Regina documentation

COLLABORATORS			
	TITLE : Regina documentation		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		October 9, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Regina documentation

1.1 Regina documentation

1.2 Table Of Contents

1.3 Chapter 1

REXX SYNTAX

In this chapter, the syntax of Rexx keywords are explained. At the end of the

chapter is a section describing how Regina may differ from standard Rexx as it is described in the first part of this chapter.

1.4 Definitions

variablelist is a space list of one or more variables, which may be simple symbols, stem symbols or compound symbols.

1.5 Clauses

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

The ADDRESS keyword controls where external environment commands 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, except that the environment in which it is to be executed can be explicitly specified in the ADDRESS clause. The environment term must be a symbol or a literal string. Even if it is a symbol, it will not be expanded. command can be any Rexx expression.

Rexx maintains a list of environments, of size two. If you select a new environment, it will be put in the front of this list, possible squeezing the backend environment out of the list. Note that if command is specified, the contents of the environment stack will never be changed. If you omit command, environment will always be put at the top of the stack.

What happends if you specify an environment that is already in the list seems to be implementation dependent. 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 envionment in the front. In these implementations.

If you don’t specify any subkeywords or parameters to ADDRESS, the effect is to swap the two first entries in the list. Consequently, executing ADDRESS can be used to toggle between to different environments.

The second form of ADDRESS is just a special case of the first form when command is omitted. If the first word after ADDRESS is VALUE, then the rest of the clause is interpreted as being an expression, which result names the environment which is to be made the current default environment. Using VALUE makes it possible to circumvent the restriction that the name of the new environment must be a symbol or a literal string. It is, however, not possible to combine both VALUE and command in a single clause.

To further complicate things, the VALUE keyword may be omitted if the expression (supposing there is one) following ADDRESS starts with a special character which isn't allowed in environment names. Confused? Let's look at some examples:

```
ADDRESS COMMAND ADDRESS SYSTEM 'copy' fromfile tofile
ADDRESS SYSTEM
ADDRESS VALUE newenv
ADDRESS
ADDRESS (oldenv)
```

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

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

```
ADDRESS OLDENV
ADDRESS (OLDENV)
```

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

If you are still confused, don't panic; the syntax of ADDRESS is somewhat bizarre, and you should not put too much effort into learning all aspects of it. Just make sure that you understand how to use it in simple situations. Changes 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 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 inplementation specific documentation for more information about this. Actually, there is not really any restriction on what constitues a legal environment name (also the nullstring is legal).

Nor does the definition of Rexx say anything about which environment is preselected when you invoke the interpreter. In extreme cases you might not even have an environment available before you have executed the first ADDRESS clause.

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

```
ARG [ template ]
```

The ARG clause will parse the argumentstrings at the current procedureal level into the template. Parsing will be performed in upper case mode. This clause is identical to:

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

For more information, see the PARSE clause.

```
CALL symbol [ parameter ] [, [ parameter ] ... ]
```

Transfer control to the statement suceeding the label symbol. Before control is transferred, the special variable SIGL is set to the linenumber of the CALL-statement. Control is later returned when a RETURN-statement is executed.

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

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

    conditional = WHILE exprw
                  UNTIL expru
```

The DO statement is the statement used for looping and grouping several statements into one block. The most simple case is when there is no repetitor or conditional, in which case it works like BEGIN/END in Pascal or f...g in C.

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

If you specify the controlvariable symbol, it will get the initial value expri at the start of the loop. At the start of each new iteration it will be checked if it has reached the value exprt. At the end of each iteration the value exprb is added to the controlvariable. The loop will terminate after exprf iterations.

You may also specify UNTIL or WHILE, which take a boolean expression. WHILE is checked before each iteration, UNTIL is checked after each iteration.

The FOREVER keyword is only needed when there is no conditional, and the repetitor would also be empty if FOREVER was not specified. The TO, BY and FOR may come in any order.

```
DROP variable [ variable ... ]
```

Makes the named variables undefined.

```
EXIT [ expr ]
```

Terminates the program, and returns expr to the caller.

The Standard says that this can be any string, and that if no expr is specified, nothing is returned to the caller. In UNIX the expr must be an integer, and zero is assumed in the absence of an expr.

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

A perfectly normal IF statement.

```
INTERPRET [ expr ]
```

```
ITERATE [ symbol ]
```

This statement will iterate the innermost loop, or the loop having symbol as control variable. The simple DO/END statements without a repetitor and/or conditional are not effected by ITERATE. The effect of an ITERATE is to imediately transfer control to the END statement of that loop, so that the next iteration (if any) of the loop can be started.

Two types of errors can occur. Either symbol does not refer to any loop active in the current scope, or then (if symbol is not specified) there does not exist any loops in the current scope. Both errors are reported with errorcode 28 ("Invalid LEAVE or ITERATE").

```
LEAVE [ symbol ]
```

This statement will leave the innermost loop, or the loop having symbol as control variable. As for scope, errors and functionality, it is identical to ITERATE, except that LEAVE terminates the loop, while ITERATE lets the loop start on the next iteration.

```
NOP
```

The "No OPeration" statement, it does not do anything.

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

```
OPTIONS expr
```

```
PARSE [ UPPER ] type [ template ]

    type = [ ARG | LINEIN | PULL | SOURCE | VERSION ]
           VALUE [ expr ] WITH
           VAR symbol
```

```
PROCEDURE [ EXPOSE variablelist ]
```

```
PULL [ template ]
```

This statement will pull 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]".

```
PUSH [ expr ]
```

This statement will push a new line onto the top of the stack. The contents of the line will be determined by the optional expression of the statement.

```
QUEUE [ expr ]
```

```
RETURN [ expr ]
```

```
SAY [ expr ]
```

Writes out the value of expr, terminated by a CRLF. If expr is not specified, it will write out an empty line.

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

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

```
SIGNAL label
       [ VALUE ] expr
       { ON | OFF } condition [ NAME handler ]
```

```
TRACE [ setting | [ VALUE ] expr ]
```

1.6 Chapter 2

REXX BUILTIN FUNCTIONS

This chapter describes the Rexx library of builtin functions. It is divided into three parts:

1. First a general introduction to builtin functions, pointing out concepts, fitfalls, parameter conventions, pecularities, and possible system dependencies.

2. Then there is the reference section, which describes in detail each function in the builtin library.

3. 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 Reginas extensions to the builtin library.

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

1.7 General Information

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

1.8 The Syntax Format

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

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

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

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

1.9 Precision and Normalization

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

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

1.10 Standard Parameter Names

In the descriptions of the builtin 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 methology used when naming parameters.

* Length is a non-negative whole number within the internal precision of the builtin 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 concequently 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 dependant.

* 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 builtin functions that do not easily fall into the categories above. These are given other names, and their type and functionality will be described together with the functions in which they occur.

1.11 Error Messages

There are several errors that might occur in the builtin functions. Just one error message is only relevant for all the builtin 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 builtin functions.

Depending on the implementation, other error messages might be used as well. Error message number 26 (Invalid whole number) might be used for any case where a parameter should have been a whole number, or where a whole number is out of range. It is implided 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 builtin 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 converions 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 relavent builtin function, and this will not be restated for each function. When other error messages than these are relevant, it will be mentioned in the text.

In Rexx, it is in general not an error to specify a start position that is larger than the length of the string, or a length that refers to parts of a

string that is beyond the end of that string. The meaning of such instances will depend on the context, and are described for each function.

1.12 Possible System Dependencies

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

* ADDRESS() is dependant on your operating system and the implementation of Rexx, since there is not standard for naming environments.

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

* BITAND(), BITOR() and BITXOR() are dependant 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 uneffected 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 filesystem. They might have system dependant pecularities in several ways. Firstly, the naming of streams is very dependant 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 dependant on the condition system, which in turn depends on such implementation dependant 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 choosen.

* 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 choosen language.

* DELWORD(), SUBWORD(), WORD(), WORDINDEX(), WORDLENGTH(), WORDPOS() and

WORDS() requires the concept of a "word", which is defined as a non-blank characters separated by blanks. However, the interpretation of what is a blank character depends upon the implementation.

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

* QUEUED() refers to the system specific concept of a "stack", which is external to Rexx. The result of this function may therefore be dependant on how the stack is implemented on your system.

* RANDOM() will differ from machine to machine, since the algorithm is implementation dependant. 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 reproduceable 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 underlaying operating system to produce the timing information. In particular, the granularity and accuracy of this information may vary.

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

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

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

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

1.13 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 binded to the spacebar on a normal computer keyboard.

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

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

1.14 Rexx Standard Builtin Functions

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

ABBREV(long,short[,length])

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

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

```
ABBREV( 'Foobar', 'Foo' )     ->  1
ABBREV( 'Foobar', 'Foo', 4)   ->  0  /* Too short */
ABBREV( 'Foobar', 'foo' )     ->  0  /* Different case */
```

ABS(number)

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

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

ADDRESS()

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

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

ARG([argno[,option]])

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

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

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

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

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

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

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

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

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

```
If called as: CALL FUNCTION 'This' 'is', 'a',, 'test',,

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

B2X(binstring)

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

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

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

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

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

The behavior of this function when the two strings do not have equal length is defined by the padchar character. If it is udefined, the remaining part of the longer string is appended to the result after all characters in the shorter srting 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 dependant on the character set used. To lowercase a string in EBCDIC, use BITAND() with a padchar value of 'bf'x. To do the same in ASCII, use BITOR() with a padchar value of '20'x.

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

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

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

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

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

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

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

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

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

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

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

Note that this function is very dependant 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 builtin library, so size restrictions on whole numbers that generally applies for builtin functions, do not apply in this case.

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

```
C2D('ffff'x,3)   ->  '65535'
C2D('fff9'x,2)   ->  '-6'
C2D('ff80'x,2)   ->  '-128'
```

C2X(string)

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

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

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

This function has two names, to support both American and British spelling. It will center string in a string total of length length characters. If length (which must be a non-negative wholenumber) 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**'
```

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

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

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

The length parameter specifies the number of characters to read. If the

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

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

What happens when an error occurs or the End-Of-File (EOF) is seen during reading, is implementation dependant. The implementation may choose to set the NOTREADY condition (does not exist in Rexx language level 3.50). For more information, se chapter on stream I/O.

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

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

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

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

If start is specified, the current write position will be set to the startth 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. Independant 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 info.

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

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

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

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

CHARS([streamid])

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

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

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

COMPARE(string1,string2[,padchar])

This function will compare string1 to string2, and return a whole number which will be 0 if they are equal, otherwise the position of the first character at which the two strings differ is returned. The comparisation 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 comparation. If a padchar is not specified, space is used.

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

CONDITION([option])

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

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

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

- 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 3. Note that condition may in several ways be dependant on the implementation and system, so read system and implementation dependant information too.

COPIES(string,copies)

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

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

DATATYPE(string[,option])

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

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

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

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

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

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

N (Numeric) If string is a valid Rexx number, i.e. DATATYPE(string) would return NUM.

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

U (Upper) Consists of only upper case alphabetic characters.

W (Whole) If string is a valid Rexx whole number under the current setting of NUMERIC. Note that 13.0 is a whole number since the decimal part is zero, while 13E+1 is not a whole number, since it must be interpreted as

130 plus/minus 5.

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

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

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

```
DATATYPE(' - 1.35E-5 ')           ->  'NUM'
DATATYPE('1E999999999')           ->  'CHAR'
DATATYPE('1E9999999999')          ->  'CHAR'
DATATYPE('!@#&#$(&*%" Link })           ->  '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])

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

B (Base) The number of days from January 1st 0001 until yesterday inclusive, as a whole number. This function uses the Gregorian calendar extended backwards.

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

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

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

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

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

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

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

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

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

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

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

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

Assuming that today is January 6th 1992:

```
DATE('B')   ->  '727203'
DATE('C')   ->  '33609'
DATE('D')   ->  '6'
DATE('E')   ->  '06/01/92'
DATE('M')   ->  'January'
DATE('N')   ->  '6 Jan 1992'
DATE('O')   ->  '92/01/06'
DATE('S')   ->  '19920106'
DATE('U')   ->  '01/06/92'
DATE('W')   ->  'Monday'
```

DELSTR(string,start[,length])

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

```
DELSTR('Foobar', 3)      ->  'Foo'
```

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

DELWORD(string,start[,length])

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

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

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

DIGITS()

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

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

D2C(integer[,length])

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

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

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

D2X(integer[,length])

Returns a hexadecimal number that is the hexadecimal representation of integer. Integer must be a whole number under the current settings of NUMERIC, it is not effected by the precision of the builtin 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 signextended on the left side to make it the right length. If length is not large enough to hold integer, an error is reported.

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

ERRORTEXT(errno)

Returns the Rexx error message associated with error number errno. If the error message is not defined, a nullstring is returned.

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

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

FORM()

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

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

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

This function is used to control the format of numbers, and you may request the size and format in which the number is written. The parameter number is the number to be formated, 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 ommitted if the field holding the fractional part is empty. If there are less digits in the number than the size of the field, it is padded with zeros at the right. If there is more digits then it is possible to fit into the field, the number will be rounded (not truncated) to fit the field.

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

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

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

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

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

FUZZ()

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

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

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

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

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

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

LASTPOS(needle,haystack[,start])

Searches the string haystack for the string needle, and returns the position in haystack of the first character in the substring that matched needle. The search is started from the right side, so if needle occurs several times, the last occurence 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 startth character. It is not stated whether a match can partly be to the right of the start position, so some implementations may differ on that point.

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

LEFT(string,length[,padchar])

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

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

LENGTH(string)

Returns the number of characters in string.

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

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

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

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

The line parameter (which must be a positive whole number) might be specified to set the current position in the file to the beginning of line number line before the read operation starts. If line is unspecifiedified, 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 dependant. The implementation may interpret the EOF as an implicit End-Of-Line (EOL) mark is none such was explicitly present. The implementation may also choose to raise the NOTREADY condition flag (this condition is new from Rexx language level 4.00).

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

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

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

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

Returns the number of lines remaining after having positioned the stream streamid to the start of line line and written out string as a line of text. If streamid is omitted, the default output stream is used. If line

(which must be a positive whole number) is omitted, the stream will not be repositioned before the write. If string is omitted, nothing is written to the stream. If string is specified, a system-specific action is taken after it has been written to stream, to mark a new line.

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

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

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

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

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

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

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

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

LINES([streamid])

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

End-Of-File as an implicit EOL mark too, while others might not.

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

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

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

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

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

MAX(number1[,number2]...)

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

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

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

MIN(number[,number]...)

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

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

```
MAX(1, 2, 05.0E-1, 4)   ->  '0.50'
```

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

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

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

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

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

POS(needle,haystack[,start])

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

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

QUEUED()

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

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

RANDOM(max)

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

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

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

If seed is specified, you may control which numbers the pseudo-random algoritm 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 pseudorandom numbers, so the reproducability 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 favorized if the 232 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 RANDOM(10) and RANDOM(10,). The former will give a pseudo-random number in the range 0 10, while the latter will give a pseudo-random number in the range 10 999.

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

```
RANDOM()           ->  '123'  /* Between 0 and 999 */
RANDOM(10)         ->  '5'  /* Between 0 and 10 */
RANDOM(, 10)       ->  '3'  /* Between 0 and 10 */
RANDOM(20, 30)     ->  '27'  /* Between 20 and 30 */
```

```
RANDOM(,, 12345)   ->  '765'  /* Between 0 and 999, and sets seed */
```

REVERSE(string)

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

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

RIGHT(string,length[,padchar])

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

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

SIGN(number)

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

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

SOURCELINE([lineno])

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

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

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

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

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

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

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

Length must be a non-negative whole number. If specified, consequitive blanks within string is replaced by exactly length instances of space (or padchar if specified). However, padchar will only be used in the output string, in the input string, blanks will still be the "magic" characters. As a consequence, if there exist any padchars 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'
```

STEAM(streamid[,option[,command]])

This function was added to Rexx in language level 4.00. It provides a general machanism 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 dependant.

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 dependant.

- D (Description) Returns a description of the state of streamid. The return value is implementation dependant.

- S (State) 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 about stream I/O.

Note that the options Description and State really have the same function, but that State in general is implementation independant, while Description is implementation dependant.

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

Returns string after possibly stripping it of any number of leading and/or trailing characters. The default action is to strip off both leading and trailing blanks. If char (which must be a string containing exactly one character) is specified, that character will be stripped off instead of blanks. Interword blanks (or chars 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 chars if specified.

T (Trailing) Only strip off trailing blanks, or chars if specified.

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

```
STRIP(' Foo  bar ')           ->  'Foo  bar'
STRIP(' Foo  bar ', 'L')      ->  'Foo  bar '
STRIP(' Foo  bar ', 't')      ->  ' Foo  bar'
STRIP(' Foo  bar ', 'Both')   ->  'Foo  bar'
STRIP('0.1234500',, '0')      ->  '.12345'
STRIP('0.1234500 ',, '0')     ->  '.1234500'
```

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

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

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

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

SUBWORD(string,start[,length])

Returns the part of string that starts at blankdelimited 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 interword blanks will be preserved as is.

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

SYMBOL(name)

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

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

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

TIME([option])

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

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

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

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

L (Long) Returns the exact time, down to the microsecond. This is called the long format. The output might be "hh:mm:ss.mmmmmm". Be aware that most computers do not have a clock of that accuracy, so the actual granularity you can expect, will be about a few milliseconds. The hh, mm and ss parts will be identical to what is returned by the options H, M and S respectively, except that each part will have leading zeros as indicated by the format.

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

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

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

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

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

None of the formats will have leading or trailing spaces.

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

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

TRACE([setting])

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

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

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

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

Two translation tables might be specified as the strings tablein and tableout. If one or both of the tables are specified, each character in

string that exists in tablein is translated to the character in tableout that occupies the same position as the character did in tablein. The tablein defaults to the whole character set (all 256) in numeric sequence, while tableout defaults to an empty set. Characters not in tablein are left unchanged.

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

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

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

TRUNC(number[,length])

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

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

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

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

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

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

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

programs or in the operating system.

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

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

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

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

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

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

If option is specified, it can be one of:

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

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

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

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

WORD(string,wordno)

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

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

WORDINDEX(string,wordno)

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

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

WORDLENGTH(string,wordno)

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

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

WORDPOS(phrase,string[,start])

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

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

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

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

WORDS(string)

Returns the number of blankdelimited words in the string.

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

XRANGE([start][,end])

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

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

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

X2B(hexstring)

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

X2C(hexstring)

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

```
X2C(")                ->  "
X2C('466f6f 426172')   ->  'FooBar'
X2C('46 6f 6f')        ->  'Foo'
```

X2D(hexstring[,length])

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

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

```
X2D('03 24')      ->  '792'
X2D('0310')       ->  '784'
X2D('ffff')       ->  '65535'
```

```
X2D('ffff', 5)    ->  '65535'
X2D('ffff', 4)    ->  '-1'
X2D('ff80', 3)    ->  '-128'
X2D('12345', 3)   ->  '837'
```

1.15 Implementation specific documentation for Regina

Deviations from the Standard

Interpreter Internal Debugging Functions

Rexx UNIX Interface Functions

1.16 Deviations from the Standard

* For those builtin function 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 the environments available are described in chapter ??.

* parameter calling

* stream I/O

* conditions

* national character sets

* blanks

* queues

* random()

* sourceline

* time

* character sets

1.17 Interpreter Internal Debugging Functions

ALLOCATED([option])

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

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

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

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

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

S Returns a string that is nicely formatted and contains all the other three options, with labels. The format of this string is: "Memory: Allocated=XXX, Current=YYY, Leaked=ZZZ".

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

DUMPTREE()

Prints out the internal parse tree for the Rexx program currently being executed. This output is not very interesting unless you have good knowlegde 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 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.

1.18 Rexx UNIX Interface Functions

CHDIR(string)

Set string as current working directory.

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

CLOSE(streamid)

Closes the file named by string. This file must have been opened by the OPEN function call earlier.

This function is now obsolete, instead you should use:

```
STREAM( streamid, 'Command', 'CLOSE' )
```

GETENV(environmentvar)

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

This function is now obsolete, instead you should use:

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

OPEN(streamid,access)

Opens the file named 'string1' with the access 'string2'. If string2 is not specified, the access "R" will be used. string2 may contain the following characters:

- R Opened for readaccess. The file pointer will be positioned at the start of the file, and only read operations are allowed.
- W Opened for writeaccess. Like the "R", but the file is opened for write access. An error is returned if it was not possible to get appropriate access.
- A Opened for appending. Like "W", but the filepointer is automatically initiated to the end of the file.

The combination of "R" and either "W" or "A" will result in opening the

file for both reading and writing. The combination of "W" and "A" will
result in a initial filepointer set to the end of file.

The 1 is used as a sort of internal file descriptor in the Rexx interpreter.
All references to the file must have string1 as parameter to identify the
specific file (unless otherwise specified in the documentation).

Note that if you open the files "foobar" and "./foobar" they will
actually point to the same physical file. Rexx will however interpret them as
different files, and will open a file descriptor for each one.

However, if you try to open an already open file, it will have no effect.

This function is now obsolete, instead you should use:

```
STREAM( streamid, 'Command', 'OPEN' access )
```

UNIXERROR(errorno)

This function returns the string associated with the errno error number
that errorno specifies. When some UNIX interface function returns an error,
it really is a reference to an error message which can be obtained through
UNIXERROR.

This function is just an interface to the strerror() function call in UNIX,
and the actual error messages might differ with the operating system.

This function is now obsolete, instead you should use:

```
ERRORTEXT( 100 + errorno )
```

1.19 Chapter 3

CONDITIONS

In this chapter, the Rexx concept of "conditions" is described. Conditions
allow the programmer to handle abnormal control flow, and enable him to
assign special pieces of Rexx code to be executed in case of certain
incidences.

* In the first section the concept of conditions is explained.

* Then, there is a description of how a standard condition in Rexx would
  work, if it existed.

* In the third section, all the existing conditions in Rexx are presented,
  and the differences compared to the standard condition described in the
  previous section are listed.

* The fourth sections contains a collections of random notes on the
  conditions in Rexx.

* The last section describes differences, extensions and peculiarities in Regina on the of subject conditions, and the lists specific behavior.

1.20 What are Conditions

In this section, the concept of "conditions" are explained: What they are, how they work, and what they mean in programming.

1.21 What Do We Need Conditions for?

1.22 Terminology

First, let's look at the terminology used in this chapter. If you don't get a thorough understanding of these terms, you will probably not understand much of what is said in the rest of this chapter.

Incident: A situation, external or internal to the interpreter, which it is required to respond to in certain predefined manners. The interpreter recognizes incidents of several different types. The incident will often have a character of "suddenness", and will also be independent of the normal control flow.

Event: Data Structure describing one incident, used as a descriptor to the incident itself.

Condition: Names the Rexx concept that is equivalent to the incident.

Raise a Condition: The action of transforming the information about an incident into an event. This is done after the interpreter senses the condition. Also includes deciding whether to ignore or produce an event.

Handle a Condition: The act of executing some predefined actions as a response to the event generated when a condition was raised.

(Condition) Trap: Data Structure containing information about how to handle a condition.

(Trap) State: Part of the condition trap.

(Condition) Handler: Part of the condition trap, which points to a piece of Rexx code which is to be used to handle the condition.

(Trap) Method: Part of the condition trap, which defined how the condition handler is to be invoked to handle the condition.

Trigger a Trap: The action of invoking a condition handler by the method specified by the trap method, in order to handle a condition.

Trap a Condition: Short of trigger a trap for a particular condition.

Current Trapped Condition: The condition currently being handled. This is the same as the most recent trapped condition on this or higher procedure level.

(Pending) Event Queue: Data Structure storing zero or more events in a specific order. There are only one event queue. The event queue contains events of all condition types, which have been raised, but not yet handled.

Default-Action: The predefined default way of handling a condition, taken if the trap state for the condition raised is OFF.

Delay-Action: The predefined default action taken when a condition is raised, and the trap state is DELAY.

1.23 The Mythical Standard Condition

Rexx Language Level 4.00 has six different conditions. However, each of these is a special case of a mythical, non-existing, standard condition. In order to better understand the real conditions, we start by explaining how a standard condition work.

In the examples below, we will call our non-existing standard condition MYTH. Note that these examples will not be executable on any Rexx implementation.

Information Regarding Conditions (data structures)

How to Set up a Condition Trap

How to Raise a Condition

1.24 Information Regarding Conditions (data structures)

There are mainly five conceptual data structures involved in conditions.

Event queue. There is one interpreter-wide queue of pending conditions. Raising a condition is identical to adding information about the condition to this queue (FIFO). The order of the queue is the same order in which the conditions are to be handled.

Every entry in the queue of pending conditions contains some information about the event: the line number of the Rexx script when the condition was raised, a descriptive text and the condition type.

Default-Action. To each, there exists information about the default-action to take if this condition is raised but the trap is in state OFF. This is called the "default-action". The standard default-action is to ignore the condition, while some conditions may abort the execution.

Delay-Action. Each condition will also have delay-action, which tells what to do if the condition is raised when condition trap is in state DELAY. The standard delay-action is to queue the condition in the queue of pending conditions, while some conditions may ignore it.

Condition traps. For each condition there is a trap which contains three pieces of status information: the state; the handler; and the method. The state can be ON, OFF or DELAY.

The handler names the Rexx label in the start of the Rexx code to handle the event. The method can be either SIGNAL or CALL, and denotes the method in which the condition is to be handled. If the state is OFF, then neither handler nor method is defined.

Current Trapped Condition. This is the most recently handled condition, and is set whenever a trap is triggered. It contains information about method, which condition, and a context-dependent description. In fact, the information in the current trapped condition is the same information that was originally put into the pending event queue.

Note that the event queue is a data structure connected to the interpreter itself. You operate on the same event queue, independent of subroutines, even external ones. On the other hand, the condition traps and the current trapped condition are data structures connected to each single routine. When a new routine is called, it will get its own condition traps and a current trapped condition. For internal routines, the initial values will be the same values as those of the caller. For external routines, the values are the defaults.

The initial value for the event queue is to be empty. The default-action and

the delay-action are static information, and will always retain their values during execution. The initial values for the condition traps are that they are all in state OFF. The initial value for the current trapped condition is that all information is set to the nullstring to signalize that no condition is currently being trapped.

1.25 How to Set up a Condition Trap

How do you set the information in a condition trap? You do it with a SIGNAL or CALL clause, with the ON or OFF subkeyword. Remember that a condition trap contain three pieces of information? Here are the rules for how to set them:

* To set the trap method, use either SIGNAL or CALL as keyword.

* To set state to ON or OFF, use the appropriate subkeyword in the clause. Note that there is no clause or function in Rexx, capable of setting the state of a trap to DELAY.

* To set the condition handler, append the term "NAME handler" to the command. Note that this term is only legal if you are setting the state to ON; you can not specify a handler when setting the state to OFF.

The trap is said to be "enabled" when the state is either ON or DELAY, and "disabled" when the state is OFF. Note that neither the event queue, nor the current trapped condition can be set explicitly by Rexx clauses. They can only be set as a result of incidents, when raising and trapping conditions.

It sounds very theoretical, doesn't it? Look at the following examples, which sets the trap MYTH:

```
/* 1 */ SIGNAL ON MYTH NAME TRAP_IT
/* 2 */ SIGNAL OFF MYTH
/* 3 */ CALL ON MYTH NAME MYTH_TRAP
/* 4 */ CALL ON MYTH
/* 5 */ CALL OFF MYTH
```

Line 1 sets state to ON, method to SIGNAL and handler to TRAP_IT. Line 2 sets state to OFF, handler and method becomes undefined. Line 3 sets state to ON, method to CALL, and handler to MYTH_TRAP. Line 4 sets state to ON, method to CALL and handler to MYTH (the default). Line 5 sets state to OFF, handler and method become undefined.

Why should method and handler become undefined when the trap in in state OFF? For two reasons: firstly, these values are not used when the trap is in state OFF; and secondly, when you set the trap to state ON, they are redefined. So it really does not matter what they are in state OFF.

What happens to this information when you call a subroutine? All information about traps are inherited by the subroutine, provided that it is an internal routine. External routines do not inherit any information about traps, but use the default values. Note that the inheritance is done by copying, so any changes done in the subroutine (internal or external), will only have effect until the routine returns.

1.26 How to Raise a Condition

How do you raise a condition? Well, there are really no explicit way in Rexx to do that. The conditions are raised when an incident occurs. What sort of situations that is, depends on the context. There are in general three types of incidents, classified by the origin of the event:

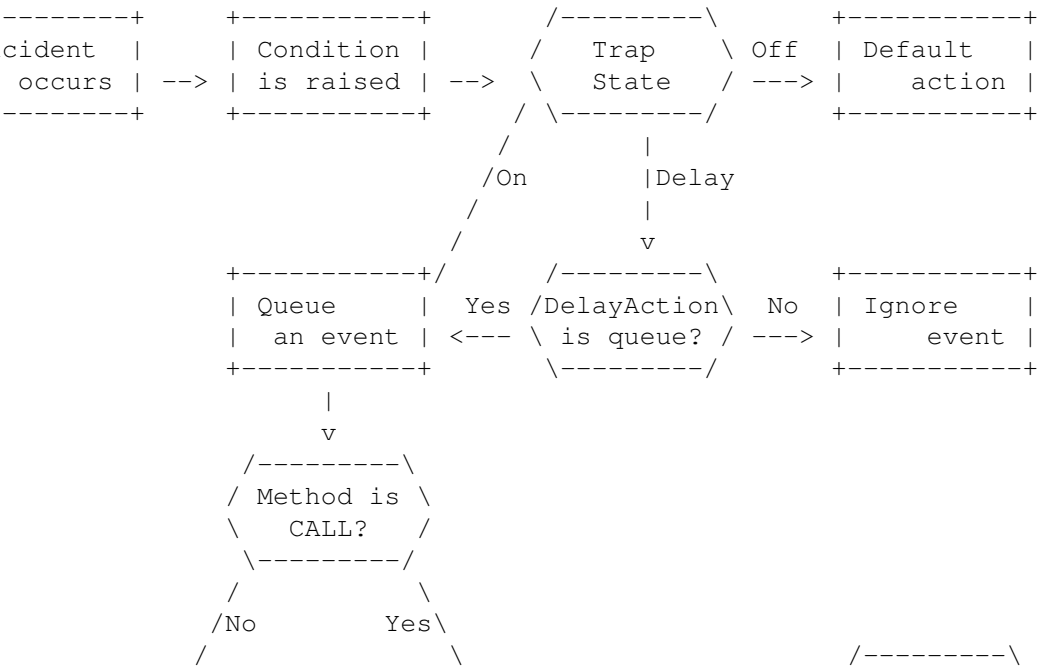
* Internal origin. The incident is only dependent on the behavior of the Rexx script. The SYNTAX condition is of this type.

* External origin. The Rexx script and the interpreter has really no control over when this incident. It happens completely independent of the control of the Rexx script or interpreter. The HALT condition is of this type.

* Mixed origin. The incident is of external origin, but the situation that created the incident, was an action by the Rexx script or the interpreter. The ERROR condition is of this type: the incident is a command returning error, but it can only occur when the interpreter is executing commands.

For conditions trapped by method CALL, standard Rexx requires an implementation to at least check for incidents and raise condition at clause boundaries. (But it is allowed to do so elsewhere too; although the actual triggering must only be performed at clause boundaries.) Consequently, you must be prepared that in some implementations, conditions trappable by method CALL might only be raised (and the trap triggered) at clause boundaries, even if they are currently trapped by method SIGNAL.

The six standard conditions will be raised as result of various situations, read the section describing each one of them for more information.



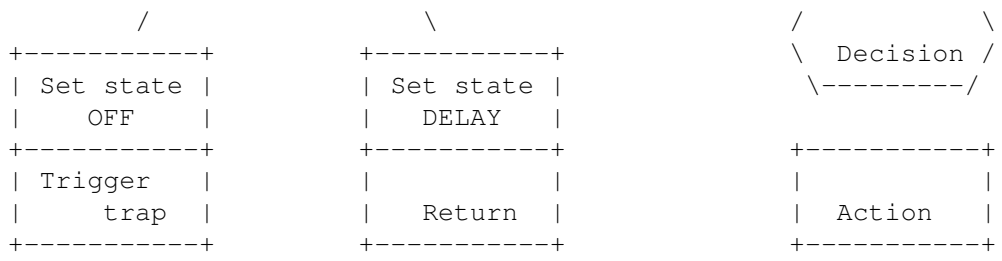


Figure 3.1: The triggering of a condition

When an incident occurs and the condition is raised, the interpreter will check the state of the condition trap for that particular condition at the current procedure level.

* If the trap state is OFF, the default-action of the condition is taken immediately. The "standard" default-action is to ignore the condition.

* If the trap state is DELAY, the action will depend on the delay-action of that condition. The standard delay-action is to ignore, then nothing further is done. If the delay-action is to queue, the interpreter continues as if the state was ON.

* If the state of the trap is ON, an event is generated which describes the incident, and it is queued in the pending event queue. The further action will depend on the method of trapping.

  If the method is CALL, the state of the trap will be set to DELAY. Then the normal execution is resumed. The idea is that the interpreter will check the event queue later (at a clause boundary), and trigger the appropriate trap, if it finds any events in the event queue.

  Else, if method of trapping is SIGNAL, then the action taken is this: First set the trap to state OFF, then terminate clause the interpreter was executing at this procedure level. Then it explicitly trigger the condition trap.

This process has be shown in fig. 3.1. It shows how an incident makes the interpreter raise a condition, and that the state of the condition trap determines what to do next. The possible outcomes of this process are: to take the default-action; to ignore if delay-action is not to queue; to just queue and the continue execution; or to queue and trigger the trap.

1.27 How to Trigger a Condition Trap

What are the situations where a condition trap might be triggered? It depends on the method currently set in the condition trap.

If the method is SIGNAL, then the interpreter will explicitly trigger the relevant trap when it has raised the condition after having sensed the incident. Note that only the particular trap in question will be triggered in this case; other traps will not be triggered, even if the pending event queue is non-empty.

In addition, the interpreter will at each clause boundary check for any pending events in the event queue. If the queue is non-empty, the interpreter will not immediately execute the next normal statement, but it will handle the condition(s) first. This procedure is repeated until there are no more events queued. Only then will the interpreter advance to execute the next normal statement.

Note that the Rexx standard does not require the pending events to be handled in any particular order, although the model shown in this documentation it will be in the order in which the conditions were raised. Consequently, if one clause generates several events that raise conditions before or at the next clause boundary, and these conditions are trapped by method CALL. Then, the order on which the various traps are triggered is implementations-dependent. But the order in which the different instances of the same condition is handled, is the same as the order of the condition indicator queue.

1.28 Trapping by Method SIGNAL

Assume that a condition is being trapped by method SIGNAL, that the state is ON and the handler is MYTH_TRAP. The following Rexx clause will setup the trap correctly:

```
SIGNAL ON MYTH NAME MYTH_TRAP
```

Now, suppose the MYTH incident occurs. The interpreter will sense it, queue an event, set the trap state to OFF and then explicitly trigger the trap, since the method is SIGNAL. What happens when the trap is triggered?

* It collects the first event from the queue of pending events. The information is removed from the queue.

* The current trapped condition is set to the information removed from the pending event queue.

* Then, the interpreter simulates a SIGNAL clause to the label named by trap handler of the trap for the condition in question.

  As all SIGNAL clauses, this will have the side-effects of setting the SIGL special variable, and terminating all active loops at the current procedure level.

That's it for method SIGNAL. If you want to continue trapping condition MYTH, you have to execute a new SIGNAL ON MYTH clause to set the state of the trap to ON. But no matter how quick you reset the trap, you will always have a short period where it is in state OFF. This means that you can not in general use the method SIGNAL if you really want to be sure that you don't loose any MYTH events, unless you have some control over when MYTH condition may arise.

Also note that since the statement being executed is terminated; all active loops on the current procedure level are terminated; and the only indication where the error occurred is the line number (the line may contain several clauses), then it is in general impossible to pick up the normal execution after a condition trapped by SIGNAL. Therefore, this method is best suited

for a "graceful death" type of traps. If the trap is triggered, you want to terminate what you were doing, and pick up the execution at an earlier stage, e.g. the previous procedure level.

1.29 Trapping by Method CALL

Assume that the condition MYTH is being trapped by method CALL, that the state is ON and the handler is MYTH_HANDLER. The following Rexx clause will setup the trap correctly:

```
CALL ON MYTH NAME MYTH_HANDLER
```

Now, suppose that the MYTH incident occurs. When the interpreter senses that, it will raise the MYTH condition. Since the trap state is ON and the trap method is CALL, it will create an event and queue it in the pending event queue and set the trap state to DELAY. Then it continues the normal execution. The trap is not triggered before the interpreter encounters the next clause boundary. What happens then?

* At the every clause boundaries, the interpreter check for any pending events in the event queue. If one is found, it is handled. This action is done repeatedly, until the event queue is empty.

* It will simulate a normal function call to the label named by the trap handler. As with any CALL clause, this will set the special variable SIGL to the line of from which the call was made. This is done prior to the call. Note that this is the current line at the time when the condition was raised, not when it was triggered. All other actions normally performed when calling a subroutine are done. Note that the arguments to the subroutine are set to empty.

* However, just before execution of the routine starts, it will remove the first event in the pending event queue, the information is instead put into the current trapped condition. Note that the current trapped condition is information that is saved across subroutine calls. It is set after the condition handler is called, and will be local to the condition handler (and functions called by the condition handler). To the "caller" (i.e. the procedure level active when the trap was triggered), it will seem as if the current trapped condition was never changed.

* Then the condition handler finishes execution, and returns by executing the RETURN clause. Any expression given as argument to RETURN will be ignored, i.e. the special variable RESULT will not be set upon return from a condition handler.

* At the return from the condition handler, the current trapped condition and the setup of all traps are restored, as with a normal return from subroutine. As a special case, the state of the trap just triggered, will not be put back into DELAY state, but is set to state ON.

* Afterwards (and before the next normal clause), the interpreter will again check for more events in the event queue, and it will not continue on the Rexx script before the queue is empty.

During the triggering of a trap by method CALL at a clause boundary, the state of the trap is not normally changed, it will continue to be DELAY, as was set when the condition was raised. It will continue to be in state DELAY until return from the condition handler, at which the state of the trap in the caller will be changed to ON. If, during the execution of the condition trap, the state of the condition being trapped is set, that change will only last until the return from the condition handler.

Since new conditions are generally delayed when an condition handler is executing, new conditions are queued up for execution. If the trap state is changed to ON, the pending event queue will be processed as named at the next clause boundary. If the state is changed to OFF, the default action of the conditions will be taken at the next clause boundary.

1.30 The Current Trapped Condition

The interpreter maintains a data structure called the current trapped condition. It contains information relating the the most recent condition trapped on this or higher procedure level. The current trapped condition is normally inherited by subroutines and functions, and restored after return from these.

* When trapped by method SIGNAL the current trapped condition of the current procedure level is set to information describing the condition trapped.

* When trapped by method CALL, the current trapped condition at the procedure level which the trap occurred at, is not changed. Instead, the current trapped condition in the condition handler is set to information describing the condition.

The information stored in the current trapped condition can be retrieved by the builtin function CONDITION(). The syntax format of this function is:

```
CONDITION(option)
```

where option is an option string of which only the first character matters. The valid options are: Condition name, Description, Instruction and State. These will return: the name of the current trapped condition; the descriptive text; the method; and the current state of the condition, respectively. The default option is Instruction. See the documentation on the builtin functions. See also the description of each condition below.

Note that the State option do not return the state at the time when the condition was raised or the trap was triggered. It returns the current state of the trap, and may change during execution. The other information in the current trapped condition may only change when a new condition is trapped or at treturn from subroutines.

1.31 The Real Conditions

We have now described how the standard condition and condition trap works in
Rexx. Let's look at the six conditions defined which do exist. Note that none of these behaves exactly as the standard condition.

The SYNTAX condition

The HALT condition

The ERROR condition

The FAILURE condition

The NOVALUE condition

The NOTREADY condition

1.32 The SYNTAX condition

The SYNTAX condition is of internal origin, and is raised when any syntax or runtime error is discovered by the Rexx interpreter. It might be any of the situations that would normally lead to the abortion of the program and the report of a Rexx error message, except error message number 4 (Program interrupted), which is handled by the HALT condition.

There are several differences between this condition and the standard condition:

* It is not possible to trap this condition with the method CALL, only method SIGNAL. The reason for this is partly that method CALL tries to continue execution until next boundary before triggering the trap. That might not be possible with syntax or runtime errors.

* When this condition is trapped, the special variable RC is set to the Rexx error number of the syntax or runtime error that caused the condition. This is done just before the setting of the special variable SIGL.

* The default action of this condition if the trap state is OFF, is to abort the program with a traceback and error message.

* There is not delay-action for condition SYNTAX, since it can not be trapped by method CALL, and consequently never can get into state DELAY.

The descriptive text returned by CONDITION() when called with the Description option for condition SYNTAX, is implementation dependent, and may also be a nullstring. Consult the implementation-specific documentation for more information.

1.33 The HALT condition

The HALT condition of external origin, which is raised as a result of an action from the user, normally a combination of keys which tries to abort the program. Which combination of keys will vary between operating systems. Some systems might also simulate this event by other means than key combinations. Consult system for more information.

The differences between HALT and the standard condition are:

* The default-action for the HALT condition is to abort execution, as though a Rexx runtime error error number 4 (Program interrupted) had been reported. But note that SYNTAX will never be raised if HALT is not trapped.

* The delay-action of this condition is to ignore, not queue.

The standard allows the interpreter to limit the search for situations that would set the HALT condition, to clause boundaries. As a result, the response time from pressing the key combination to actually raising the condition or triggering the trap may vary, even if HALT is trapped by method SIGNAL. If a clause for some reason has blocked execution, and never finish, you may not be able to break the program.

The descriptive text returned by CONDITION() when called with the Description option for condition HALT, is implementation dependent, and may also be a nullstring. In general, it will describe the way in which the interpreter was attempted halted, in particular if there are more than one way to do raise a HALT condition. Consult the implementation documentation for more information.

1.34 The ERROR condition

The ERROR is a condition of mixed origin, it is raised when a command returns a return value which indicates error during execution. Often, commands return a numeric value, and a particular value is considered to mean success. Then, other values might raise the ERROR condition.

Differences between ERROR and the standard condition:

* The delay action of ERROR is to ignore, not to queue.

* The special variable RC is always set before this condition is raised. So even if it is trapped by method SIGNAL, you can rely on RC to be set to the return value of the command.

Unfortunately, there is no universal standard on return values. As stated, they are often numeric, but some operating system use non-numeric return values. For those which do use numeric values, there are no standard telling which values and ranges are considered errors and which are considered success. In fact, the interpretation of the value might differ between commands within the same operating system.

Therefore, it is up to the Rexx implementation to define which values and ranges that are considered errors. You must expect that this information can differ between implementations as well as between different environments within one implementation.

The descriptive text returned by CONDITION() when called with the Description option for condition ERROR, is the command which caused the error. Note that this is the command as the environment saw it, not as it was entered in the Rexx script source code.

1.35 The FAILURE condition

The FAILURE is a condition of mixed origin, it is raised when a command returns a return value which indicates failure during execution, abnormal termination, or when it was impossible to execute a command. It is a subset of the ERROR condition, and if it is in state OFF, then the ERROR condition will be raised instead. But note that an implementation is free to consider all return codes from commands as ERRORs, and none as FAILURES. In that case, the only situation where a FAILURE would occur, is when it is impossible to execute a command.

Differences between FAILURE and the standard condition:

* The delay action of FAILURE is to ignore, not to queue.

* The special variable RC is always set before this condition is raised. So even if it is trapped by method SIGNAL, you can rely on RC to be set to the return value of the command, or the return code that signalize that the command was impossible to execute.

As for ERROR, there is no standard the defines which return values are failures and which are errors. Consult the system and implementation independent documentation for more information.

The descriptive text returned by CONDITION() when called with the Description option for condition FAILURE, is the command which caused the error. Note that this is the command as the environment saw it, not as it was entered in the Rexx script source code.

1.36 The NOVALUE condition

The NOVALUE condition is of internal origin. It is raised in some circumstances if the value of an unset symbol (which is not a constant symbol) is requested. Normally, this would return the default value of the symbol. It is considered bad programming practice not to initialize variables, and setting the NOVALUE condition is one method of finding the parts of your program that uses this programming practice.

Note however, there are only three instances where this condition may be raised: that is when the value of an unset (non-constant) symbol is used

requested: in an expression; after the VAR subkeyword in a PARSE clause; and as an indirect reference in either a template, a DROP or a PROCEDURE clause. In particular, this condition is not raised if the VALUE() or SYMBOL() builtin functions refer to an unset symbol.

Differences between NOVALUE and the standard condtition are:

* It may only be trapped by method SIGNAL, never method CALL. This requirement might seem somewhat strange, but the idea is that since an implementation is only forced to check for conditions trapped by method CALL at clause boundaries, incidences that may occur at any point within clauses (like NOVALUE) can only be trapped by method SIGNAL. (However, condition NOTREADY can occur within a clause, and may be trapped by method CALL so this does not seem to be absolute consistent.)

* There is not delay-action for condition NOVALUE, since it can not be trapped by method CALL, and consequently never can get into state DELAY.

The descriptive text returned by calling CONDITION() with the Description option, is the derived (i.e. tail has be substituted if possible) name of the variable that caused the condition to be raised.

1.37 The NOTREADY condition

The condition NOTREADY is a condition of mixed origin. It is raised as a result of problems with stream I/O. Exactly what causes it, may vary between implementations, but some of the more probable causes are: waiting for more I/O on transient streams; access to streams not allowed; I/O operation would block if attempted; etc. See the chapter on stream I/O for more information.

Differences between NOTREADY and the standard condition are:

* It will be ignored rather than queued if condition trap is in state DELAY.

* This condition differs from the rest in that it can be raised during execution of a clause, but can still be trapped by method CALL.

The descriptive text returned by CONDITION() when called with the Description option for condition NOTREADY, is the name of the stream which caused the problem. This is probably the same string that you used as the first parameter to the functions that operates on stream I/O. For the default streams (default input and output stream), the string returned by CONDITION() will be nullstrings.

Note that if the NOTREADY trap is in state DELAY, then all I/O for files which has tried to raise NOTREADY within the current clause will be simulated as if operation had succeeded.

1.38 Further Notes on Conditions

1.39 Conditions under Language Level 3.50

The concept of conditions was very much expanded from Rexx language level 3.50 to level 4.00. Many of the central features in conditions are new in level 4.00, these include:

* The CALL method is new, previously only the SIGNAL method was available, which made it rather difficult to resume execution after a problem. As a part of this, the DELAY state has been added too.

* The condition NOTREADY has been added, to allow better control over problems involving stream I/O.

* The builtin function CONDITION() has been added, to allow extraction of information about the current trapped condition.

1.40 Pitfalls when Using Condition Traps

There are several pitfalls when using conditions:

* Remember that some information are saved across the functions. Both the current trapped condition and the settings of the traps. Consequently, you can not set a trap in a procedure level from a lower level. (I.e. calling a subroutine to set a trap is will not work.)

* Remember that SIGL is set when trapped by method CALL. This means that whenever a condition might be trapped by CALL, the SIGL will be set to a new value. Consequently, never trust the contents of the SIGL variable for more than one clause at a time. This is very frustrating, but at least it will not happen often. When it do happen, though, you will probably have a hard time debugging it.

* Also remember that if you use the PROCEDURE clause in a condition handler called by method CALL, remember to EXPOSE the special variables SIGL if you want to use it inside the condition handler. Else it will be shadowed by the PROCEDURE.

1.41 The Correctness of this Description

In this description of conditions in Rexx, I have gone further in the description of how conditions work, their internal data structures, the

order in which things are executed etc, than the standard does. I have tried to interpret the set of distinct statements that is the documentation on condition, and design a complete and consistent system describing how such conditions work. I have done this to try to clarify an area of Rexx which at first glance is very difficult and sometimes unintuitive.

I hope that the liberties I have taken have helped describe conditions in Rexx. I do not feel that the adding of details that I have done in any way change how conditions work, but at least I owe the reader to list which concepts that are genuine Rexx, and which have been filled in by me to make the picture more complete. These are not a part of the standard Rexx.

* Rexx does not have anything called a standard condition. There just "are" a set of conditions having different attributes and values. Sometimes there are default values to some of the attributes, but still the are no default condition.

* The terms "event" and "incident" are not used. Instead the term "condition" is somewhat overloaded to mean several things, depending on the situation. I have found it advantageous to use different terms for each of these concepts.

* Standard Rexx does not have condition queue, although a structure of such a kind is needed to handled the queuing of pending conditions when the trap state is DELAY.

* The values default-action and delay-action are really non-existing in the Standard Rexx documentation. I made them up to make the system more easy to explain.

* The two-step process of first raising the flag, and then (possibly at a later stage) triggering the trap, is not really a Rexx concept. Originally, Rexx seems to allow implementations to select certain places of the interpreter where events are sought for. All standard conditions that can be called by method CALL, can be implemented by checking only at clause boundaries.

  Consequently, a Rexx implementation can choose to trigger the trap immediately after a condition are raised (since conditions are only raised immediately before the trap would trigger anyway). This is also the common way used in language level 3.50, when only method SIGNAL was implemented.

  Unfortunately, the introduction of the state DELAY forces the interpreter to keep a queue of pending conditions, so there is nothing to gain on insisting that raising should happen immediately before triggering. And the picture is even more mudded when the NOTREADY condition is introduced. Since it explicitly allows raising of condition to be done during the clause, even though the triggering of the trap must happen (if method is CALL) at the end of the clause.

I really hope that these changes has made the concept of conditions easier to understand, not harder. Please feel free to flame me for any of these which you don’t think is representative for Rexx.

1.42 Conditions in Regina

Here comes documentation that are specific for the Regina implementation of Rexx.

1.43 How to Raise the HALT condition

The implementation connect the HALT condition to an external event, which might be the pressing of certain key combination. The common conventions of the operating system will dictate what that combination of keystrokes is.

Below is a list, which describes how to invoke an event that will raise the HALT condition under various the operating systems which Regina runs under.

* Under various variants of the Unix operating system, the HALT event it connected to the signal "interrupt" (SIGINT). Often this signal is bound to special keystrokes. Depending on your version of Unix, this might be ctrl-c (mostly BSD-variants) or the del key (mostly System V). It is also possible to send this signal from the command line, in general using the program kill(1); or from program, in general using the call signal(3). Refer to your Unix documentation for more information.

* Under VAX/VMS, the keysequence ctrl-c is used to raise the HALT condition in the interpreter.

1.44 Extended builtin functions

Regina has a few extra builtin functions that are added to support the debugging of the interpreter. Under some circumstances, these might also be useful when debugging Rexx scripts. Note that these functions are not a part of standard Rexx and should never be used when portability is required. The functions are:

RAISE_COND(condition)
    is used to explicitly raise a condition during execution of a Rexx script. The interpreter will accept the execution of this function as an event, just as if the event had occurred. Returns the nullstring.

COND_INFO([condition])
    is a function that will return information about the current settings of

the condition indicator for condition, including the state of the flag, and the contents of the pending queue. If called without a parameter, it will return a space-separated list of those conditions which have non-empty pending condition queue.

TRAP_INFO([condition])
is a functions that returns the status information about a trigger at the current procedure level. The info returned will be the state, the method and the condition handler. If called without a parameter, a space-separated list of condition enabled (state ON or DELAY) at the current procedure level, is returned.

These functions are described in detail elsewhere. Note that these functions will only be available if the interpreter was compiled with the certain preprocessor flag set. If the code was included in the compilation, the availability of these function will still be dependent on the selection of extensions with the clause OPTIONS, where the extension DBG_FUNCS should be chosen. See chapter on extensions for more information.

1.45 Extra Condition in Regina

Regina has some other extra conditions. These conditions are:

* A condition DEBUG, that is very similar to the condition HALT. The condition is raised as a result of an event of external origin, generally a special combination of keystrokes is pressed.

  The default-action of this condition is to set the trace mode to Normal and interactive. Consequently, the user will generally get into interactive tracing at the next clause boundary. This way, the user may be able to stop the program during execution, and perform debugging.

  The delay-action of this condition is to ignore it.

  On Unix machines, this is the signal QUIT (SIGQUIT), which is normally bound the the ctrl-\ key. Just like condition HALT, this might also be simulated from the command line, or from other programs. Consult the Unix documentation for more information. On VAX/VMS machines, this event is normally bound to the ctrl-y key.

  This extended condition will only be available if the extension COND_DEBUG has been chosen.

Whether or not the conditions listed here are available, may also depend on whether particular preprocessor flag was set during compilation. For more information, see the chapter on extensions.

1.46 Various Other Existing Extensions

Here is a list of other current extensions in Regina. See chapter on extensions for more information.

* Regina allows the condition NOVALUE to be trapped by method CALL, which is not allowed according to the standard. This extension will only be available if the extension CALL_ON_NOVALUE has been chosen, and the code was compiled with certain preprocessor flags set.

  If NOVALUE is being trapped by method CALL, the current clause will be completed as if NOVALUE was not trapped at all, returning the default value for an unset symbol as variable value.

1.47 Possible Future extensions

Here is a list of possible future extensions to Rexx which has not been implemented into Regina. Some of these exist in other implementations of Rexx, and some of them are just suggestions or ideas thrown around by various people.

* Another extension could have been included, but have been left out so far. It is the delay-action, which in standard Rexx can be either to ignore or to queue. There is at least one other action that make sense: to replace. That is, when a trap is in state DELAY, and a new condition has been raised, the pending queue is emptied, before the new condition is queued. That way, the new condition will effectively replace any conditions already in the queue.

  If there are several new conditions raised while the condition handler is executing (and the trap state is DELAY), only the very last of them is remembered.

* It should be possible to set the state for a trap to DELAY, so that any new instances of the condition is handles by the delay-action. As a special case, the SYNTAX condition trap might not be set in state DELAY.

1.48 Chapter 4

FILE INPUT AND OUTUT

In this chapter treats input and output to files using the built-in functions. An overview over the other parts of the I/O system are given, but it is not discussed in detail.

Rexx's Notion of a

1.49 Rexx's Notion of a

Rexx regards a stream as a sequence of characters, conceptually equivalent to what a user might type at his keyboard. Note that this definition of a "stream" is not exactly equivalent to what is generally known as a "file". A "file" refers to a collection of information stored on a storage device (e.g. disk or tape); while a "stream" is more like the interface to "file".

There are basically two ways of reading and writing a stream: line-wise, and character-wise. When reading line-wise, the underlaying storage method of the stream has embedded information describing where each lines starts and ends.

Some filesystems store this information as one or more special characters, while others structure the file into "records"; each containing a single line. This introduces a slightly subtle point; even though a stream A returns same data when read by LINEIN() on two different machines; the data read from A may differ between the two machines when the stream is read by CHARIN(). This is so because the end-of-line markers may vary between the two operating systems.

Therefore, for maximum portability, the line-oriented built-in functions (LINEIN(), LINEOUT() and LINES()) should only be used for line-oriented streams. While the character-oriented built-in functions (CHARIN(), CHAROUT() and CHARS) should only be used for character-oriented data. This means that you can't mix character- and line-oriented data in a single stream, and simultanously maintain maximum portability.

The difference between character- and line-oriented streams are roughly the equivalent to the difference between binary and text streams.

The end-of-file marker may be differently implemented on different systems. On some systems, this marker is implicitly present at the end-of-file --- which is calculated from the size of the file (e.g. Unix). Other systems may put a character signifying end-of-file at the end of the file. These concepts vary between operating systems, any you can only hope that the interpreter

handles each concept intelligently. Check with the implementation-specific documentation for further information.

1.50 Positioning within a File

As mentioned, Rexx supports two strategies for reading and writing streams: character-wise, and line-wise.

For each open file, there is a current read position or a current write position, depending on whether the file is open for read or write. If the file is open for read and write simultaneously, it will have both a current read position and a current write position, and the two will be independant and --- in general --- different.

Please note that Rexx starts numbering at 1, not 0. Therefore, the first character and the first line in a file are both numbered 1. This differs from several other programming languages, which starts numbering at 0.

Just after a stream has been opened, the initial values of the current read position is at (or before) the first character in the file, while the current write position is the end-of-file, i.e. the position just after the last character in the file. Then, reading will return the first character (or line) in the stream, and writing will append the new character (or line) to the stream.

These 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 the current read and write positions. See your 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 number of characters in the stream. In particular, it is illegal to set the current read position to the position after the last character in the stream; although this is legal in some 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, see above. In addition, and unlike the current read position, the current write position may 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 it is not allowed to reposition the current write position further out after the end-of-file --- which would create a "hole" in the stream --- even though this is allowed in some other languages and operating systems.

Rexx only keeps one current read position and one current write position for each stream. So both line-wise and character-wise reading (and positioning of the current read position) will operatate on the same read current position. And similar for the current write position. When repositioning using lines, the current read or write position will be set to the first character in the

next line to be processed.

Note that if you want to reposition the current write position using a line count, the stream will probably have to be open for read too. This is because the operating system may need to read the contents of the stream in order to find the lines. Depending on your operating system, this may even apply if you reposition using character count.

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.

1.51 Persistent and Transient Streams

Rexx knows two different types of streams: persistent streams and transient streams. They differ conceptually in the way they can be operated on, 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-oriented data), and both are read and written using the same functions.

Persistent streams (often referred to just as "files") are conceptually stored in a permanent storage in the computer (e.g. a disk), as an array of characters. Random access to any part of the stream is allowed for persistent files.

Transient streams are typically not available for random access, either becuase it is not stored permanently, but read as a sequence of data; or because it is available as a sequential storage (e.g. magnetic tape) where random access is difficult. Typical examples of transient files might be devices like the keyboards and printers, communication interfaces, pipelines, etc.

Rexx does not allow any sort of repositioning on transient stream; such operations are not meaningful for a conceptually transient stream. It is possible to read a persistent stream as a transient stream, but not vice versa. Some implementations therefore may allow you to open a persistent stream as transient. One good candidate for such a treatment are files to which you have just append-only (i.e. you are not allowed to read, and writes can only be performed at the end of file).

1.52 Errors: Discovery, Handling and Recovery

TRL2 contains several important improvement over TRL1 in the area of handling errors in file I/O. The two most important differences are the NOTREADY condition and the STREAM() built-in function.

You discover that an error occurred in a I/O operation in one of the following ways: It may trigger a SYNTAX condition; it may trigger a NOTREADY condition; or it may just not return that data it was supposed to. There is no clear border between which situations that should trigger SYNTAX and which

should trigger NOTREADY. Error in parameters to the I/O functions, like a negative start position, are clearly a SYNTAX condition, while reading off the end-of-line are equally clearly a NOTREADY condition. Inbetween lays more uncertain situations like trying to position current read position after the end-of-file, or trying to read a non-existent file or using an illegal filename.

These situations are likely to be differently handled in various implementations, but you can assume that they are handled as either SYNTAX or CONDITION. Defencive programming requires you to check for both of these.

If not trapped, SYNTAX will terminate the program while NOTREADY will be ignored, so the implementors decision about which of these to use may even depend on the severity of the problem (i.e. if the problem is not big, raising SYNTAX may be a little to strict).

Personally, I think SYNTAX should be raised only if the parameters is outside the valid range for all contexts.

1.53 Naming Files

Ulike other programming languages, Rexx does not use filepointers; the name of the file is supplied to all I/O functions operating on that file. However, under the surface, the Rexx interpreter is likely to use native filepointers of the operating system, in order to improve speed.

The format of the file name will be very dependent upon the operating system you are running. For portability concerns, you should try not to specify the file name as a literal string in the I/O calls, but set a variable to the file name, and use the variable when calling I/O functions. That will reduce the number of place you need to change if you need to port the program to another system.

If the stream name is omitted from the built-in I/O functions (the built-in I/O functions are CHARIN(), CHAROUT(), CHARS(), LINEIN(), LINEOUT() and LINES()), a default stream is used: input functions use the default input stream, while output functions use the default output stream.
Unfortunately, there is no standard way to explicitly specify a stream name as the default input or output stream. And consequently, there is no standard way to refer to the default input or output stream in the built-in function STREAM(). You must refer to implementation specific documentation for information about this. But the interpreter itself, or the operating system may have means to connect any file or stream to the default I/O stream.

The use of file names instead of file descriptors are deeply rooted in the Rexx philosophy: Datastructures are text string carrying the information, rather than opaque datablocks in internal, binary format. This opens for some intriguing possibilities. I many 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 are the same file, although a Rexx interpreter are likely to interpret them as three different files, since the names themselves differ.

1.54 Non-standard Operations on Files

1.55 Where Implementations are Allowed to Differ

The TRL is rather relaxed in its specifications of what an implementation 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 that number, these functions may return "1" unless it is absolutely certain that there are no more data left. This leads to some rather klugdy programming techniques.

* Rexx does not contain a standard way to flush or close files. There is one method available to the implementors: If you call either CHAROUT() or LINEOUT(), but omit the data to be written, and you don't specify a new current write position, the implementation are allowed by TRL to do something "magic". It is not explicitly defined what this magic is, but TRL suggests that it may be closing the file, flushing the file or committing changes done to the file.

  You should always confer the implementation specific documentation before using this feature. The difference in the action of closing and flushing a file, can make a Rexx script that works under one implementation totally crash under another, so this feature is of very limited value if you are trying to write portable programs.

1.56 Where Implementations might Differ anyway

In the section above, some areas where the stanard 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 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.

1.57 Typical Problems when Handling Files

```
                This is that section of peculiar behavior. What sort of behavior ↩
                   do you ought
to avoid in order to keep out of trouble.
```


1.58 The Stream was Renamed During Execution

Suppose you start reading from a stream, let us call it 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. What will happen?

Strictly speaking, the file foo does not exist anymore, so the second read should raise the NOTREADY condition. However, your Rexx interpreter is likely to have opened the file 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 exist before each I/O operation (that would require a lot of extra checking). On a large number of 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 it has changed its name.

1.59 LINES() and CHARS() are Inaccurate

1.60 If You don’t Close Your Files

Rexx has a "auto-open-on-demand" feature, which enables the user to use streams without opening them first. Some Rexx interpreters may require you to perform some implementation dependant operation before accessing streams, but most implementations are likely to open any stream when it is first referenced in an I/O function.

However, the Rexx interpreter will never know when to close a stream, unless explictly told. It can never foretell when a particular stream is to be used next, and 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’ve 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 computer. The rule is that any Rexx script that uses more than a couple of streams, should close every stream after use, and try to keep the number of simultaneously open streams as low as possible.

A good Rexx interpreter might try to defend itself against this sort of open-many-close-none programming, by the use of various programming techniques.

1.61 Stream I/O in Regina

Regina tries to implement stream I/O which closely resembles how it is described in TRL.

* In order to defend itself against "open-many-close-none" programming, Regina tries to "swap out" files that has not been used for some time. Assume that your operating system limits Regina to 100 simultaneously open files, when your try to open your 101st file, Regina will close the least recently used file, and use that file descriptor for the new file.

  So, what happends if you decide to use the first file, the file that was just closed? Well, Regina only closes the file in the operating system, but retains all vital information about the file itself. If you reaccess the file again, Regina will reopen it, and position the current read and write position at the correct (i.e. previous) positions.

  This will introduce some uncertainty into file processing. Renaming a file will affect a file only if it gets swapped out (See 4.9.1). But it is better than terminating the interpreter because the system file table overflows.

  Regina will not allow any file to be swapped out. Transient file are not swapped, 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.

* Regina allows files to be explicitly opened and closed through the use of the built-in function STREAM(). The exact syntax of this function is described elsewhere. Old versions of Regina supported two non-standard built-in functions OPEN() and CLOSE() for these operations. These functions are still supported for compatibilty reasons, but might be removed in future releases.

* When line-repositioning the current write position to the middle of a file, Regina will truncate the file at that point. If it didn't, the file might contain half a line, and some lines might disappear, and the linecount would in general be disrupted. Unfortunately, the operation of truncating a file is not standard, and it might not exist on all systems, so on some rare systems, this truncating will not take place.

1.62 Chapter 5

Interfacing Rexx to other programs

This chapter describes an interface between a Rexx interpreter and another

program, typically written in C or another high level, compiled language. It is intended for application programmers who are implementing Rexx support in their programs. It describes the interface known as the Rexx SAA API.

1.63 Overview of functions in SAA

The functionality of the interface is devided into some main areas ↩ :

Subcommand handlers (section 5.3) which trap and handle a command to an external environment.

Interpreting Rexx scripts (section 5.4), either from a disk file, or from memory.

Variable interface (section 5.5), which makes it possible to access the variables in the interpreter, and allows operations like setting, fetching and dropping variables.

System exits (sections 5.2.2 and 5.6), which are used to hook into certain key points in the interpreter while it executes a script.

In the following sections each of these areas are described in detail, and a number of brief but complete examples are given at the end of the chapter.

The description is of a highly technical nature, since it is assumed that the reader will be an application programmer seeking information about the interface. Therefore, much of the content is given as prototypes and C style datatype definitions. Although this format is cryptic for non-C programmers, it will convay exact, compact, and complete information to the intended readers. Also, the problems with ambiguity and incompleteness that often accompany a descriptive prose text are avoided.

1.64 Include Files and Libraries

All the C code that uses the Rexx application interface, must include a special header file that contains the necessary definitions. This file is called rexxsaa.h. Where you will find this file, will depend on you system and which compiler you use.

Also, the interface part between the application and the Rexx interpreter may be implemented as a library, which you link with the application using the functions described in this chapter. The name of this library, and its location might differ from system to system. With Regina, it is called libregina.a, and is produced when you compile Regina.

1.65 Preprocessor Symbols

Including a header file ought to be enough; unfortunately, that is not so. Each of the domains of functionality listed above are defined in separate "sections" in the rexxsaa.h header file. In order for these to be made available, certain proprocessor symbols have to be set. For instance, you have to include the following definition:

```
#define INCL_RXSHV
```

in order to make available the definitions and datatypes concerning the variable pool interface. The various definitions that can be set are:

INCL_RXSUBCOM Must be defined in order to get the prototypes, datatypes and symbols needed for the subcommand interface of the API.

INCL_RXSYSEXIT Must be defined in order to get the prototypes, datatypes, and symbols needed for the system exit functions

INCL_RXSHV Must be set in order to get the prototypes, symbols and datatype definitions necessary to use the Rexx variable pool.

1.66 Allocating and Deallocating Space

For several of the functions described in this chapter, the application calling them must allocate or deallocate dynamic memory. Depending on the operating system, compiler and Rexx interpreter, the method for these allocations and deallocations might vary. Regina uses malloc() and free() in all these situations.

1.67 Datastructures

In this section, some datastructures relevant to the application interface to
Rexx are defined and described. These are:

RXSTRING Holds a Rexx string.

RXSYSEXIT Holds a definition of a system exit handler. Used when starting a Rexx script with RexxStart, and when defining the system exit handlers.

The RXSTRING structure

The RXSYSEXIT structure

1.68 The RXSTRING structure

The SAA API interface uses "Rexx strings" which are stored in the structure RXSTRING. There is also a datatype PRXSTRING, which is a pointer to RXSTRING. Their definitions are:

```
typedef struct {
   unsigned char *strptr ;    /* Pointer to string contents */
   unsigned long strlength ;  /* Length of string */
} RXSTRING ;

typedef RXSTRING *PRXSTRING ;
```

The strptr field is a pointer to an array of characters making up the contents of the "Rexx string", while strlength holds the number of characters in that array.

Unfortunately, there are some inconsistencies in naming of various special kinds of strings. In Rexx (TRL), a "null string" is a string that has zero length. On the other hand, the SAA API operates with two kinds of special strings: "null strings" and "zero length strings". The latter is a string with zero length (equals null strings in Rexx), while the former is a sort of "undefined" or "empty" string, which denotes a string without a value. The "null strings" of SAA API are used to denote unspecified values (e.g. a parameter left out in a subroutine call). In this chapter, when the terms "null strings" and "zero length strings" are quoted, they refer to the SAA API style meaning.

A number of macros are defined, which simplifies operations on RXSTRINGs for the programmer. In the list below, all parameters called x are of type RXSTRING.

MAKERXSTRING(x,content,length) The parameter contents must be a pointer to char, while length is integer. The x parameter will be set to the contents and length supplied. The only operations are assignments; no new space is allocated and the contents of the string is not copied.

RXNULLSTRING(x) Returns true only if x is a "null string". I.e. x.strptr is NULL.

RXSTRLEN(x) Returns the length of the string x as an unsigned long. Zero is returned both when x is a "null string" or a "zero length string".

RXSTRPTR(x) Returns a pointer to the first character in the string x, or NULL if x is a "null string". If x is a "zero length string", and non-NULL pointer is returned.

RXVALIDSTRING(x) Returns true only if x is neither a "null string" nor a "zero length string", i.e. x must have non-empty contents.

RXZEROLENSTRING(x) Returns true only if x is a "zero length string". I.e. x.strptr is non-NULL, and x.strlength is zero.

These definitions are most likely to be defined as preprocessor macros, so you should never "call" them with "parameters" having any side effects. Also note that at least MAKERXSTRING() is likely to be implemented as two statements, and might not work properly if following e.g. an if statement. Check the actual definitions in the rexxsaa.h header file before using them in a fancy context.

One definition of these might be (don't expect this to be the case with your implementation):

```
#define MAKERXSTRING(x,c,l) ((x).strptr=(c),(x).strlength=(l))
#define RXNULLSTRING(x)     (!(x).strptr)
#define RXSTRLEN(x)         ((x).strptr ? (x).strlength : 0UL)
#define RXSTRPTR(x)         ((x).strptr)
#define RXVALIDSTRING(x)    ((x).strptr && (x).strlength)
#define RXZEROLENSTRING(x)  ((x).strptr && !(x).strlength)
```

Note that these definitions of strings differ from the normal definition in C programs; where a string is an array of characters, and its length is implicitly given by a terminating ASCII NUL character. In the RXSTRING definition, a string can contain any character, including an ASCII NUL, and the length is explicitly given.

1.69 The RXSYSEXIT structure

This structure is used for defining which system exit handlers are to handle which system exits. The two relevant datatypes are defined as:

```
typedef struct {
   unsigned char *sysexit_name ;
   short sysexit_code ;
} RXSYSEXIT ;

typedef RXSYSEXIT *PRXSYSEXIT ;
```

In this structure, sysexit_name is a pointer to the ASCII NUL terminated string containing the name of a prevously registered (and currently active) system exit handler. The sysexit_code field is main function code of a system exit.

The system exits are divided into main functions and subfunctions. An exit is defined to handle a main function, and must thus handle all the subfunctions for that main function. All the functions and subfunctions are listed in the description of the EXIT structure.

1.70 The Subcommand Handler Interface

This sections describes the subcommand handler interface, which enables the application to trap commands in a Rexx script being executed and handle this commands itself.

1.71 What is a Subcommand Handler

A subcommand handler is a piece of code, that is called to handle a command to an external environment in Rexx. It must be either a subroutine in the application that started the interpreter, or a subroutine in a dynamic link library. In any case, when the interpreter needs to execute a command to an external environment, it will call the subcommand handler, passing the command as a parameter. Typically, an application will set up a subcommand handler before starting a Rexx script. That way, it can trap and handle any command being executed during the course of the script.

Each subcommand handler handles one environment, which is refered to by a name. It seems to be undefined whether upper and lower case letters differ in the environment name, so you should assume they differ. Also, there might be an upper limit for the length of an environment name, and some letters may be illegal as part of an environment name.

Regina allows any letter in the environment name, except ASCII NUL; and sets no upper limit for the length of an environment name. However, for compatibility reasons, you should avoid "uncommon" letters and keep the length of the name fairly short.

The prototype of a subcommand handler function is:

```
unsigned long handler(
   RXSTRING *command,
   unsigned short *flags,
```

```
      RXSTRING *returnstring
   ) ;
```

After registration, this function is called whenever the application is to handle a subcommand for a given environment. The value of the paramaters are:

command The command string that is to be executed. This is the resulting string after the command expression has been evaluated in the Rexx interpreter. It can not be empty, although it can be a "zero-length-string".

flags Points to a unsigned short which is to receive the status of the completion of the handler. This can be one of the three following: RXSUBCOM_OK, RXSUBCOM_ERROR, or RXSUBCOM_FAILURE. The contents will be used to determine whether to raise any condition at return of the subcommand. Do not confuse it with the return value.

returnstring Points to a RXSTRING which is to receive the return value from the subcommand. Passing the return value as a string makes it possible to return non-numeric return codes. As a special case, you might set returnstring.strptr to NULL, instead of specifying a return string of the ASCII representation of zero.

Note that it is not possible to return "nothing" in a subcommand, since this is interpreted as zero. Nor is is possible to return a numeric return code as such; you must convert it to ASCII representation before you return.

The returnstring string will provide a 256 byte array which the programmer might use if the return data is not longer that that. If that space is not sufficient, the handler can provide another area itself. In that case, the handler should not deallocate the default area, and the new area should be allocated in a standard fashion.

1.72 The RexxRegisterSubcomExe() function

This function is used to register a subcommand handler with the interface. The subcommand handler must be a procedure located within the code of the application. After registration, the Rexx interpreter can execute subcommands by calling the subcommand handler with parameters describing the subcommand.

The prototype for RexxRegisterSubcom() is:

```
   unsigned long RexxRegisterSubcomExe(
      signed char* EnvName,
      unsigned long (*EntryPoint)(),
      unsigned char* UserArea
   ) ;
```

All the parameters are input, and their significance are:

EnvName Points to an ASCII NUL terminated character string which defines the name of the environment to be registered. This is the same name as the Rexx interpreter uses with the ADDRESS clause in order to select an

external environment.

EntryPoint Points to the entrypoint of the subcommand handler routine for the environment to be registered. See the section on Subcommand Handlers for more information. There is an upper limit for the length of this name.

UserArea Pointer to an 8 byte area of information that is to be associated with this environment. This pointer can be NULL if no such area is necessary.

The areas pointed to by EnvName and UserArea are copied to a private area in the interface, so the programmer may deallocate or reuse the area used for these parameters after the call has returned.

The RexxRegisterSubcom returns an unsigned long, which carries status information describing the outcome of the operation. The status will be one of the RXSUBCOM values:

RXSUBCOM_OK The subcommand handler was sucessfully registered.

RXSUBCOM_DUP The subcommand handler was sucessfully registered. There already existed another subcommand handler which was registed with RexxRegisterSubcomDll(), but this will be shadowed by the newly registered handler.

RXSUBCOM_NOTREG Due to some error, the handler was not registered. Probably because a handler for EnvName was already defined at a previous call to RexxRegisterSubcomExe().

RXSUBCOM_NOEMEM The handler was not registed, due to lack of memory.

RXSUBCOM_BADTYPE Indicates that the handler was not registered, due to one or more of the parameters having invalid values.

1.73 The RexxRegisterSubcomDll function

This function is used to set up a routine that is located in a module in a dynamic link library, as a subcommand handler. Some operating systems don't have dynamic linking, and thus cannot make use of this facility. The prototype of this function is:

```
unsigned int RexxRegisterSubcomDll(
   unsigned char *EnvName,
   unsigned char *ModuleName,
   unsigned char *EntryPoint,
   unsigned char *UserArea,
   unsigned long DropAuth
) ;
```

This function is not yet supported by Regina.

1.74 The RexxDeregisterSubcom function

This function is used to remove a particular environment from the list of registered environments. The prototype of the function is:

```
unsigned int RexxDeregisterSubcom(
   signed char* EnvName,
   signed char* ModuleName
) ;
```

Both parameters are input values:

EnvName Pointer to ASCII NUL terminated string, which represents the name of the environment to be removed.

ModuleName Also an ASCII NUL terminated string, which points to the name of the module containing the subcommand handler of the environment to be deleted. Not currently in use by Regina.

The list of defined environments is searched, and if an environment matching the one named by the first parameter are found, it is deleted. The returnvalue will be a RXSUBCOM_ macro, which is OK if the environment was successfully deleted, NOTREG if the environment was not found, and BADTYPE if illegal parameters are found.

Most systems that do have dynamic linking have no method for reclaiming the space used by dynamically linked routines. So, even if you were able to load a "dll", there are no guarantees that you will be able to unload it.

1.75 The RexxQuerySubcom() function

This function retrieves information about a previously registered subcommand handler. The prototype of the function is:

```
unsigned int RexxQuerySubcom(
   signed char *EnvName,
   signed char *ModuleName,
   unsgined short *Flag,
   unsigned char *UserWord
) ;
```

Note that some of the documentation I've seen skips the Flag parameter. I don't know why, and I don't know what is correct, but I guess it should be there. The significance of the parameters are:

EnvName Pointer to an ASCII NUL terminated character string, which names the subcommand handler about which information is to be returned.

ModuleName Pointer to an ASCII NUL terminated character string, which names a dynamic link library. Only the named library will be searched for the subcommand handler named by EnvName. This parameter must be NULL if all subcommand handlers are to be searched.

Flag Pointer to a short which is to receive the value RXSUBCOM_OK or RX-
SUBCOM_NOTREG. In fact, this is the same as the return value from the
function.

UserWord Pointer to an area of 8 bytes. The "userarea" of the subcommand
handler is copied to the area pointed to by UserWord. This parameter
might be NULL if the data of the "userarea" is not needed.

Note that the datatype of the UserWord may differ. All seem to agree that
its size is 8 byte, but the datatype might differ between unsigned long
or unsigned char. You would do best in allocating it as 8 byte with good
alignment

The returned value from RexxQuerySubcom() can be one of:

SUBCOM_OK The subcommand handler was found, and the required information has
been returned in the Flag and UserWord variables.

SUBCOM_NOTREG The subcommand handler was not found. The Flag variable will
also be set to this value, and the UserWord variable is not changed.

SUBCOM_BADTYPE One or more of the parameters had illegal values, and the
operation was not carried through.

1.76 Executing Rexx Code

This sections describes the RexxStart() function, which allows the ↩
ap-
plication to startup the interpreter and make it interpret pieces of Rexx
code.

The RexxStart() function

1.77 The RexxStart() function

This function is used to invoke the Rexx interpreter in order to execute a
piece of Rexx code, which may be located on disk, as a pretokenized macro, or
as ASCII source code in memory.

```
long RexxStart(
   long ArgCount,
   struct rxstring *ArgList,
   char *ProgramName,
   struct rxstring *Instore,
   char *EnvName,
   long CallType,
   struct rxsysexit Exits,
   long *ReturnCode,
   strcut rxstring *Result
```

```
   ) ;
```

Of these parameters, ReturnCode and Result are output-only, while Instore is both input and output. The rest of the parameters are input-only. The significance of the parameters are:

ArgCount The number of parameter strings given to the procedure. This is the number of defined Rexx-strings pointed to by the ArgList parameter.

ArgList Pointer to an array of Rexx-strings, constituting the parameters to this call to Rexx. The size of this array is give by the parameter ArgCount. If ArgCount is greater than one, the first and last parameters are ArgList[0] and ArgList[ArgCount-1]. If ArgCount is 0, the value of ArgList is irrelevant.

If the strptr of one of the elements in the array pointed to by ArgList is NULL, that means that this parameter is empty (i.e. unspecified, as opposed to a string of zero size).

ProgName An ASCII NUL terminated string, specifying the name of the Rexx script to be executed. The value of Instore will determine whether this value is interpreted as the name of a (on-disk) script, or a pretokenized macro. If it refers to a filename, the syntax of the contents of this parameter depends on the operating system.

Instore Parameter used for storing tokenized Rexx scripts. This parameter might either be NULL, else it will be a pointer to two RXSTRING structures, the first holding the ASCII version of a Rexx program, the other holding the tokenized version of that program. See below for more information about how to use Instore.

EnvName Pointer to ASCII NUL terminated string naming the environment which is to be the initial current environment when the script is started. If this parameter is set to NULL, the filetype is used as the initial environment name. What the filetype is, may depend on your operating system, but in general it is everything after the last "." in the filename.

CallType A value describing whether the Rexx interpreter is to be invoked in command, function or subroutine mode. Actually, this has little significance. The main difference is that in command mode, only one parameterstring can be passed, and in function mode, a value must be returned. In addition, the mode chosen will affect the output of the PARSE SOURCE instruction in Rexx.

Three symbolic values of integral type are defined, which can be used for this parameter: RXCOMMAND, RXFUNCTION and RXSUBROUTINE.

SysExists A pointer to an array of exit handlers to be used. If no exit handlers are to be defined, NULL may be specified. Each element in the array defines one exit handler, and the element immediately following the last definition must have a sysexit_code set to RXENDLST.

ReturnCode Pointer to an long where the returncode is stored, provided that the returned value is numeric, and within the range -(2**15) to 2**15-1. I don't know what happens to ReturnCode if either of these conditions is not satisfied. It probably becomes undefined, which means that it is

totally useless since the program has to inspect the return string in order to determine whether ReturnCode is valid. Nor do I know why only 16 bits are used, when a long is 32 bits.

Result Points to a Rexx string into which the result string is written. The caller may or may not let the strptr field be supplied. If supplied (i.e. it is non-NULL), that area will be used, else a new area will be allocated. If the supplied area is used, its size is supposed to be given by the strlength field. If the size if not sufficient, a new area will be allocated, by some systemdependent channel (i.e. malloc()), and the caller must see to that it is properly deallocated (using free().

Note that the ArgCount parameter need not be the same as the ARG() built-in function would return. Differences will occurr if the last entries in ArgList are "null strings".

The Instore parameter needs some special attention. It is used to directly or indirectly specify where to fetch the code to execute. The following "algorithm" is used to determine what to execute:

* If Instore is NULL, then ProgName names the filename of an on-disk Rexx script which it to be read and executed.

* Else, if Instore is not NULL, the script is somewhere in memory, and no reading from disk is performed. If both Instore[0].strptr and Instore[1].strptr are NULL, then the script to execute is a preloaded macro which must have been loaded with a call to either RexxAddMacro() or RexxLoadMacroSpace(); and ProgName is the name of the macro to execute.

* Else, if Instore[1].strptr is non-NULL, then Instore[1] contains the pretokenized image of a Rexx script, and it is used for the execution.

* Else, if Instore[0].strptr is non-NULL, then Instore[0] contains the ASCII image of a Rexx script, just as if the script had been read directly from the disk (i.e. including linefeeds and such). This image is passed to the interpreter, which tokenizes it, and stores the tokenized script in the Instore[1] string, and then proceeds to execute that script. Upon return, the Instore[1] will be set, and can later be used to reexecute the script without the overhead of tokenizing.

The user is responsible for deallocating any storage used by Instore[1]. Note that after tokenizing, the sourcecode in Instore[0] is strictly speaking not needed anymore. It will only be consulted if the user calls the SOURCELINE() built-in function. It is not an error to use SOURCELINE() if the source is not present, but nullstrings and zero will be returned.

The valid return values from RexxStart() are:

Negative indicates that a syntax error occurred during interpretation. In general, you can expect the error value to have the same absolute value as the Rexx syntax error (but opposite signs, of course).

Zero indicates that the interpreter finished executing the script without errors.

Positive indicates probably that some problem occurred, that made it impossible to execute the script, e.g. a bad parameter value. However, I can't

find any references in the documentation which states which values it is supposed to return.

During the course of an execution of RexxStart(), subcommand handlers and exit handlers might be called. These may call any function in the application interface, including another invocation of RexxStart().

Often, the application programmer is interested in providing support simplifing the specification of filenames, like an environment variable search path or a default file type. The Rexx interface does support a default file type: .CMD, but the user may not set this to anything else. Therefore, it is generally up to the application programmer to handle search paths, and also default file types (unless .CMD is ok).

If the initial environment name (EvnName) is NULL, then the initial environment during interpretation will be set equal to the file type of the script to execute. If the script does not have a file type, it is probably set to some interpreter specific value.

1.78 Variable Pool Interface

This section describes the variable pool part of the application interface, which allows the application programmer to set, retrieve and drop variables in the Rexx interpreter from the application program. It also allows access to other information.

The C preprocessor symbol INCL_RXSHV must be defined if the definitions for the variable pool interface are to be made available when rexxsaa.h is included.

1.79 Symbolic or Direct

First, let us define two terms, "symbolic" variable name and "direct" variable name, which are used in connection with the variable pool.

A symbolic variable name is the name of a variable, but it needs normalization and tail substitution before it names the real variable. The name foo.bar is a symbolic variable name, and it is transformed by normalization, to FOO.BAR, and then by tail substitution to FOO.42 (assuming that the current value of BAR is 42).

Normalization is the process of uppercasing all characters in the symbolic name; and tail substitution is the process of substituting each distinct simple symbol in the tail for its value.

On the other hand, a direct variable refers directly to the name of the variable. In a sense, it is a symbolic variable that has already been normalized and tail substituted. For instance, foo.bar is not a valid direct variable name, since lower case letters are not allowed in the variable stem. The direct variable FOO.42 is the same as the variable above. For simple variables, the only difference between direct and symbolic variable names is that lower case letters are allowed in symbolic names

Note that the two direct variable names FOO.bar and FOO.BAR refer to different variables, since upper and lower case letters differ in the tail. In fact, the tail of a compound direct variable may contain any character, including ASCII NUL. The stem part of a variable, and all simple variables can not contain any lower case letters.

As a remark, what would the direct variable FOO. refer to: the stem FOO. or the compound variable having stem FOO. and a nullstring as tail? Well, I suppose the former, since it is the more useful. Thus, the latter is inaccessable as a direct variable.

1.80 The SHVBLOCK structure

All requests to manipulate the Rexx variable pool are controlled by a structure called SHVBLOCK, having the definition:

```
typedef struct shvnode {
   struct shvnode *shvnext ;     /* ptr to next in blk in chain */
   RXSTRING shvname ;            /* name of variable */
   RXSTRING shvvalue ;           /* value of variable */
   unsigned long shvnamelen ;    /* length of shvname.strptr */
   unsigned long shvvaluelen ;   /* length of shvvalue.strptr */
   unsigned char shvcode ;       /* operation code */
   unsigned char shvret ;        /* return code */
} SHVBLOCK ;

typedef SHVBLOCK *PSHVBLOCK ;
```

The fields shvnext and shvcode are purely input, while shvret is purely output. The rest of the fields might be input or output, depending on the requested operation, and the value of the fields. The significance of each field is:

shvnext One call to RexxVariablePool() may sequentially process many requests. The shvnext field links one request to the next in line. The last request must have set shvnext to NULL. The requests are handled individually and there is no difference between calling RexxVariablePool() with several requests, and making one call to RexxVariablePool() for each request.

shvname Contains the name of the variable to operate on, as a RXSTRING. This

field is only relevant for some requests, and its use may differ.

shvvalue Contains the value of the variable to operate on as a RXSTRING. Like shvname, this might not be relevant for all types of requests.

shvnamelen The length of the array that shvname.strptr points to. This field holds the maximum number of characters that shvname.strptr can hold (i.e. allocated). While shvname.strlength holds the number of characters that are actually in use (i.e. defined).

shvvaluelen The length of the array that shvvalue.strptr points to. Relates to shvvalue like shvnamelen relates to shvname.

shvcode The code of operation; decides what type of request to perform. A list of all the available requests is given below.

shvret A return code describing the outcome of the request. This code is a bit special. The lower seven bits are flags which are set depending on whether some condition is met or not. Values above 127 are not used in this field.

There is a difference between strnamelen and strname.strlength. The former is the total length of the array of characters pointed to by strname.strptr (if set). While the latter is the number of these characters that are actually in use. When a SHVBLOCK is used to return data from RexxVariablePool(), and a preallocated string space has been supplied, both these will be used; strname.strptr will be set to the length of the data returned, while strnamelen is never changed, only read to find the maximum number of characters that shvname can hold.

Even though shvnamelen is not really needed when shvname is used for input, it is wise to set it to its proper value (or at least set it to the same as shvname.strlength). The same applies for shvvalue and shvvaluelen.

The field shvcode can take one of the following symbolic values:

RXSHV_DROP The variable named by the direct variable name shvname is dropped (i.e. becomes undefined). The fields shvvalue and shvvaluelen do not matter.

RXSHV_EXIT This is used to set the return value for an external function or exit handler.

RXSHV_FETCH The value of the variable named by the direct variable name shvname is retrieved and stored in shvvalue. If shvvalue.strptr is NULL, the interpreter will allocate sufficient space to store the value (but it is the responsibility of the application programmer to release that space). Else, the value will be stored in the area allocated for shvvalue, and shvvaluelen is taken to be the maximum size of that area.

RXSHV_NEXTV This code is used to retrieve the names and values of all variables at the current procedure level; i.e. excluding variables shadowed by PROCEDURE. The name and value of each variable are retrieved simultaneously into shvname and shvvalue, respectively.

Successive requests for RXSHV_NEXTV will traverse the interpreter's internal data structure for storing variables, and return a new pair of

variable name and value for each request. Each variable that is visible in the current scope, is returned once and only once, but the order is indeterministic.

When all available variables in the Rexx interpreter have already been retrieved, subsequent RXSHV_NEXTV requests will set the flag RXSHV_LVAR in the shvret field. There are a few restrictions. The traversal will be reset whenever the interpreter resumes execution, so an incomplete traversal can not be continued in a later external function, exit handler, or subcommand handler. Also, any set, fetch or drop operation will reset the traversal. These restrictions have been added to ensure that the variable pool is static throughout one traversal.

RXSHV_PRIV Retrieves some piece of information from the interpreter, other than a variable value, based on the value of the shvname field. The value is stored in shvvalue as for a "normal" fetch. A list of possible names is shown below.

RXSHV_SET The variable named by the direct variable name shvname is set to the value given by shvvalue.

RXSHV_SYFET Like RXSHV_FETCH, except that shvname is a symbolic variable name.

RXSHV_SYDRO Like RXSHV_DROP, except that shvname is a symbolic variable name.

RXSHV_SYSET Like RXSHV_SET, except that shvname is a symbolic variable name.

One type of request that needs some special attention is the RXSHV_PRIV, which retrieves a kind of "meta-variable". Depending on the value of shvname, it returns a value in shvvalue describing some aspect of the interpreter. For RXSHV_PRIV the possible values for shvname are:

PARM Returns the ASCII representation of the number of parameters to the currently active Rexx procedure. This must not the same value as the built-in function ARG() returns, but the number ArgCount in RexxStart. The two might differ if a routine was called with trailing omitted parameters.

PARM.n The n must be a positive integer; and the value returned will be the n'th parameter at the current procedure level. This is not completely equivalent to the information that the builtin function ARG() returns. For parameters where ARG() would return the state omitted, the returned value is a "null string", while for parameters where ARG() would return the state "existing", the return value will be the parameter string (which may be a "zero length string".

QUENAME The name of the currently active external data queue. This feature has not yet been implemented in Regina, which always return "default".

SOURCE Returns the same string that is used in the PARSE SOURCE clause in Rexx, at the current procedure level of interpretation.

VERSION Returns the same string that is used in the PARSE VERSION clause in Rexx.

The value returned by a variable pool request is a bit uncommon. A return

value is computed for each request, and stored in the shvret field. This is a one-byte field, of which the most significant bit is never set. A symbolic value RXSHV_OK is defined as the value zero, and the shvret field will be equal to this name if none if the flags listed below is set. The symbolic value for these flags are:

RXSHV_BADF The shvcode of this request contained a bad function code.

RXSHV_BADN The shvname field contained a string that is not valid in this context. What exactly is a valid value depends on whether the operation is a private, a symbolic variable, or direct variable operation.

RXSHV_LVAR Set if and only if the request was RXSHV_NETXV, and all available variables have already been retrieved by earlier requests.

RXSHV_MEMFL There was not enough memory to complete this request.

RXSHV_NEWV Set if and only if the referenced variable did not previously have a value. It can be returned for any set, fetch or drop operation.

RXSHV_TRUNC Set if the retrieved value was truncated when it was copied into either the shvname or shvvalue fields. See below.

These flags are directrly suitable for logical OR, without shifting, e.g. to check for truncation and no variables left, you can do something like:

```
   if (req->shvret & (RXSHV_TRUNC _ RXSHV_LVAR))
      printf("Truncation or no vars left\n") ;
```

RXSHV_TRUNC can only occur when the interface is storing a retrieved value in a SHVBLOCK, and the preallocated space is present, but not sufficiently large. As described for RXSHV_FETCH, the interpreter will allocate enough space if shvvalue.strptr is NULL, and then RXSHV_TRUNC will never be set. Else the space supplied by shvvalue.strptr is used, and shvvaluelen is taken as the maximum length of shvvalue, and truncation will occur if the supplied space is too small.

Some implementations will consider SHV_MEMFL to be so severe as to skip the rest of the operations in a chain of requests. In order to write compatible software, you should never assume that requests following in a chain after a request that returned SHV_MEMFL have been performed.

The RXSHV_BADN is returned if the supplied shvname contains a value that is not legal in this context. For the symbolic set, fetch and drop operations, that means a symbol that is a legal variable name; both upper and lower case letters are allowed. For the direct set, fetch and drop operations, that means a variable name after normalization and tail substitution is not a legal variable name. For the RXSHV_PRIV, it must be one of the values listed above.

There is a small subtlety in the above description. TRL states that when a Rexx assignment assignes a value to a stem variable, all possible variables having that stem are assigned a new value (independent of whether they had an explicit value before). So strictly speaking, if a stem is set, then a RXSHV_NETV sequence should return an (almost) infinite sequence of compound variables for that stem. Of course, that is completely useless, so you can assume that only compound variables of that stem given an explicit value

after the stem was assigned a value will be returned by RXSHV_NEXTV. However, because of that subtlety, the variables returned by RXSHV_NEXTV for compound variables might not be representative for the state of the variables.

E.g. what would a sequence of RXSHV_NEXT requests return after the following Rexx code:

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

The second statement here, might not change the returned values! After the first statement, only the stem foo. would probably have been returned, and so also if all variables were fetched after the second statement.

1.81 Regina Notes for the Variable Pool

Due to the subtleties described at the end of the previous subsection, some notes on how Regina handles RXSHV_NEXTV requests for compound variables are in order. The following rules applies:

* Both the stem variable FOO. and the compound variable having FOO. as stem and a nullstring as tail, are returned with the name of FOO.. In this situation, a sequence of RXSHV_NEXTV requests may seem to return values for the same variable twice. This is unfortunate, but it seems to be the only way. In any case, you'll have to perform the RXSHV_SYFET in order to determine which is which.

* If a stem variable has not been assigned a value, its compound variables are only returned if they have been assigned an explicit value. I.e. compound variables for that stem that have either never been assigned a value, or have been dropped, will not be reported by RXSHV_NEXTV. There is nothing strange about this.

* If a stem variable has been assigned a value, then its compound variables will be reported in two cases: Firstly, the compound variables having explicitly been assigned a value afterwards. Secondly, the compound variables which have been dropped afterwards, which are reported to have their initial value, and the flag RXSHV_NEWV is set in shvret.

It may sound a bit stupid that unset variables are listed when the request is to list all variables which have been set, but that is about the best I can do, if I am to stay within the standard definition and return a complete and exact status of the variable pool.

If the return code from RexxVariablePool() is less than 128, Regina is guaranteed to have tried to process all requests in the chain. If the return code is above 127, some requests may not have been processed. Actually, the number 127 (or 128) is a bit inconvenient, since it will be an problem for later expansion of the standard. A much better approach would be to have a preprocessor symbol (say, RXSHV_FATAL, and if the return code from the RexxVariablePool() function was larger than that, it would be a "direct" error code, and not a "composite" error code built from the shvret fields of the requests. The RXSHV_FATAL would then have to be the addition of all the atomic composite error codes.

(Warning: author is mounting the soapbox.) The "right" way to fix it, would be to let RexxVariablePool() set another flag in shvret, a flag called something like RXSHV_STEM, which is set during a RXSHV_NEXTV if and only if the value returned is a stem variable. That way, the application programmer would be able to differ between stem variables and compound variable with a null string tail.

To handle the other problem with compound variables and RXSHV_NEXTV, I would have liked to return a "null string" in shvvalue if and only if the variable is a compound variable having its initial value, and the stem of that compound variable has been assigned a value. Then, the value of the compound variable is equal to its name, and is available in the shvname field.

I'd also like to see that the shvret value contained other information about the variables, e.g. whether the variable was exposed at the current procedure level. Of course, Regina does not contain any of these extra, unstandard features. (Author is dismounting the soapbox.)

When Regina is returning variables with RXSHV_NEXTV, the variables are returned in the order in which the occurr in the open hashtable in the interpreter. I.e. the order in which variables belonging to different bins are returned is consistent, but the order in which variables hashed to the same bin are returned, is indeterministic. Note that all compound variables belonging to the same stem are returned in one sequence.

1.82 The RexxVariablePool() function

This function is used to process a sequence of variable requests, and process them sequentially. The prototype of this function is:

```
ULONG RexxVariablePool(
   SHVBLOCK *Request
) ;
```

Its only parameter is a pointer to a SHVBLOCK structure, which may be the first of the linked list. The function performs the operation specified in each block. If an error should occur, the current request is terminated, and the function moves on to the next request in the chain.

The result value is a bit peculiar. If the returned value is less than 128, it is calculated by logically OR'ing the returned shvret field of all the requests in the chain. That way, you can easily check whether any of the requests was e.g. skipped because of lack of memory. To determine which request, you have to interate through the list.

If the result value is higher than 127, it signifies an error. If any of these values are set, you can not assume that any of the requests have been processed. The following symbolic name gives its meaning.

RXSHV_NOAVL Means that the interface is not available for this request. This might occur if the interface was not able to start the interpreter, or if an operation requested a variable when the interpreter is not currently

executing any script (i.e. idle and waiting for a script to execute).

1.83 The System Exit Handler Interface

The exit handlers provide a mechanism for governing important aspects of the Rexx interpreter from the application: It can trap situations like the interpreter writing out text, and then handle them itself, e.g. by displaying the text in a special window. You can regard system exits as a sort of "hooks".

The System Exit Handler

List of System Exit Handlers

RXFUN --- The External Function Exit Handler

RXCMD --- the Subcommand Exit Handler

RXMSQ --- the External Data Queue Exit Handler

RXSIO --- the Standard I/O Exit Handler

RXHLT --- the Halt Condition Exit Handler

RXTRC --- the Trace Status Exit Handler

RXINI --- the Initialization Exit Handler

RXTER --- the Termination Exit Handler

1.84 The System Exit Handler

Just like the subcommand handler, the system exit handler is a routine supplied by the application, and is called by the interpreter when certain situations occur. These situations are described in detail later. For the examples below, we will use the output from SAY as an example.

If a system exit handler is enabled for the SAY instruction, it will be called with a parameter describing the text that is to be written out. The system exit handler can choose to handle the situation (e.g. by writing the text itself), or it can ignore it and let the interpreter perform the output. The return code from the system exit tells the interpreter whether a system exit handled the situation or not.

A system exit handler must be a routine defined according to the prototype:

```
signed long my_exit_handler(
   signed long ExitNumber,
```

```
      signed long Subfunction,
      PEXIT ParmBlock
   ) ;
```

In this prototype, the type PEXIT is a pointer to a parameter block containing all the parameters necessary to handle the situation. The actual definition of this parameter block will vary, and is described in detail in the list of each system exit.

The exits are defined in a two-level hierarchy. The ExitNumber defines the main function for a system exit, while the Subfunction defines the subfunction within that main function. E.g. for SAY, the mainfunction will be RXSIO (the system exit for standard I/O) and the subfunction will be RXSIOSAY. The RXSIO main function has other subfunctions for handling trace output, interactive trace input, and PULL input from standard input.

The value returned from the system exit handler must be one of the following symbolic values:

RXEXIT_HANDLED Signals that the system exit handler took care of the situation, and that the interpreter should not proceed to do the default action. For the SAY instruction, this means that the interpreter will not print out anything.

RXEXIT_NOT_HANDLED Signals that the system exit handler did not take care of the situation, and the interpreter will proceed to perform the default action. For the SAY instruction, this means that it must print out the argument to SAY.

RXEXIT_RAISE_ERROR Signals that the interpreter's default action for this situation should not be performed, but instead a SYNTAX condition should be raised. Don't get confused by the name, it is not the ERROR condition, but the SYNTAX condition is raised,using the syntax error "Failure in system service" (normally numbered 48).

In addition to returning information as the numeric return value, information may also be returned by setting variables in the parameter block. For instance, if the system exit is to handle interactive trace input, that is how it will supply the interpreter with the input string.

It is a good and disiplined practice to let your exit handlers start by verifying the ExitNumber and Subfunction codes, and immediately return RXEXIT_NOT_HANDLED if it does not recognize both of them. That way, your application will be upwards compatible with future interpreters which might have more subfunctions for any given main function.

1.85 List of System Exit Handlers

1.86 RXFUN --- The External Function Exit Handler

1.87 RXCMD --- the Subcommand Exit Handler

The main function code for this exit handler is given by the symbolic name RXCMD. It is called whenever the interpreter is about to call a subcommand, i.e. a command to an external environment. It has only one subfunction: RXCMDHST.

The ParmBlock parameter for this subfunction has the following definition:

```
typedef struct {
   typedef struct {
      unsigned int rxfcfail:1 ;
      unsigned int rxfcerr:1 ;
   } rxcmd_flags ;
   unsigned char *rxcmd_address ;
   unsigned short rxcmd_addressl ;
   unsigned char *rxcmd_dll ;
   unsigned short rxcmd_dll_len ;
   RXSTRING rxcmd_command ;
   RXSTRING rxcmd_retc ;
} RXCMDHST_PARM ;
```

The significance of each variable is:

rxcmd_flags.rxfcfail If this flag is set, the interpreter will raise a FAILURE condition at the return of the exit handler.

rxcmd_flags.rxfcerr Like the previous, but the ERROR condition is raised instead.

rxcmd_address Points to a character array containing the name of the environment to which the command normally would be sent.

rxcmd_addressl Holds the length of rxcmd_address. Note that the last character is the letter "ell", not the number one.

rxcmd_dll Defines the name for the DLL which is to handle the command. I'm not sure what this entry is used for. It is not currently in use for Regina.

rxcmd_dll_len Holds the length of rxcmd_dll. If this length is set to zero, the subcommand handler for this environment is not a DLL, but an exe handler.

rxcmd_command Holds the command string to be executed, including command name and parameters.

rxcmd_retc Set by the exit handler to the string which is to be considered the return code from the command. It is assigned to the special variable RC at return from the exit handler. The user is responsible for allocating space for this variable. I have no clear idea what happens if rxcmd_retc.strptr is set to NULL; it might set RC to zero, to the null string, or even drop it.

It seems that this exit handler is capable of raising both the ERROR and the FAILURE conditions simultaneously. I don't know whether that is legal, or whether only the FAILURE condition is raised, since the ERROR condition is a sort of "subset" of FAILURE.

Note that the return fields of the parameter block are only relevant if
the value RXEXIT_HANDLED was returned. This applies to the rxcmd_flags and
rxcmd_retc fields of the structure.

1.88 RXMSQ --- the External Data Queue Exit Handler

1.89 RXSIO --- the Standard I/O Exit Handler

The main code for this exit handler has the symbolic value RXSIO. There are
four subfunctions:

RXSIODTR Called whenever the interpreter needs to read a line from the user
    during interactive tracing. Note the difference between this subfunction
    and RXSIOTRD.

RXSIOSAY Called whenever the interpreter tries to write something to standard
    output in a SAY instruction, even a SAY instruction without a parameter.

RXSIOTRC Called whenever the interpreter tries to write out debugging
    information, e.g. during tracing, as a trace back, or as a syntax error
    message.

RXSIOTRD Called whenever the interpreter need to read from the standard input
    stream during a PULL or PARSE PULL instruction. Note that it will not be
    called if there is sufficient data on the stack to satisfy the operation.

Note that these function are only called for the exact situations that are
listed above. E.g. the RXSIOSAY is not called during a call to the Rexx
built-in function LINEOUT() that writes to the default output stream. TRL
says that SAY is identical to calling LINEOUT() for the standard output
stream, but SAA API still manages to see the difference between stem
variables and compound variables with a "zero-length-string" tail. Please
bear with this inconsistency.

Depending on the subfunction, the ParmBlock parameter will have four only
slightly different definitions. It is kind of frustrating that the ParmBlock
takes so many different datatypes, but it can be handled easily using unions,
see a later section. The definitions are:

```
typedef struct {
   RXSTRING rxsiodtr_retc ;  /* the interactive trace input */
} RXSIODTR_PARM ;

typedef struct {
   RXSTRING rxsio_string ;   /* the SAY line to write out */
} RXSIOSAY_PARM ;

typedef struct {
   RXSTRING rxsio_string ;   /* the debug line to write out */
} RXSIOTRC_PARM ;

typedef struct {
```

```
      RXSTRING rxsiotrd_retc ;  /* the line to read in */
   } RXSIOTRD_PARM ;
```

In all of these, the RXSTRING structure either holds the value to be written out (for RXSIOSAY and RXSIOTRC), or the value to be used instead of reading standard input stream (for RXSIOTRD and RXSIODTR). Note that the values set by RXSIOTRD and RXSIODTR are ignored if the exit handler does not return the value RXEXIT_HANDLED.

Any end-of-line marker are stripped off the strings in this context. If the exit handler writes out the string during RXSIOSAY or RXSIOTRC, it must supply any end-of-line action itself. Similarily, the interpreter does not expect a end-of-line marker in the data returned from RXSIODTR and RXSIOTRD.

The space used to store the return data for the RXSIODTR and RCSIOTRD subfunctions, must be provided by the exit handler itself, and the space is not deallocated by the interpreter. The space can be reused by the application at any later time. The space allocated to hold the data given by the RXSIOSAY and RXSIOTRC subfunctions, will be allocated by the interpreter, and must neither be deallocated by the exit handler, nor used after the exit handler has terminated.

1.90 RXHLT --- the Halt Condition Exit Handler

1.91 RXTRC --- the Trace Status Exit Handler

1.92 RXINI --- the Initialization Exit Handler

RXTER and this exit handler are a bit different from the others. RXINI provides the application programmer with a method of getting control before the execution of the script starts. Its main purpose is to enable manipulation of the variable pool in order to set up certain variables before the script starts, or set the trace mode.

It has only one subfunction, RXINIEXT, called once duing each call to RexxStart(): just before the first Rexx statement is interpreted. Variable manipulations performed during this exit will have effect when the script starts.

As there is no information to be communicated during this exit, the value of ParmBlock is undefined. It makes no difference whether you return RXEXIT_HANDLED or RXEXIT_NOT_HANDLED, since there is no situation to handle.

1.93 RXTER --- the Termination Exit Handler

This exit resembles RXINI. Its sole subfunction is RXTEREXT, which is called once, just after the last statement of the Rexx script has been interpreted. The state of all variables are intact during this call; so it can be used to retrieve the values of the variables at the exit of a script. (In fact, that

```
is the whole purpose of this exit handler.)

Like RXINI, there is no information to be communicated during the exit,
so ParamBlock is undefined in this call. And also like RXINI, it is more
of a hook than an exit handler, so it does not matter whether you return
RXEXIT_HANDLED or RXEXIT_NOT_HANDLED.
```