Extended Standard Programming Language Rexx

Rexx Language ARB

1 May 2023

THE REXX LANGUAGE ASSOCIATION
RexxLA Symposium Proceedings Series
ISSN 1534-8954

Publication Data

A publication of **RexxLA Press**

The responsible publisher of this edition is identified as *IBizz IT Services and Consultancy*, Amsteldijk 14, 1074 HR Amsterdam, a registered company governed by the laws of the Kingdom of The Netherlands.

The RexxLA Symposium Series is registered under ISSN 1534-8954
The 2023 edition is registered under ISBN 978-90-819090-1-3

ISBN 978-90-819090-1-3

9 789081 909013 >

2023-03-31 First printing

Contents

1

Foreword

1.1 Purpose

This standard provides an unambiguous definition of the programming language Rexx. Its purpose is to facilitate portability of Rexx programs for use on a wide variety of computer systems. History The computer programming language Rexx was designed by Mike Cowlishaw to satisfy the following principal aims:

- to provide a highly readable command programming language for the benefit of programmers and program readers, users and maintainers;
- to incorporate within this language program design features such as natural data typing and control structures which would contribute to rapid, efficient and accurate program development;
- to define a language whose implementations could be both reliable and efficient on a wide variety of computing platforms.

In November, 1990, X3 announced the formation of a new technical committee, X3J18, to develop an American National Standard for Rexx. This standard was published as ANSI X3.274-1996.

The popularity of “Object Oriented” programming, and the need for Rexx to work with objects created in various ways, led to Rexx extensions and to a second X3J18 project which produced this standard. (Ed - hopefully)

1.2 Committee lists

(Here)

This standard was prepared by the Technical Development Committee for Rexx, X3J18. There are annexes in this standard; they are informative and are not considered part of this standard.

Suggestions for improvement of this standard will be welcome. They should be sent to the

Information Technology Industry Council, 1250 Eye Street, NW, Washington DC 20005-3922.

This standard was processed and approved for submittal to ANSI by the Accredited Standards Committee on Information Processing Systems, NCITS. Committee approval of this standard does not necessarily imply that all committee members voted for its approval. At the time it approved this standard, the NCITS Committee had the following members:

To be inserted The people who contributed to Technical Committee J18 on Rexx, which developed this standard, include:

2 Introduction

This standard provides an unambiguous definition of the programming language Rexx.

3

Scope, purpose, and application

3.1 Scope

This standard specifies the semantics and syntax of the programming language Rexx by specifying requirements for a conforming language processor. The scope of this standard includes

- the syntax and constraints of the Rexx language;
- the semantic rules for interpreting Rexx programs;
- the restrictions and limitations that a conforming language processor may impose;
- the semantics of configuration interfaces.

This standard does not specify

- the mechanism by which Rexx programs are transformed for use by a data-processing system;
- the mechanism by which Rexx programs are invoked for use by a data-processing system;
- the mechanism by which input data are transformed for use by a Rexx program;
- the mechanism by which output data are transformed after being produced by a Rexx program;
- the encoding of Rexx programs;
- the encoding of data to be processed by Rexx programs;
- the encoding of output produced by Rexx programs;
- the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular language processor;
- all minimal requirements of a data-processing system that is capable of supporting a conforming language processor;
- the syntax of the configuration interfaces.

3.2 Purpose

The purpose of this standard is to facilitate portability of Rexx programs for use on a wide variety of configurations.

3.3 Application

This standard is applicable to Rexx language processors.

3.4 Recommendation

It is recommended that before detailed reading of this standard, a reader should first be familiar with the Rexx language, for example through reading one of the books about Rexx. It is also recommended that the annexes should be read in conjunction with this standard.

4

Normative references

There are no standards which constitute provisions of this American National Standard.

5

Definitions and document notation

Lots more for NetRexx

5.1 Definitions

5.1.1 application programming interface:

A set of functions which allow access to some Rexx facilities from non-Rexx programs.

5.1.2 arguments:

The expressions (separated by commas) between the parentheses of a function call or following the name on a CALL instruction. Also the corresponding values which may be accessed by a function or routine, however invoked.

5.1.3 built-in function:

A function (which may be called as a subroutine) that is defined in section nnn of this standard and can be used directly from a program.

5.1.4 character string:

A sequence of zero or more characters.

5.1.5 clause:

A section of the program, ended by a semicolon. The semicolon may be implied by the end of a line or by some other constructs.

5.1.6 coded:

A coded string is a string which is not necessarily comprised of characters. Coded strings can occur as arguments to a program, results of external routines and commands, and the results of some built-in functions, such as D2C.

5.1.7 command:

A clause consisting of just an expression is an instruction known as a command. The expression is evaluated and the result is passed as a command string to some external environment. ### condition:

A specific event, or state, which can be trapped by CALL ON or SIGNAL ON.

5.1.8 configuration:

Any data-processing system, operating system and software used to operate a language processor.

5.1.9 conforming language processor:

A language processor which obeys all the provisions of this standard.

5.1.10 construct:

A named syntax grouping, for example “expression”, “do_ specification”.

5.1.11 default error stream:

An output stream, determined by the configuration, on which error messages are written.

5.1.12 default input stream:

An input stream having a name which is the null string. The use of this stream may be implied.

5.1.13 default output stream:

An output stream having a name which is the null string. The use of this stream may be implied.

5.1.14 direct symbol:

A symbol which, without any modification, names a variable in a variable pool.

5.1.15 directive:

Clauses which begin with two colons are directives. Directives are not executable, they indicate the structure of the program. Directives may also be written with the two colons implied.

5.1.16 dropped:

A symbol which is in an unitialized state, as opposed to having had a value assigned to it, is described as dropped. The names in a variable pool have an attribute of ‘dropped’ or ‘not-dropped’.

5.1.17 encoding:

The relation between a character string anda corresponding number. The encoding of character strings is determined by the configuration.

5.1.18 end-of-line:

An event that occurs during the scanning of a source program. Normally the end-of-lines will relate to the lines shown if the configuration lists the program. They may, or may not, correspond to characters in the source program.

5.1.19 environment:

The context in which a command may be executed. This is comprised of the environment name, details of the resource that will provide input to the command, and details of the resources that will receive output of the command.

5.1.20 environment name:

The name of an external procedure or process that can execute commands. Commands are sent to the current named environment, initially selected externally but then alterable by using the ADDRESS instruction.

5.1.21 error number:

A number which identifies a particular situation which has occurred during processing. The message prose associated with such a number is defined by this standard.

5.1.22 exposed:

Normally, a symbol refers to a variable in the most recently established variable pool. When this is not the case the variable is referred to as an exposed variable.

5.1.23 expression:

The most general of the constructs which can be evaluated to produce a single string value.

5.1.24 external data queue:

A queue of strings that is external to REXX programs in that other programs may have access

to the queue whenever REXX relinquishes control to some other program.

5.1.25 external routine:

A function or subroutine that is neither built-in nor in the same program as the CALL instruction or function call that invokes it.

5.1.26 external variable pool:

A named variable pool supplied by the configuration which can be accessed by the VALUE built-in function.

5.1.27 function:

Some processing which can be invoked by name and will produce a result. This term is used for both Rexx functions (See nnn) and functions provided by the configuration (see n).

5.1.28 identifier:

The name of a construct.

5.1.29 implicit variable:

A tailed variable which is in a variable pool solely as a result of an operation on its stem. The names in a variable pool have an attribute of ‘implicit’ or ‘not-implicit’.

5.1.30 instruction:

One or more clauses that describe some course of action to be taken by the language processor.

5.1.31 internal routine:

A function or subroutine that is in the same program as the CALL instruction or function call that invokes it.

5.1.32 keyword:

This standard specifies special meaning for some tokens which consist of letters and have particular spellings, when used in particular contexts. Such tokens, in these contexts, are keywords.

5.1.33 label:

A clause that consists of a single symbol or a literal followed by a colon.

5.1.34 language processor:

Compiler, translator or interpreter working in combination with a configuration.

5.1.35 notation function:

A function with the sole purpose of providing a notation for describing semantics, within this standard. No Rexx program can invoke a notation function.

5.1.36 null clause:

A clause which has no tokens.

5.1.37 null string:

A character string with no characters, that is, a string of length zero.

5.1.38 production:

The definition of a construct, in Backus-Naur form.

5.1.39 return code:

A string that conveys some information about the command that has been executed. Return codes usually indicate the success or failure of the command but can also be used to represent other information.

5.1.40 routine:

Some processing which can be invoked by name.

5.1.41 state variable:

A component of the state of progress in processing a program, described in this standard by a named variable. No Rexx program can directly access a state variable.

5.1.42 stem:

If a symbol naming a variable contains a period which is not the first character, the part of the symbol up to and including the first period is the stem.

5.1.43 stream:

Named streams are used as the sources of input and the targets of output. The total semantics of such a stream are not defined in this standard and will depend on the configuration. A stream may be a permanent file in the configuration or may be something else, for example the input from a keyboard.

5.1.44 string:

For many operations the unit of data is a string. It may, or may not, be comprised of a sequence of characters which can be accessed individually.

5.1.45 subcode:

The decimal part of an error number.

5.1.46 subroutine:

An internal, built-in, or external routine that may or may not return a result string and is invoked by the CALL instruction. If it returns a result string the subroutine can also be invoked by a function call, in which case it is being called as a function. ### symbol:

A sequence of characters used as a name, see nnn. Symbols are used to name variables, functions, etc.

5.1.47 tailed name:

The names in a variable pool have an attribute of ‘tailed’ or ‘non-tailed’. Otherwise identical names are distinct if their attributes differ. Tailed names are normally the result of replacements in the tail of a symbol, the part that follows a stem.

5.1.48 token:

The unit of low-level syntax from which high-level constructs are built. Tokens are literal strings, symbols, operators, or special characters.

5.1.49 trace:

A description of some or all of the clauses of a program, produced as each is executed.

5.1.50 trap:

A function provided by the user which replaces or augments some normal function of the language processor.

5.1.51 variable pool:

A collection of the names of variables and their associated values.

5.2 Document notation

5.2.1 Rexx Code

Some Rexx code is used in this standard. This code shall be assumed to have its private set of variables. Variables used in this code are not directly accessible by the program to be processed. Comments in the code are not part of the provisions of this standard.

5.2.2 Italics

Throughout this standard, except in Rexx code, references to the constructs defined in section nnn are italicized.

6

Conformance

6.1 Conformance

A conforming language processor shall not implement any variation of this standard except where this standard permits. Such permitted variations shall be implemented in the manner prescribed by this standard and noted in the documentation accompanying the processor. A conforming processor shall include in its accompanying documentation

- alist of all definitions or values for the features in this standard which are specified to be dependent on the configuration.
- a statement of conformity, giving the complete reference of this standard (ANSI X3.274-1996) with which conformity is claimed.

6.2 Limits

Aside from the items listed here (and the assumed limitation in resources of the configuration), a conforming language processor shall not put numerical limits on the content of a program. Where a limit expresses the limit on a number of digits, it shall be a multiple of three. Other limits shall be one of the numbers one, five or twenty five, or any of these multiplied by some power of ten. Limitations that conforming language processors may impose are:

- NUMERIC DIGITS values shall be supported up to a value of at least nine hundred and ninety nine.
- Exponents shall be supported. The limit of the absolute value of an exponent shall be at least as large as the largest number that can be expressed without an exponent in nine digits.
- String lengths shall be supported. The limit on the length shall be at least as large as the largest number that can be expressed without an exponent in nine digits.
- String literal length shall be supported up to at least two hundred and fifty.
- Symbol length shall be supported up to at least two hundred and fifty.

7

Configuration

Any implementation of this standard will be functioning within a configuration. In practice, the boundary between what is implemented especially to support Rexx and what is provided by the system will vary from system to system. This clause describes what they shall together do to provide the configuration for the Rexx language processing which is described in this standard.

We don't want to add undue "magic" to this section. It seems we will need the concept of a "reference" (equivalent to a machine address) so that this section can at least have composite objects as arguments. (As it already does but these are not Rexx objects)

Possibly we could unify "reference" with "variable pool number" since object one-to-one with its variable pool is a fair model. That way we don't need a new primitive for comparison of two references.

JAVA is only a "reference" for NetRexx so some generalized JAVA-like support is needed for that. It would provide the answers to what classes were in the context, what their method signatures were etc.

7.1 Notation

The interface to the configuration is described in terms of functions. The notation for describing the interface functionally uses the name given to the function, followed by any arguments. This does not constrain how a specific implementation provides the function, nor does it imply that the order of arguments is significant for a specific implementation.

The names of the functions are used throughout this standard; the names used for the arguments are used only in this clause and nnn.

The name of a function refers to its usage. A function whose name starts with

- Config_ is used only from the language processor when processing programs;
- API_is part of the application programming interface and is accessible from programs which are not written in the Rexx language;
- Trap_ is not provided by the language processor but may be invoked by the language processor. As its result, each function shall return a completion Response. This is a string indicating how the function behaved. The completion response may be the character 'N' indicating the normal behavior occurred; otherwise the first character is an indicator of a different

behavior and the remainder shall be suitable as a human-readable description of the function's behavior.

This standard defines any additional results from Config_ functions as made available to the language processor in variables. This does not constrain how a particular implementation should return these results.

7.1.1 Notation for completion response and conditions

As alternatives to the normal indicator 'N', each function may return a completion response with indicator 'X' or 'S'; other possible indicators are described for each function explicitly. The indicator 'X' means that the function failed because resources were exhausted. The indicator 'S' shows that the configuration was unable to perform the function. Certain indicators cause conditions to be raised. The possible raising of these conditions is implicit in the use of the function; it is not shown explicitly when the functions are used in this standard. The implicit action is call #Raise 'SYNTAX', Message, Description where: #Raise raises the condition, see nnn. Message is determined by the indicator in the completion response. If the indicator is 'X' then Message is 5.1. If the indicator is 'S' then Message is 48.1. Description is the description in the completion response. The 'SYNTAX' condition 5.1 can also be raised by any other activity of the language processor.

7.2 Processing initiation

The processing initiation interface consists of a function which the configuration shall provide to invoke the language processor. We could do REQUIRES in a macro-expansion way by adding an argument to Contig_SourceChar to specify the source file. However, I'm assuming we will prefer to recursively "run" each required file. One of the results of that will be the classes and methods made public by that REQUIRES subject.

7.2.1 API Start

Syntax:

API Start(How, Source, Environment, Arguments, Streams, Traps, Provides)

where: How is one of 'COMMAND', 'FUNCTION', or 'SUBROUTINE' and indicates how the program is invoked.

What does OO! say for How when running REQUIREd files?

Source is an identification of the source of the program to be processed.

Environment is the initial value of the environment to be used in processing commands. This has components for the name of the environment and how the input and output of commands is to be directed.

Arguments is the initial argument list to be used in processing. This has components to specify the number of arguments, which arguments are omitted, and the values of arguments that are not omitted.

Streams has components for the default input stream to be used and the default output streams to be used.

Traps is the list of traps to be used in processing (see nnn). This has components to specify whether each trap is omitted or not.

Semantics:

This function starts the execution of a Rexx program.

If the program was terminated due to a RETURN or EXIT instruction without an expression the completion response is 'N'.

If the program was terminated due to a RETURN or EXIT instruction with an expression the indicator in the completion response is 'R' and the description of the completion response is the value of the expression.

If the program was terminated due to an error the indicator in the completion response is 'E' and the description in the completion response comprises information about the error that terminated processing.

If How was 'REQUIRED' and the completion response was not 'E', the Provides argument is set to reference classes made available. See nnn for the semantics of these classes.

7.3 Source programs and character sets

The configuration shall provide the ability to access source programs (see nnn). Source programs consist of characters belonging to the following categories:

- syntactic_characters;
- extra_letters;
- other_blank_characters;
- other_negators;
- other_characters.

A character shall belong to only one category.

7.3.1 Syntactic_characters

The following characters represent the category of characters called syntactic_-characters, identified by their names. The glyphs used to represent them in this document are also shown. Syntactic_characters shall be available in every configuration:

- & ampersand;
- apostrophe, single quotation mark, single quote;

- asterisk, star;
- blank, space;
- A-Z capital letters A through Z;
- colon;
- , comma;
- 0-9 digits zero through nine;
- = equal sign;
- exclamation point, exclamation mark;
- greater-than sign;

hyphen, minus sign;

< less-than sign;

- [left bracket, left square bracket; (_ left parenthesis;
- % percent sign;
- . period, decimal point, full stop, dot;
- plus sign;
- ? question mark;
- ” quotation mark, double quote; reverse slant, reverse solidus, backslash;] right bracket, right square bracket;
-) right parenthesis; ; semicolon; /_ slant, solidus, slash; a-z small letters a through z;
- ~ tilde, twiddle;
- _ underline, low line, underscore;
- vertical line, bar, vertical bar.

7.3.2 Extra_letters

A configuration may have a category of characters in source programs called extra_letters. Extra_letters are determined by the configuration.

7.3.3 Other_blank_characters

A configuration may have a category of characters in source programs called other_blank_characters. Other_blank_characters are determined by the configuration. Only the following characters represent possible characters of this category:

- carriage return;
- form feed;
- horizontal tabulation;
- new line;
- vertical tabulation.

7.3.4 Other_negators

A configuration may have a category of characters in source programs called other_negators. Other_negators are determined by the configuration. Only the following characters represent possible characters of this category. The glyphs used to represent them in this document are also shown:

- ▪ circumflex accent, caret;
- — not sign.

7.3.5 Other_characters

A configuration may have a category of characters in source programs called other_characters. Other_characters are determined by the configuration.

7.4 Configuration characters and encoding

The configuration characters and encoding interface consists of functions which the configuration shall provide which are concerned with the encoding of characters. The following functions shall be provided:

- Config_SourceChar;
- Config_OtherBlankCharacters;
- Config_Upper;
- Config_Compare;
- Config_B2C;
- Config_C2B;
- Config_Substr;
- Config_Length;
- Config_Xrange.

7.4.1 Config_SourceChar

Syntax:

Config SourceChar ()

Semantics: Supply the characters of the source program in sequence, together with the EOL and EOS events. The EOL event represents the end of a line. The EOS event represents the end of the source program. The EOS event must only occur immediately after an EOL event. Either a character or an event is supplied on each invocation, by setting #Outcome.

If this function is unable to supply a character because the source program encoding is incorrect the indicator of the completion response is ‘E’ and the description of the completion information is the encoding which is incorrect, in hexadecimal notation.

7.4.2 Config_OtherBlankCharacters

Syntax: Config OtherBlankCharacters ()

Semantics: Get other_blank_characters (see nnn). Set #Outcome to a string of zero or more distinct characters in arbitrary order. Each character is one that the configuration considers equivalent to the character Blank for the purposes of parsing.

7.4.3 Config_Upper

Syntax: Config Upper (Character)

where: Character is the character to be translated to uppercase. Semantics: Translate Character to uppercase. Set #Outcome to the translated character. Characters which have been subject to this translation are referred to as being in uppercase. Config_Upper applied to a character in uppercase must not change the character.

7.4.4 Config_Lower

Syntax:

Config Lower (Character) where: Character is the character to be translated to lowercase. Semantics: Translate Character to lowercase. Set #Outcome to the translated character. Characters which have been subject to this translation are referred to as being in lowercase. Config_Lower applied to a character in lowercase must not change the character. Config_Upper of the outcome of Config_Lower(Character) shall be the original character.

7.4.5 Config_Compare

Syntax: Config Compare(Characterl, Character2)

where:

Character1 is the character to be compared with Character2. Character2 is the character to be compared with Character1.

Semantics:

Compare two characters. Set #Outcome to

- ‘equal’ if Character1 is equal to Character2;
- ‘greater’ if Character1 is greater than Character2;
- ‘lesser’ if Character’ is less than Character2. The function shall exhibit the following characteristics. If Config _Compare(a,b) produces
- ‘equal’ then Config_Compare(b,a) produces ‘equal’;
- ‘greater’ then Config_Compare(b,a) produces ‘lesser’;
- ‘lesser’ then Config_Compare(b,a) produces ‘greater’;

- ‘equal’ and Config_Compare(b,c) produces ‘equal’ then Config_Compare(a,c) produces ‘equal’;
- ‘greater’ and Config_Compare(b,c) produces ‘greater’ then Config _Compare(a,c) produces ‘greater’;
- ‘lesser’ and Config _Compare(b,c) produces ‘lesser’ then Config_Compare(a,c) produces ‘lesser’;
- ‘equal’ then Config_Compare(a,c) and Config_Compare(b,c) produce the same value. Syntactic characters which are different characters shall not compare equal by Config_Compare, see nnn.

7.4.6 Config _B2C

Syntax: Config B2C (Binary) where: Binary is a sequence of digits, each ‘0’ or ‘1’. The number of digits shall be a multiple of eight. Semantics:

Translate Binary to a coded string. Set #Outcome to the resulting string. The string may, or may not, correspond to a sequence of characters.

7.4.7 Config_C2B

Syntax: Config C2B (String)

where: String is a string. Semantics: Translate String to a sequence of digits, each ‘0’ or ‘1’. Set #Outcome to the result. This function is the inverse of Config_-B2C.

7.4.8 Config_Substr

Syntax: Config Substr(String, n)

where: String is a string. nis an integer identifying a position within String. Semantics: Copy the n-th character from String. The leftmost character is the first character. Set Outcome to the resulting character. If this function is unable to supply a character because there is no n-th character in String the indicator of the completion response is ‘M’. If this function is unable to supply a character because the encoding of String is incorrect the indicator of the completion response is ‘E’ and the description of the completion information is the encoding which is incorrect, in hexadecimal notation.

7.4.9 Config Length

Syntax: Config Length (String)

where: String is a string. Semantics: Set #Outcome to the length of the string, that is, the number of characters in the string. If this function is unable to determine a length because the encoding of String is incorrect, the indicator of the completion response is ‘E’ and the description of the completion information is the encoding which is incorrect, in hexadecimal notation.

7.4.10 Config_Xrange

Syntax:

Config Xrange(Characterl, Character2) where: Character1 is the null string, or a single character. Character2 is the null string, or a single character. Semantics: If Character1 is the null string then let LowBound be a lowest ranked character in the character set according to the ranking order provided by Config_Compare; otherwise let LowBound be Character1. If Character2 is the null string then let HighBound be a highest ranked character in the character set according to the ranking order provided by Config_Compare; otherwise let HighBound be Character2lf #Outcome after Config_Compare(LowBound,HighBound) has a value of

- ‘equal’ then #Outcome is set to LowBound;
- ‘lesser’ then #Outcome is set to the sequence of characters between LowBound and HighBound inclusively, in ranking order;
- ‘greater’ then #Outcome is set to the sequence of characters HighBound and larger, in ranking order, followed by the sequence of characters LowBound and smaller, in ranking order.

7.5 Objects

The objects interface consists of functions which the configuration shall provide for creating objects.

7.5.1 Config_ObjectNew

Syntax: Config ObjectNew

Semantics:

Set #Outcome to be a reference to an object. The object shall be suitable for use as a variable pool, see nnn. This function shall never return a value in #Outcome which compares equal with the value returned on another invokation of the function.

7.5.2 Config_Array_Size

Syntax: Config Array Size(Object, size) where: Object is an object. Size is an integer greater or equal to 0. Semantics: The configuration should prepare to deal efficiently with the object as an array with indexes having values up to the value of size.

7.5.3 Config_Array_Put

Syntax: Config Array Put(Array, Item, Index)

where: Array is an array. Item is an object Index is an integer greater or equal to 1. Semantics: The configuration shall record that the array has Item associated with Index.

7.5.4 Config_Array_At

Syntax: Config Array At(Array, Index)

where: Array is an array. Index is an integer greater or equal to 1. Semantics: The configuration shall return the item that the array has associated with Index.

7.5.5 Config_Array_Hasindex

Syntax: Config Array At(Array, Index) where: Array is an array. Index is an integer greater or equal to 1. Semantics: Return ‘1’ if there is an item in Array associated with Index, ‘0’ otherwise.

7.5.6 Config_Array_Remove

Syntax: Config Array At(Array, Index)

where: Array is an array. Index is an integer greater or equal to 1. Semantics: After this operation, no item is associated with the Index in the Array.

7.6 Commands

The commands interface consists of a function which the configuration shall provide for strings to be passed as commands to an environment.

See nnn and nnn for a description of language features that use commands.

7.6.1 Config _Command

Syntax:

Config Command(Environment, Command) where: Environment is the environment to be addressed. It has components for:

- the name of the environment;
- the name of a stream from which the command will read its input. The null string indicates use of the default input stream;

- the name of a stream onto which the command will write its output. The null string indicates use of the default output stream. There is an indication of whether writing is to APPEND or REPLACE;
- the name of a stream onto which the command will write its error output. The null string indicates use of the default error output stream. There is an indication of whether writing is to APPEND or REPLACE. Command is the command to be executed. Semantics: Perform a command.
- set the indicator to ‘E’ or ‘F’ if the command ended with an ERROR condition, or a FAILURE condition, respectively;
- set #RC to the return code string of the command.

7.7 External routines

The external routines interface consists of a function which the configuration shall provide to invoke external routines. See nnn and nnn for a description of the language features that use external routines.

7.7.1 Config_ExternalRoutine

Syntax: Config ExternalRoutine(How, NameType, Name, Environment, Arguments, Streams, Traps) where: How is one of ‘FUNCTION’ or ‘SUBROUTINE’ and indicates how the external routine is to be invoked.

NameType is a specification of whether the name was provided as a symbol or as a string literal. Name is the name of the routine to be invoked.

Environment is an environment value with the same components as on API_-Start.

Arguments is a specification of the arguments to the routine, with the same components as on APL Start.

Streams is a specification of the default streams, with the same components as on API_ Start.

Traps is the list of traps to be used in processing, with the same components as on API_ Start.

Semantics:

Invoke an external routine. Set {Outcome to the result of the external routine, or set the indicator of the completion response to ‘D’ if the external routine did not provide a result.

If this function is unable to locate the routine the indicator of the completion response is ‘U’. As a result SYNTAX condition 43.1 is raised implicitly.

If How indicated that a result from the routine was required but the routine did not provide one the indicator of the completion response is ‘H’. As a result SYNTAX condition 44.1 is raised implicitly.

If How indicated that a result from the routine was required but the routine provided one that was too long (see #Limit_String in nnn) the indicator of the completion response is ‘L’. As a result SYNTAX condition 52 is raised implicitly.

If the routine failed in a way not indicated by some other indicator the indicator of the completion response is ‘F’. As a result SYNTAX condition 40.1 is raised implicitly.

7.7.2 Config_ExternalMethod

OO! has external classes explicitly via the ::CLASS abc EXTERNAL mechanism. Analogy with classic would also allow the subject of ::REQUIRES to be coded in non-Rexx. However ::REQUIRES subject is coded, we need to gather in knowledge of its method names because of the search algorithm that determines which method is called. Hence reasonable that the ultimate external call is to a method. Perhaps combine Config_ExternalRoutine with Config_ExternalMethod. There is a terminology clash on “environment”. Perhaps easiest to change the classic to “address_environment”. (And make it part of new “environment”?) There are terminology decisions to make about “files”, “programs”, and “packages”. Possibly “program” is the thing you run (and we don’t say what it means physically), “file” is a unit of scope (ROUTINEs in current file before those in REQUIREd), and “package” we don’t use (since a software package from a shop would probably have several files but not everything to run a program.) Using “file” this way may not be too bad since we used “stream” rather than “tile” in the classic definition. The How parameter will need ‘METHOD’ as a value. Should API_Start also allow ’METHOD”. If we pass the new Environment we don’t have to pass Streams separately.

Text of Config_ExternalMethod waiting on such decisions. Syntax:

Config ExternalMethod (How, NameType, Name, Environment, Arguments, Streams, Traps) where: How is one of ‘FUNCTION’ or ‘SUBROUTINE’ and indicates how the external routine is to be invoked. NameType is a specification of whether the name was provided as a symbol or as a string literal. Name is the name of the routine to be invoked. Environment is an environment value with the same components as on API_ Start. Arguments is a specification of the arguments to the routine, with the same components as on API_Start. Streams is a specification of the default streams, with the same components as on API_ Start. Traps is the list of traps to be used in processing, with the same components as on API_ Start. Semantics: Invoke an external routine. Set {Outcome to the result of the external routine, or set the indicator of the completion response to ‘D’ if the external routine did not provide a result. If this function is unable to locate the routine the indicator of the completion response is ‘U’. As a result SYNTAX condition 43.1 is raised implicitly. If How indicated that a result from the routine was required but the routine did not provide one the indicator of the completion response is ‘H’. As a result SYNTAX condition 44.1 is raised implicitly. If How indicated that a result from the routine was required but the routine provided one that was too long (see #Limit_String in nnn) the indicator of the completion response is ‘L’. As a result SYNTAX condition 52 is raised implicitly. If the routine failed in a way not indicated by some other indicator the indicator of the completion response is ‘F’. As a result SYNTAX condition 40.1 is raised implicitly. 5.8 External data queue The external data queue interface consists of functions which the configuration shall provide to manipulate an external data queue mechanism. See nnn, nnn, nnn, nnn, and nnn for a description of language features that use the external data queue. The configuration shall provide an external data queue mechanism. The following functions shall be provided:

- Config_Push;
- Config _Queue;
- Config_ Pull;
- Config_Queued.

The configuration may permit the external data queue to be altered in other ways. In the absence of such alterations the external data queue shall be an ordered list. Config_Push adds the specified string to one end of the list, Config _Queue to the other. Config_Pull removes a string from the end that Config_-Push adds to unless the list is empty.

7.7.3 Config Push

Syntax: Config Push(String)

where: String is the value to be retained in the external data queue. Semantics: Add String as an item to the end of the external data queue from which Config_-Pull will remove an item.

7.7.4 Contig_Queue

Syntax: Config Queue (String)

where: String is the value to be retained in the external data queue. Semantics: Add String as an item to the opposite end of the external data queue from which Config_Pull will remove an item.

7.7.5 Config_Pull

Syntax: Config Pull()

Semantics: Retrieve an item from the end of the external data queue to which Config_Push adds an element to the list. Set #Outcome to the value of the retrieved item. If no item could be retrieved the indicator of the completion response is ‘F’.

7.7.6 Contig_Queued

Syntax: Config Queued ()

Semantics:

Get the count of items in the external data queue. Set #Outcome to that number.

7.8 Streams

The streams interface consists of functions which the configuration shall provide to manipulate streams. See nnn, nnn, and nnn for a description of language features which use streams. Streams are identified by names and provide for the reading and writing of data. They shall support the concepts of characters, lines, positioning, default input stream and default output stream. The concept of a persistent stream shall be supported and the concept of a transient stream may be supported. A persistent stream is one where the content is not expected to change except when the stream is explicitly acted on. A transient stream is one where the data available is expected to vary with time.

The concepts of binary and character streams shall be supported. The content of a character stream is expected to be characters. The null string is used as a name for both the default input stream and the default output stream. The null string names the default output stream only when it is an argument to the Config_Stream_Charout operation.

The following functions shall be provided:

- Config_Stream_Charin;
- Config_Stream_Position;
- Config_Stream_Command;
- Config_Stream_State;
- Config_Stream_Charout;
- Config_Stream_Qualified;
- Config_Stream_Unique;
- Config_Stream_Query;
- Config_Stream_Close;
- Config_Stream_Count. The results of these functions are described in terms of the following stems with tails which are stream names:
- #Charin_Position.Stream;
- #Charout_Position.Stream;
- #Linein_Position.Stream;
- #Lineout_Position.Stream.

7.8.1 Config _Stream_Charin

Syntax:

Config Stream Charin(Stream, OperationType) where: Stream is the name of the stream to be processed. OperationType is one of ‘CHARIN’, ‘LINEIN’, or ‘NULL. Semantics: Read from a stream. Increase #Linein_Position.Stream by one when the end-of-line indication is encountered. Increase #Charin_-Position.Stream when the indicator will be ’N’. If OperationType is ‘CHARIN’ the state variables describing the stream will be affected as follows: - when the configuration is able to provide data from a transient stream or the character at position #Charin_Position.Stream of a persistent stream then #Outcome shall be set to contain the data. The indicator of the response shall be ‘N’;

- when the configuration is unable to return data because the read position is at the end of a persistent stream then the indicator of the response shall be ‘O’;
- when the configuration is unable to return data from a transient stream because no data is available and no data is expected to become available then the indicator of the response shall be ‘O’:
- otherwise the configuration is unable to return data and does not expect to be able to return data by waiting; the indicator of the response shall be ‘E’. The data set in #Outcome will either be a single character or will be a sequence of eight characters, each ‘0’ or ‘1’. The choice is decided by the configuration. The eight character sequence indicates a binary stream, see nnn. If OperationType is ‘LINEIN’ then the action is the same as if Operation had been ‘CHARIN’ with the following additional possibility. If end-of-line is detected any character (or character sequence) which is an embedded indication of the end-of-line is skipped. The characters skipped contribute to the change of #Charin_Position.Stream. #Outcome is the null string. If OperationType is ‘NULL’ then the stream is accessed but no data is read.

7.8.2 Config_Stream_Position

Syntax:

Config Stream Position(Stream, OperationType, Position) where: Stream is the name of the stream to be processed. Operation is ‘CHARIN’, ‘LINEIN’, ‘CHAROUT’, or ‘LINEOUT’. Position indicates where to position the stream. Semantics: If the operation is ‘CHARIN’ or ‘CHAROUT’ then Position is a character position, otherwise Position is a line position. If Operation is ‘CHARIN’ or ‘LINEIN’ and the Position is beyond the limit of the existing data then the indicator of the completion response shall be ‘R’. Otherwise if Operation is ‘CHARIN’ or ‘LINEIN’ set #Charin_Position.Stream to the position from which the next Config_Stream_Charin on the stream shall read, as indicated by Position. Set #Linein_Position.Stream to correspond with this position. If Operation is ‘CHAROUT’ or ‘LINEOQUT’ and the Position is more than one beyond the limit of existing data then the indicator of the response shall be ‘R’. Otherwise if Operation is ‘CHAROUT’ or ‘LINEOUT’ then #Charout_Position.Stream is set to the position at which the next Config_Stream_Charout on the stream shall write, as indicated by Position. Set #Lineout_Position.Stream to correspond with this position. If this function is unable to position the stream because the stream is transient then the indicator of the completion response shall be ‘T’.

7.8.3 Config _Stream_Command

Syntax:

Config Stream Command (Stream, Command) where: Stream is the name of the stream to be processed. Command is a configuration-specific command to be performed against the stream. Semantics: Issue a configuration-specific command against a stream. This may affect all state variables describing Stream which hold position information. It may alter the effect of any subsequent operation on the specified stream. If the indicator is set to ‘N’, #Outcome shall be set to information from the command.

7.8.4 Config_Stream_State

Syntax:

Config Stream State (Stream) where: Stream is the name of the stream to be queried. Semantics: Set the indicator to reflect the state of the stream. Return an indicator equal to the indicator that an immediately subsequent Config_Stream_Charin(Stream, ‘CHARIN’) would return. Alternatively, return an indicator of ‘U’.

The remainder of the response shall be a configuration-dependent description of the state of the stream.

7.8.5 Config_Stream_Charout

Syntax:

Config Stream Charout (Stream, Data) where: Stream is the name of the stream to be processed. Data is the data to be written, or ‘EOL’ to indicate that an end-of-line indication is to be written, or a null string. In the first case, if the stream is a binary stream then Data will be eight characters, each ‘0’ or ‘1’, otherwise Data will be a single character. Semantics: When Data is the null string, no data is written. Otherwise write to the stream. The state variables describing the stream will be affected as follows:

- when the configuration is able to write Data to a transient stream or at position #Charout_Position.Stream of a persistent stream then the indicator in the response shall be ‘N’. When Data is not ‘EOL’ then #Charout_-Position.Stream is increased by one. When Data is ‘EOL’, then #Lineout_-Position.Stream is increased by one and #Charout_Position.Stream is increased as necessary to account for any end-of-line indication embedded in the stream;

- when the configuration is unable to write Data the indicator is set to ‘E’.

7.8.6 Config_Stream_Qualified

Syntax:

Config Stream Qualified (Stream) where: Stream is the name of the stream to be processed. Semantics: Set #Outcome to some name which identifies Stream. Return a completion response with indicator ‘B’ if the argument is not acceptable to the configuration as identifying a stream.

7.8.7 Config_Stream_Unique

Syntax:

Config Stream Unique () Semantics: Set #Outcome to a name that the configuration recognizes as a stream name. The name shall not be a name that the configuration associates with any existing data.

7.8.8 Config_Stream_Query

Syntax: Config Stream Query (Stream) where: Stream is the name of the stream to be queried. Semantics: Set #Outcome to ‘B’ if the stream is a binary stream, or to ‘C’ if it is a character stream.

7.8.9 Config_Stream_Close

Syntax:

Config Stream Close (Stream) where: Stream is the name of the stream to be closed. Semantics: #Charout_Position.Stream and #Lineout_Position.Stream are set to 1 unless the stream has existing data, in which case they are set ready to write immediately after the existing data. If this function is unable to position the stream because the stream is transient then the indicator of the completion response shall be ‘T’.

7.8.10 Config _Stream_Count

Syntax: Config Stream Count (Stream, Operation, Option) where: Stream is the name of the stream to be counted. Operation is ‘CHARS’, or ‘LINES’.

Option is ‘N’ or ‘C’. Semantics: If the option is ‘N’, #Outcome is set to zero if:

- the file is transient and no more characters (or no more lines if the Operation is ‘LINES’) are expected to be available, even after waiting;

- the file is persistent and no more characters (or no more lines if the Operation is ‘LINES’) can be obtained from this stream by Config_Stream_-Charin before use of some function which resets #Charin_Position.Stream and #Linein_Position.Stream.

If the option is ‘N’ and #Outcome is set nonzero, #Outcome shall be 1, or be the number of characters (or the number of lines if Operation is ‘LINES’) which could be read from the stream before resetting.

If the option is ‘C’, #Outcome is set to zero if:

- the file is transient and no characters (or no lines if the Operation is ‘LINES’) are available without waiting;
- the file is persistent and no more characters (or no more lines if the Operation is ‘LINES’) can be obtained from this stream by Config_Stream_-Charin before use of some function which resets #Charin_Position.Stream and #Linein_Position.Stream. If the option is ‘C’ and #Outcome is set nonzero, #Outcome shall be the number of characters (or the number of lines if the Operation is ‘LINES’) which can be read from the stream without delay and before resetting.

7.9 External variable pools

The external variable pools interface consists of functions which the configuration shall provide to manipulate variables in external variable pools. See nnn for the VALUE built-in function which uses external variable pools. The configuration shall provide an external variable pools mechanism. The following functions shall be provided:

- Config_Get;
- Config_Set.

The configuration may permit the external variable pools to be altered in other ways.

7.9.1 Config Get

Syntax: Config Get (Poolid, Name) where: Poolid is an identification of the external variable pool. Name is the name of a variable. Semantics: Get the value of a variable with name Name in the external variable pool Poolid. Set Outcome to this value. If Poolid does not identify an external pool provided by this configuration, the indicator of the completion response is ‘P’. If Name is not a valid name of a variable in the external pool, the indicator of the completion response is ‘F’.

7.9.2 Config Set

Syntax: Config Set (Poolid, Name, Value)

where: Poolid is an identification of the external variable pool. Name is the name of a variable. Value is the value to be assigned to the variable. Semantics: Set a variable with name Name in the external variable pool Poolid to Value. If Poolid does not identify an external pool provided by this configuration, the indicator of the completion response is ‘P’. If Name is not a valid name of a variable in the external pool, the indicator of the completion response is ‘F’.

7.10 Configuration characteristics

The configuration characteristics interface consists of a function which the configuration shall provide which indicates choices decided by the configuration.

7.10.1 Config_Constants

Syntax:

Config Constants () Semantics: Set the values of the following state variables:

- if there are any built-in functions which do not operate at NUMERIC DIGITS 9, then set variables #Bif_Digits. (with various tails which are the names of those built-in functions) to the values to be used;
- set variables #Limit_Digits, #Limit_EnvironmentName, #Limit_ExponentDigits, #Limit_Literal, #Limit_MessageInsert, #Limit_Name, #Limit_String, #Limit_-TraceData to the relevant limits. A configuration shall allow a #Limit_-MessageInsert value of 50 to be specified. A configuration shall allow a #Limit_TraceData value of 250 to be specified;
- set #Configuration to a string identifying the configuration;
- set #Version to a string identifying the language processor. It shall have five words. Successive words shall be separated by a blank character. The first four letters of the first word shall be ‘REXX’. The second word shall be the four characters ‘5.00’. The last three words comprise a date. This shall be in the format which is the default for the DATE() built-in function.
- set .nil to a value which compares unequal with any other value that can occur in execution.
- set .local .kernel .system?

7.11 Configuration routines

The configuration routines interface consists of functions which the configuration shall provide which provide functions for a language processor. The following functions shall be provided:

- Config_Trace_Query;
- Config_Trace_Input;
- Config_Trace_Output;
- Config_Default_Input;
- Config_Default_Output;
- Config_Initialization;
- Config_Termination;
- Config_Halt_Query;
- Config_Halt_Reset;
- Config_NoSource;
- Config_Time;
- Config_Random_Seed;
- Config_Random_Next.

7.11.1 Config_Trace_Query

Syntax: Config Trace Query ()

Semantics: Indicate whether external activity is requesting interactive tracing. Set #Outcome to 'Yes' if interactive tracing is currently requested. Otherwise set #Outcome to 'No'.

7.11.2 Config_Trace_Input

Syntax: Config Trace Input ()

Semantics: Set #Outcome to a value from the source of trace input. The source of trace input is determined by the configuration.

7.11.3 Config_Trace_Output

Syntax: Config Trace Output (Line)

where: Line is a string. Semantics:

Write String as a line to the destination of trace output. The destination of trace output is defined by the configuration.

7.11.4 Config _Default_Input

Syntax: Config Default Input ()

Semantics: Set #Outcome to the value that LINEIN( ) would return.

7.11.5 Config_Default_Output

Syntax: Config Default Output (Line)

where: Line is a string. Semantics: Write the string as a line in the manner of LINEOUT(,Line).

7.11.6 Config_Initialization

Syntax:

Config Initialization ()

Semantics: This function is provided only as a counterpart to Trap_Initialization; in itself it does nothing except return the response. An indicator of ‘F’ gives rise to Msg3.1.

7.11.7 Config_Termination

Syntax:

Config Termination ()

Semantics: This function is provided only as a counterpart to Trap_Termination; in itself it does nothing except return the response. An indicator of ‘F’ gives rise to Msg2.1.

7.11.8 Config_Halt_Query

Syntax: Config Halt Query ()

Semantics: Indicate whether external activity has requested a HALT condition to be raised. Set #Outcome to ‘Yes if HALT is requested. Otherwise set #Outcome to ’No’.

5.12.9 Config _Halt_Reset

Syntax: Config Halt Reset ()

Semantics:

Reset the configuration so that further attempts to cause a HALT condition will be recognized.

7.11.9 Config _NoSource

Syntax:

Config NoSource ()

Semantics: Indicate whether the source of the program may or may not be output by the language processor. Set #NoSource to ‘1’ to indicate that the source of the program may not be output by the language processor, at various points in processing where it would otherwise be output. Otherwise, set #NoSource to ‘0’. A configuration shall allow any program to be processed in such a way that Config_NoSource() sets #NoSource to ‘0’. A configuration may allow any program to be processed in such a way that Config_NoSource() sets #NoSource to ‘1’.

7.11.10 Config_Time

Syntax:

Config Time ()

Semantics: Get a time stamp. Set #Time to a string whose value is the integer number of microseconds that have elapsed between 00:00:00 on January first 0001 and the time that Config_Time is called, at longitude zero. Values sufficient to allow for any date in the year 9999 shall be supported. The value returned may be an approximation but shall not be smaller than the value returned by a previous use of the function.

Set #Adjust<Index “#Adjust” #“ ” > to an integer number of microseconds. #Adjust<Index “#Adjust” # ™ > reflects the difference between the local date/time and the date/time corresponding to #Time. #Time + #Adjust<Index “#Adjust” # “ ” > is the local date/time.

7.11.11 Config_Random_Seed

Syntax: Config Random Seed (Seed)

where: Seed is a sequence of up to #Bif_Digits. RANDOM digits. Semantics: Set a seed, so that subsequent uses of Config_Random_Next will reproducibly return quasi-random numbers.

7.11.12 Config_Random_Next

Syntax:

Config Random Next (Min, Max) where: Min is the lower bound, inclusive, on the number returned in #Outcome. Max is the upper bound, inclusive, on the number returned in #Outcome. Semantics: Set #Outcome to a quasi-random nonnegative integer in the range Min to Max.

7.11.13 Config_Options

Syntax: Config Options (String) where: String is a string. Semantics: No effect beyond the effects common to all Config_ invocations. The value of the string will have come from an OPTIONS instruction, see nnn.

7.12 Traps

The trapping interface consists of functions which may be provided by the caller of API_Start (see nnn) as a list of traps. Each trap may be specified or omitted. The language processor shall invoke a specified trap before, or instead of, using the corresponding feature of the language processor itself. This correspondence is implied by the choice of names; that is, a name beginning Trap_ will correspond to a name beginning Config_ when the remainder of the name is the same. Corresponding functions are called with the same interface, with one exception. The exception is that a trap may return a null string. When a trap returns a null string, the corresponding Config_ function is invoked; otherwise the invocation of the trap replaces the potential invocation of the Config_ function. In the rest of this standard, the trapping mechanism is not shown explicitly. It is implied by the use of a Config_ function. The names of the traps are

- Trap_Command;
- Trap_ExternalRoutine;
- Trap_Push;
- Trap_Queue;
- Trap_Pull;
- Trap_Queued;
- Trap_Trace_Query;
- Trap_Trace_Input;
- Trap_Trace_Output;
- Trap_Default_Input;
- Trap_Default_Output;
- Trap_Initialization;
- Trap_Termination;
- Trap_Halt_Query;
- Trap_Halt_Reset.

7.13 Variable pool

How does this fit with variables as properties?

The variable pool interface consists of functions which the configuration shall provide to manipulate the variables and to obtain some characteristics of a Rexx program.

These functions can be called from programs not written in Rexx _ commands and external routines invoked from a Rexx program, or traps invoked from the language processor.

All the functions comprising the variable pool interface shall return with an indication of whether an error occurred. They shall return indicating an error and have no other effect, if #API_Enabled has a value of ‘0’ or if the arguments to them fail to meet the defined syntactic constraints.

These functions interact with the processing of clauses. To define this interaction, the functions are described here in terms of the processing of variables, see nnn.

Some of these functions have an argument which is a symbol. A symbol is a string. The content of the string shall meet the syntactic constraints of the left hand side of an assignment. Conversion to uppercase and substitution in compound symbols occurs as it does for the left hand side of an assignment. The symbol identifies the variable to be operated upon.

Some of the functions have an argument which is a direct symbol. A direct symbol is a string. The content of this string shall meet the syntactic constraints of a VAR_SYMBOL in uppercase with no periods or it shall be the concatenation of a part meeting the syntactic constraints of a stem in uppercase, and a part that is any string. In the former case the symbol identifies the variable to be operated upon. In the latter case the variable to be operated on is one with the specified stem and a tail which is the remainder of the direct symbol.

Functions that have an argument which is symbol or direct symbol shall return an indication of whether the identified variable existed before the function was executed. Clause nnn defines functions which manipulate Rexx variable pools. Where possible the functions comprising the variable pool interface are described in terms of the appropriate invocations of the functions defined in nnn. The first parameter on these calls is the state variable #Pool. If these Var_-functions do not return an indicator ‘N’, ‘R’, or ‘D’ then the API function shall return an error indication.

7.13.1 APL Set

Syntax: API Set(Symbol, Value) where: Symbol is a symbol. Value is the string whose value is to be assigned to the variable. Semantics: Assign the value of Value to the variable identified by Symbol. If Symbol contains no periods or contains one period as its last character: Var_ Set(#Pool, Symbol, ‘0’, Value) Otherwise: Var _Set(#Pool, #Symbol, ‘1’, Value) where: #Symbol is Symbol after any replacements in the tail as described by nnn.

7.13.2 API Value

Syntax: API Value (Symbol)

where: Symbol is a symbol. Semantics: Return the value of the variable identified by Symbol. If Symbol contains no periods or contains one

period as its last character this is the value of #Outcome after: Var _Value(#Pool, Symbol, ‘0’)

Otherwise the value of #Outcome after: Var Value(#Pool, #Symbol, ‘1’) where: #Symbol is Symbol after any replacements in the tail as described by nnn.

7.13.3 API_Drop

Syntax: API Drop (Symbol)

where: Symbol is a symbol. Semantics: Drop the variable identified by Symbol. If Symbol contains no periods or contains one period as its last character: Var Drop(#Pool, Symbol, ‘0’) Otherwise: Var Drop(#Pool, #Symbol, ‘1’) where: #Symbol is Symbol after any replacements in the tail as described by nnn.

7.13.4 API SetDirect

Syntax: API SetDirect (Symbol, Value)

where: Symbol is a direct symbol. Value is the string whose value is to be assigned to the variable. Semantics:

Assign the value of Value to the variable identified by Symbol. If the Symbol contains no period: Var_ Set(#Pool, Symbol, ‘0’, Value)

Otherwise: Var_ Set(#Pool, Symbol, ‘1’, Value)

7.13.5 API_ValueDirect

Syntax: API ValueDirect (Symbol)

where: Symbol is a direct symbol. Semantics: Return the value of the variable identified by Symbol. If the Symbol contains no period: Var _Value(#Pool, Symbol, ‘0’) Otherwise: Var _Value(#Pool, Symbol, ‘1’)

7.13.6 API DropDirect

Syntax: API DropDirect (Symbol)

where: Symbol is a direct symbol. Semantics:

Drop the variable identified by Symbol. If the Symbol contains no period: Var Drop(#Pool, Symbol, ‘0’)

Otherwise: Var Drop(#Pool, Symbol, ‘1’)

7.13.7 APL ValueOther

Syntax: API ValueOther (Qualifier) where: Qualifier is an indication distinguishing the result to be returned including any necessary further qualification. Semantics: Return characteristics of the program, depending on the value of Qualifier. The possibilities for the value to be returned are:

- the value of #Source;
- the value of #Version;
- the largest value of n such that #ArgExists.1.n is ‘1’, see nnn;
- the value of #Arg.1.n where n is an integer value provided as input.

7.13.8 API Next

Syntax: API Next ()

Semantics:

Returns both the name and the value of some variable in the variable pool that does not have the attribute ‘dropped’ or the attribute ‘implicit’ and is not a stem; alternatively return an indication that there is no suitable name to return. When API_Next is called it will return a name that has not previously been returned; the order is undefined. This process of returning different names will restart whenever the Rexx processor executes Var_Reset.

7.13.9 API NextVariable

Syntax: API NextVariable()

Semantics:

Returns both the name and the value of some variable in the variable pool that does not have the attribute ‘dropped’ or the attribute ‘implicit’; alternatively, return an indication that there is no suitable name to return. When API NextVariable is called it will return data about a variable that has not previously been returned; the order is undefined. This process of returning different names will restart whenever the Rexx processor executes Var_Reset. In addition to the name and value, an indication of whether the variable was ‘tailed’ will be returned.

Acknowledgments

RexxLA thanks everybody who has been involved with the Rexx symposia over more than 30 years. Mike Cowlishaw for being the Father of Rexx, Rick McGuire for being the driving force behind all IBM implementations of Rexx, Chip Davis who is Past-President for Life, Mark Hessling for maintaining Regina and building our wonderful symposium application and web site (which enable this publication). Simon Nash for being the architect of Oryx, which became Object Rexx. Jon Wolfers for all his work in converting the website to the database-driven system, and being our treasurer. Gil Barmwater for being our vice-president and treasurer for many years. Rony Flatscher for his teaching and enthousiasm, and for connecting ooRexx to the Java Virtual Machine and its libraries. All the presenters for investing their time to deliver the message. Terry Fuller for being the current vice-president and his help during the symposia. Lee Peedin for his work as President and Vice President for many years. Cathie Dager for being a Rexx "Spark Plug", Pam Taylor for all her work for RexxLA. Virgil Hein and Matthew Emmons for being our representatives within IBM and assisting us through the open sourcing of the IBM products. We thank IBM for graciously donating two of their Rexx implementations, Object Rexx and NetRexx, to RexxLA and the open source community. We thank Per-Olov Jonsson for financing and running our automated test facility - and for all his work on that very important environment. David Ashley his contributions in the early days of "Open" Object Rexx. Marc Remes for saving NetRexx from the Java Platform Modules System. Erich Steinbock and Jean-Louis Faucher for their contributions (and future contributions) to ooRexx.

RexxLA remembers our departed, Kurt Maerker, Brian Marks, Les Koehler, Mark Miesfeld, Oliver Sims and Kermit Kiser, for all their work and pleasant company.

ISBN 978-90-819090-1-3

9 789081 909013 >