ Transfer dialog, select Xmodem and configure these options.
2. Close the session profile window (click 'Save')
3. In ZOC's main window press Shift+Ctrl+F10
4. The status output will show the current transfer options "[0]kc" which you can use as a parameter to the ZocUpload command.
5. The REXX command will be `CALL ZocUpload "XMODEM:[0]kc", "datafile.zip"` (please note that the options are case sensitive).

Transfer Options ASCII
For the file transfer in ASCII mode (equivalent of Transfer, Send Text File), the options parameter has the format "mode+chardelay+linedelay", where mode specifies the end of line translation as a number or text (ASIS= 0, CRLF= 1, CRONLY= 2, LFONLY= 3) and delays are the per character and per line send delays, e.g. valid parameters would be `2+5+100` or `CRONLY+5+100` (which means CR only with 5 milliseconds per character and 100 ms extra delay at the end of each line).

Transfer Options BINARY
For a file transfer in BINARY-mode (equivalent of Transfer, Send Text File), the options parameter can be a number. This number sets the character delay in milliseconds (otherwise the text sending delay from the session profile will be used).

Transfer Options IND$FILE
The IND$FILE file transfer type is an exception to the above. The options part for IND$FILE actually contains the corresponding host command to perform the transfer (you can look up this command at the bottom of the dialog which is shown when performing a manual file transfer via IND$FILE), e.g. `CALL ZocUpload "IND$FILE:TSO IND$FILE PUT 'userid.projects.asm(report)' ASCII CRLF", "report.asm"`

5.75 ZocWait(<text>)

Waits until the given text is received. If ZocWait times out (see ZocTimeout), it returns a value of

640.

Example:

```
CALL ZocTimeout 20
timeout= ZocWait("Password")
IF timeout=640 THEN SAY "Password prompt not received within
20 seconds"
ELSE CALL ZocSend "alohomora^m"
```

Example:

```
CALL ZocTimeout 10

timeout= ZocWait("enter command>")
IF timeout=640 THEN SIGNAL the_end
CALL ZocSend "ENABLE FIREWALL^M"

timeout= ZocWait("enter command>")
IF timeout=640 THEN SIGNAL the_end
CALL ZocSend "ENABLE IPFILTER^M"

timeout= ZocWait("enter command>")
IF timeout=640 THEN SIGNAL the_end

SAY "Firewall and Ip-Filter activated!"

the_end:
Call ZocDisconnct
```

Note: ZOC automatically filters ANSI/VTxxx/etc. control sequences from the data stream to avoid interference with the ZocWait command (see ZocWaitForSeq).

Note: If you are using ZocWait from a DDE controller, the command must be sent as via sent as via DDE-Request rather than via DDE-Execute.

Note: With the TN3270 and TN5250 emulations, the command can be used to wait for `^Z`, which will be used as a signal that indicates that the emulation is ready to accept input again. Also, because these emulations are transmitting whole screens at a time, you should only issue one ZocWait per screen. In other words, if a ZocWait("LOGON:") returns, you can assume that the whole logon-screen has arrived and is ready for input.

See also: ZocWaitIdle, ZocWaitLine, ZocWaitMux, ZocTimeout, ZocReceiveBuf, ZocLastLine, ZocSyncTime, ZocWaitForSeq

5.76 ZocWaitForSeq 1|0|"on"|"off"

Normally Wait-commands ignore emulation control sequences in the data stream. If you need to wait for an emulation control, you can use this command to enable their visibility to the Wait commands.

This command also turns on/off filtering of emulation controls for ZocReceiveBuf.

Example:

```
esc= ZocCtrlString("^[")

/* wait for VT220 color reset */
CALL ZocWaitForSeq "On"
Call ZocWait esc||"[0m"
```

See also: ZocWait, ZocWaiMux, ZocReceiveBuf

5.77 ZocWaitIdle(<time>)

Wait until there was no data received from the remote host for the given amount of time (in seconds).

If the host keeps sending data, the command will time out after the time set by ZocTimeout (a value of 640 will be returned to indicate the timeout).

Example:

```
CALL ZocTimeout 60
timeout= ZocWaitIdle(2.5)
IF timeout=640 THEN SAY "Host kept sending a steady stream of
data for 60 seconds"
ELSE SAY "OK, finally the host did stop the chatter (2.5
seconds of silence detected)"
```

See also: ZocWait, ZocWaitLine, ZocWaitMux, ZocTimeout, ZocSyncTime, ZocReceiveBuf, ZocLastLine

5.78 ZocWaitLine()

Wait for the next non empty line of text from the remote host (if you want to wait for the next line, no matter if empty or not, use EXMLPL(ZocWait "^M")).

The received text will then be available using the ZocLastLine function or it can be accessed with some extra coding via ZocReceiveBuf.

If ZocWaitLine times out, it returns a value of 640.

Note: Since REXX is running in its own thread, it is possible that ZocWaitLine will miss lines if new text is received by ZOC while the the REXX program is still processing the previously received data. Thus, especially when using ZocWaitLine in a loop, you should just just collect the data until it is complete and then process it in a 2nd pass (see the sample below and the description of the ZocSyncTime command).

Example:

```
rc= ZocWaitLine()
IF rc\=640 THEN DO
    reply= ZocLastLine()
    IF reply=="CONNECT" THEN ...
```

Example:

```
/* issue a command that outputs a bunch of lines of data */
CALL ZocSend 'dig emtec.com && echo "<<END>>"^M'

/* first collect all the data and store it in an array */
n= 0
DO FOREVER
    timeout= ZocWaitLine()

    /* exit loop if no more data */
    IF timeout=640 THEN LEAVE

    line= ZocLastLine()

    /* exit loop on a line meeting some condition */
    IF line=="<<END>>" THEN LEAVE

    /* store line in array */
    n= n+1
    data.n= line
    data.0= n
END

/* when done, process each line of collected data in a 2nd
pass */
DO i= 1 TO data.0
    line= data.i

    /* line can now leisurly be processed
       without fear of missing data */
END
```

See also: ZocWait, ZocWaitIdle, ZocWaitMux, ZocTimeout, ZocSyncTime

5.79 ZocWaitMux(<text0> [, <text1> ...])

Wait for one of multiple texts in the input data stream. The command will return if one of the given texts is found in the incoming data stream or after the default timeout has expired. The return code provides information about which text was found (0, 1, 2 ...) or if the command timed out (return code 640).

Note: The sum of the lengths of all texts must not exceed 4096 characters.

Example:

```
CALL ZocTimeout 45
ret= ZocWaitMux("You have mail", "Main Menu")
SELECT
    WHEN ret=0 THEN CALL handle_maildownload
    WHEN ret=1 THEN LEAVE
    WHEN ret=640 THEN SIGNAL handle_error
END
```

See also: ZocWait, ZocWaitIdle, ZocWaitLine, ZocTimeout, ZocSyncTime

5.80 ZocWindowState(MINIMIZE|MAXIMIZE|RESTORE|ACTIVATE|MOVE:x,y|QUERY)

Set the State of ZOC's main window to the state given or moves the window to the given pixel coordinate (e.g. `MOVE:20,100`).

In used in function call syntax, the command will return new state of the window.

A parameter string of QUERY will just return the current window state as MINIMIZED, MAXIMIZED or RESTORED (please note the extra letter D at the end of those words).

Example:

```
now= ZocWindowState("QUERY")
IF now\="MINIMIZED" THEN DO
    CALL ZocWindowState "MINIMIZE"
END
```

5.81 ZocWrite <text>

Write text to screen. This command is similar to REXX's native SAY command, but it does not place the cursor in the next line after printing the text and it understands control codes like `^M` (Enter) or `^[` (ESC).

Example:

```
/* use a VT220 escape sequence to highlight a word */
CALL ZocWrite "Hello ^[[1m World^[[0m"
```

5.82 ZocWriteln <text>

Write text to screen and skip to the next line. This command is similar to the REXX SAY command,

but it resolves control codes (e.g. ^[for ESC).

Example:

```
CALL ZocWriteln "Hello ^M^J World"
SAY "Hello"||X2C(0D)||X2C(0A)||"World"
```

6 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.

6.1 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.

6.2 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.

6.3 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 afterward.

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 8th and 10th characters, respectively.

6.4 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.

6.5 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.

6.6 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`.

6.7 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.

6.8 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.

6.9 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.

6.10 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.

6.11 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 10th character before the 100th 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. Afterward, the sixth line is the remaining part of the old seventh line, and the new seventh line is the old eighth line, etc.

6.12 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.

6.13 Common Differences and Problems with Stream I/O

This section describes some of the common traps and pitfalls of REXX I/O.

6.13.1 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 kludgey 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.

6.13.2 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.

6.13.3 `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.

6.13.4 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.

6.13.5 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.

6.13.6 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.

6.13.7 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.

6.13.8 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`.

6.13.9 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!')	14	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!')	13	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.

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

6.13.10 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.

6.13.11 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')	-1	if non-existent
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.

6.13.12 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.

7 Extensions

This chapter describes how extensions to Regina are implemented. The whole contents of this chapter is specific for Regina.

7.1 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.
- Backwards compatibility.

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.

7.2 ZOC REXX Extensions

In addition to the original Regina REXX language elements, ZOC extends the language with commands to perform tasks related to terminal emulation (making connections, sending text, receiving files, etc.).

From the perspective or REXX those are a 3rd party library and they are not covered here . (The documentation for the ZOC specific commands can be found in the help menu of the ZOC program under *ZOC Rexx commands*).

7.3 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].

7.4 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 belong 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.

7.5 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 parsing the program 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 mode, 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 before the REXX program is parsed:

- It can be invoked by using a command line option to the interpreter; say -e. The word following the option is the extension or standard to invoke. Multiple `-e` options can be specified. This

method is not supported in Regina.

- It can be invoked by setting an environment variable, which must be a string of the same format as the parameters to the OPTIONS clause. Regina supports this mechanism by the use of the REGINA_OPTIONS environment variable.

7.6 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 this is the LINES BIF. In the following piece of code the same BIF returns different data:

```
/* file 'aa' contains 5 lines */
options FAST_LINES_BIF_DEFAULT
do i=1 to 2
    if i=2 then OPTIONS NOFAST_LINES_BIF_DEFAULT
    say lines('aa')
end
```

In the first iteration of the loop, LINES('aa') returns 1, indicating that there is at least 1 line remaining the the stream 'aa'. However, in the second iteration of the loop, LINES('aa') will return 5, indicating that there are 5 lines remaining in the stream.

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.

7.7 The Fundamental Extensions

Here is a description of all "atomic" extensions in Regina:

[AREXX_BIFS]

This option allows the user to enable or disable the AREXX BIFs introduced into Regina 3.1. The default is AREXX_BIFS on Amiga and AROS, but NOAREXX_BIFS on all other platforms.

[AREXX_SEMANTICS]

With the introduction of AREXX BIFs into Regina 3.1, differences in the semantics of a number of BIFs resulted. These BIFs that differ between *Standard Regina* and *AREXX* are OPEN(), CLOSE() and EOF(). This OPTION specifies that the AREXX semantics be used for these BIFs. The default is to use Regina semantics for these BIFs.

[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.

[CALLS_AS_FUNCS]

Allows the old broken syntax of :

```
call myfunc(arg1,arg2)
```

New programs should use the standard syntax for the CALL instruction. As the determination of invalid syntax is done before the code is executed, then this OPTION can only be specified using the REGINA_OPTIONS environment variable. NOCALLS_AS_FUNCS is the default.

[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 does 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.

[DESBUF_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.

[DROPBUF_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.

[EXT_COMMANDS_AS_FUNCS]

When Regina resolves an expression to a function, and that function is not a built-in or a registered external function, Regina attempts to execute the function as an operating system command. With NOEXT_COMMANDS_AS_FUNCS set, Regina will return error 43; "Routine not found". EXT_COMMANDS_AS_FUNCS is the default.

[FAST_LINES_BIF_DEFAULT]

The LINES() BIF in versions of Regina prior to 0.08g returned the actual number of lines available in a stream. Since then, the LINES() BIF has been changed to only return 0 or 1. This was done for two reasons. First, it is faster, and secondly. the ANSI standard allows for an option to return the actual number of lines. This OPTION is for backwards compatibility with programs written assuming the prior behavior of the LINES() BIF.

FAST_LINES_BIF_DEFAULT is the default.

[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.

[HALT_ON_EXT_CALL_FAIL]

This options tells the interpreter that when a called external routine fails the caller halts with a syntax error 40.1. This behaviour also occurs with the STRICT_ANSI option. NOHALT_ON_EXT_CALL_FAIL is the default.

[INTERNAL_QUEUES]

Regina implements multiple named queues both as part of the interpreter, and as an external resource. If a queue name has the character '@' embedded, Regina will assume this to be an external queue name. This OPTION allows the exclusive use of Regina's internal queuing mechanism regardless of the queue name. NOINTERNAL_QUEUES is the default.

[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.

[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.

[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. PRUNE_TRACE is the default.

[QUEUES_301]

This OPTION changes the behaviour of external queue names. In Regina 3.1 meaning was given to queue names. If a queue name had '@' in its name, it was identified as an external queue (requiring rxstack to be running). Before 3.1, any time RXQUEUE BIF was used, it always referenced an external queue. New programs should use the naming convention to identify external queues, because you will be able to use internal of external queues in other instructions like ADDRESS.WITH. The default is NOQUEUES_301.

[REGINA_BIFS]

This OPTION allows the user to turn on all non-ANSI extension BIFs. The default is REGINA_BIFS.

[STDOUT_FOR_STDERR]

All output that Regina would normally write to stderr, such as TRACE output and errors, are written to stdout instead. This is useful if you need to capture TRACE output and normal output from SAY to a file in the order in which the lines were generated. The default is NOSTDOUT_FOR_STDERR.

[STRICT_ANSI]

This OPTION results in interpretation of a program to strict ANSI standards, and will reject any Regina extensions. NOSTRICT_ANSI is the default.

[STRICT_WHITE_SPACE_COMPARISONS]

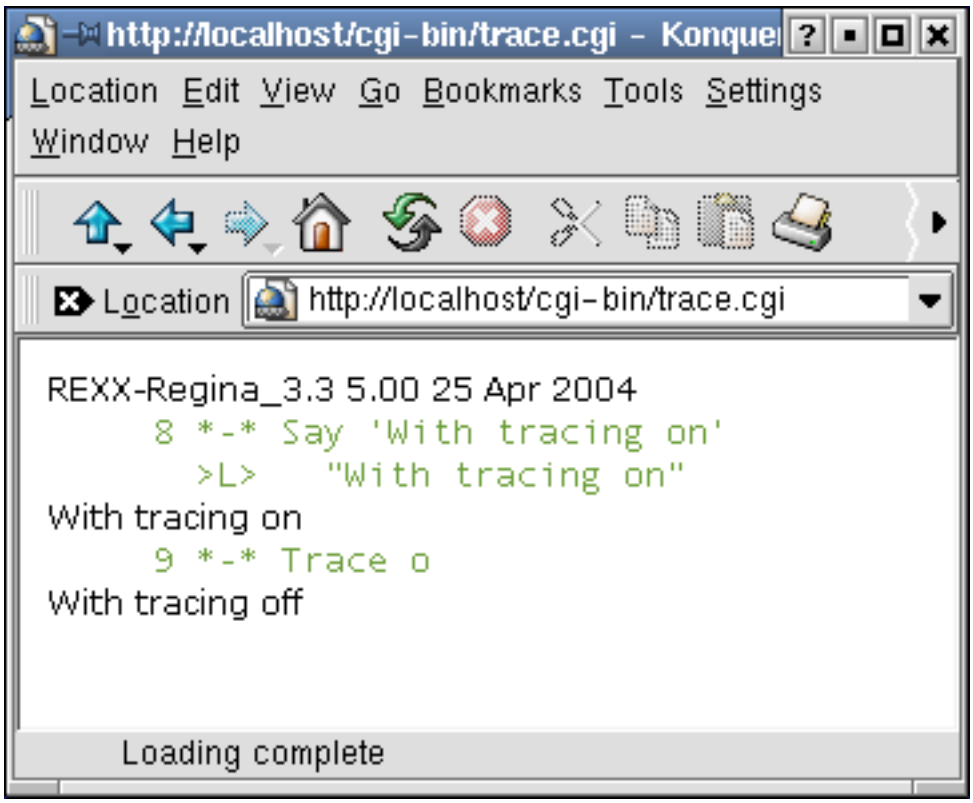
This OPTION specifies if ANSI rules for non-strict comparisons are applied. Under ANSI, when doing non-strict comparisons, only the space character is stripped from the two comparators. Under Regina's default behavior, all whitespace characters are stripped. NOSTRICT_WHITE_SPACE_COMPARISONS is the default.

[TRACE_HTML]

This OPTION generates HTML <PRE> and </PRE> tags around TRACE output, to enable tracing from within CGI scripts. The default is NOTRACE_HTML. The following code shows the necessary header information to enable this feature:

```
#!/usr/bin/rexx
OPTIONS STDOUT_FOR_STDERR TRACE_HTML
Parse Version ver
/* following 2 lines MUST be 'sayed' before TRACE turned on
*/
Say 'Content-type: text/html'
Say
Say ver
Trace i
Say 'With tracing on'
Trace o
Say 'With tracing off'
Return 0
```

The output from this would look like:



Note: OPEN_BIF, FIND_BIF, CLOSE_BIF and FILEIO OPTIONs have been removed in Regina 3.1

7.8 Meta-extensions

[ANSI]
: Combination of `STRICT_ANSI` and `STRICT_WHITE_SPACE_COMPARISONS`.

[BUFFERS]
: Combination of `BUFTYPE_BIF`, `DESBUF_BIF`, `DROPBUF_BIF` and `MAKEBUF_BIF`.

7.9 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 tasks 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.

7.10 Standards

The following table shows which options are available in different REXX Language Levels, and the default settings applicable for Regina.

[ANSI]
: REXX Language level 5.0, as described in [ANSI].

[REGINA]
: REXX Language level 5.0, plus extensions, as implemented by Regina 3.1 and above.

[SAA]
: REXX Language level ??, as defined by IBM's System Application Architecture [SAA].

[TRL1]
: REXX Language level 3.50, as described in [TRL1].

[TRL2]
: REXX Language level 4.00, as described in [TRL2].

Option	ANSI	REGINA	SAA	TRL1	TRL2
AREXX_BIFS	no	yes	no	no	no
AREXX_SEMANTICS	no	no	no	no	no
BUFTYPE_BIF	no	yes	no	no	no
CACHEEXT	no	no	no	no	no
CALLS_AS_FUNCS	no	yes	no	no	no
DESBUF_BIF	no	yes	no	no	no
DROPBUF_BIF	no	yes	no	no	no
EXT_COMMANDS_AS_FUNCS	no	yes	no	no	no
FAST_LINES_BIF_DEFAULT	yes	yes	no	no	no
FLUSHSTACK	no	no	no	no	no
HALT_ON_EXT_CALL_FAIL	no	yes	no	no	no
INTERNAL_QUEUES	no	no	no	no	no
LINEOUTTRUNC	no	yes	no	no	no
MAKEBUF_BIF	no	yes	no	no	no
PRUNE_TRACE	no	yes	no	no	no
QUEUES_301	no	yes	no	no	no
REGINA_BIFS	no	yes	no	no	no
STDOUT_FOR_STDERR	no	no	no	no	no
STRICT_ANSI	yes	no	no	no	no
STRICT_WHITE_SPACE_COMPARISONS	yes	no	no	no	no
TRACE_HTML	no	no	no	no	no

8 Implementation Limits

This chapter lists the implementation limits required by the REXX standard. All implementations are supposed to support at least these limits.

8.1 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.

8.2 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.

8.3 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.

8.4 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.

8.5 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.

8.6 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.

8.7 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)
Named Queues	100
Hexadecimal strings	source line size
Interpretable string	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.

9 Appendixes

9.1 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.

9.2 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

9.3 GNU Free Documentation License

GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition.

Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones

listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent

copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice.

These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate.

Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4.

Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so

long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.