Extended Standard Programming Language Rexx

Rexx Language ARB

2 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

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 OOI 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 Character2If #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 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.

7.7.2 Config_ExternalMethod

OOI 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 API 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 API 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.

8

Syntax constructs

8.1 Notation

8.1.1 Backus-Naur Form (BNF)

The syntax constructs in this standard are defined in Backus-Naur Form (BNF). The syntax used in these BNF productions has

- a left-hand side (called identifier);
- the characters ':=“;
- a right-hand side (called bnf_expression). The left-hand side identifies syntactic constructs. The right-hand side describes valid ways of writing a specific syntactic construct. The right-hand side consists of operands and operators, and may be grouped.

8.1.2 Operands

Operands may be terminals or non-terminals. If an operand appears as identifier in some other production it is called a non-terminal, otherwise it is called a terminal. Terminals are either literal or symbolic. Literal terminals are enclosed in quotes and represent literally (apart from case) what must be present in the source being described. Symbolic terminals formed with lower case characters represent something which the configuration may, or may not, allow in the source program, see nnn, nnn, nnn, nnn. Symbolic terminals formed with uppercase characters represent events and tokens, see nnn and nnn. ### Operators The following lists the valid operators, their meaning, and their precedence; the operator listed first has the highest precedence; apart from precedence recognition is from left to right:

- the postfix plus operator specifies one or more repetitions of the preceding construct;
- abuttal specifies that the preceding and the following construct must appear in the given order;
- the operator ‘|’ specifies alternatives between the preceding and the following constructs.

8.1.3 Grouping

Parentheses and square brackets are used to group constructs. Parentheses are used for the purpose of grouping only. Square brackets specify that the enclosed construct is optional.

8.1.4 BNF syntax definition

The BNF syntax, described in BNF, is:

production := identifier ‘:=’ bnf expression

bnf expression t= abuttal | bnf expression ‘|’ abuttal

abuttal t= [abuttal] bnf primary

bnf primary := ‘[T’ bnf expression ’]’ | ‘(’ bnf expression ‘)’ | literal |

identifier | message identifier | bnf primary ‘+’

8.1.5 Syntactic errors

The syntax descriptions (see nnn and nnn) make use of message_identifiers which are shown as Msgnn.nn or Msgnn, where nn is a number. These actions produce the correspondingly numbered error messages (see nnn and nnn).

8.2 Lexical

The lexical level processes the source and provides tokens for further recognition by the top syntax level.

8.2.1 Lexical elements

Events

The fully-capitalized identifiers in the BNF syntax (see nnn) represent events. An event is either supplied by the configuration or occurs as result of a look-ahead in left-to-right parsing. The following events are defined:

- EOL occurs at the end of a line of the source. It is provided by Config_-SourceChar, see nnn;
- EOS occurs at the end of the source program. It is provided by Config_-SourceChar;
- RADIX occurs when the character about to be scanned is ‘X’ or ‘x’ or ‘B’ or ‘b’ not followed by a general_letter, or a digit, or’.’;
- CONTINUE occurs when the character about to be scanned is ‘,’, and the characters after the ‘,’ up to EOL represent a repetition of comment or blank, and the EOL is not immediately followed by an EOS;

- EXPONENT_SIGN occurs when the character about to be scanned is '+' or '-', and the characters to the left of the sign, currently parsed as part of Const_symbol, represent a plain_number followed by 'E' or 'e', and the characters to the right of the sign represent a repetition of digit not followed by a general_letter or'.'.
- would put ASSIGN here for the leftmost '=' in a clause that is not within parentheses or brackets. But Simon not
  happy with message term being an assignment? #### Actions and tokens Mixed case identifiers with an initial capital letter cause an action when they appear as operands ina production. These actions perform further tests and create tokens for use by the top syntax level. The following actions are defined:
- Special supplies the source recognized as special to the top syntax level;
- Eol supplies a semicolon to the top syntax level;
- Eos supplies an end of source indication to the top syntax level;
- Var_symbol supplies the source recognized as Var_symbol to the top syntax level, as keywords or VAR_SYMBOL tokens, see nnn. The characters in a Var_symbol are converted by Config_Upper to uppercase. Msg30.1 shall be produced if Var_symbo/ contains more than #Limit_Name characters, see nnn;
- Const_symbol supplies the source recognized as Const_symbo! to the top syntax level. If itis a number it is passed as a NUMBER token, otherwise it is passed as a CONST_SYMBOL token. The characters in a Const_symbol are converted by Config_Upper to become the characters that comprise that NUMBER or CONST_SYMBOL. Msg30.1 shall be produced if Const_-symbo! contains more than #Limit_Name characters;
- Embedded_quotation_mark records an occurrence of two consecutive quotation marks within a string delimited by quotation marks for further processing by the String action;
- Embedded_apostrophe records an occurrence of two consecutive apostrophes within a string delimited by apostrophes for further processing by the String action;
- String supplies the source recognized as String to the top syntax level as a STRING token. Any occurrence of Embedded_quotation_mark or Embedded_apostrophe is replaced by a single quotation mark or apostrophe, respectively. Msg30.2 shall be produced if the resulting string contains more than #Limit_Literal characters;
- Binary_string supplies the converted binary string to the top syntax level as a STRING token, after checking conformance to the binary_string syntax. If the binary_string does not contain any occurrence of a binary_digit, a string of length 0 is passed to the top syntax level. The occurrences of binary_-digit are concatenated to form a number in radix 2. Zero or 4 digits are added at the left if necessary to make the number of digits a multiple of 8. If the resulting number of digits exceeds 8 times #Limit_Literal then Msg30.2 shall be produced. The binary digits are converted to an encoding, see nnn. The encoding is supplied to the top syntax level as a STRING token;

- Hex_string supplies the converted hexadecimal string to the top syntax level as a STRING token, after checking conformance to the hex_string syntax. If the hex_string does not contain any occurrence of a hex_digit, a string of length 0 is passed to the top syntax level. The occurrences of hex_digit are each converted to a number with four binary digits and concatenated. 0 to 7 digits are added at the left if necessary to make the number of digits a multiple of 8. If the resulting number of digits exceeds 8 times #Limit_Literal then Msg30.2 shall be produced. The binary digits are converted to an encoding. The encoding is supplied to the top syntax level as a STRING token;
- Operator supplies the source recognized as Operator (excluding characters that are not operator_char) to the top syntax level. Any occurrence of an ofher_negator within Operator is supplied as ' ';
- Blank records the presence of a blank. This may subsequently be tested (see nnn). Constructions of type Number, Const_symbol, Var_symbol or String are called operands. 6.2.1.3 Source characters The source is obtained from the configuration by the use of Config_SourceChar (see nnn). If no character is available because the source is not a correct encoding of characters, message Msg22.1 shall be produced. The terms extra_letter, other_blank_character, other_negator, and other_character used in the productions of the lexical level refer to characters of the groups extra_letters (see nnn),

other_blank_characters (see nnn), other_negators (see nnn) and other_characters (see nnn), respectively.

Rules

In scanning, recognition that causes an action (see nnn) only occurs if no other recognition is possible, except that Embedded_apostrophe and Embedded_-quotation_mark actions occur wherever possible.

8.2.2 Lexical level

8.2.3 Interaction between levels of syntax

When the lexical process recognizes tokens to be supplied to the top level, there can be changes made or tokens added. Recognition is performed by the lexical process and the top level process ina synchronized way. The tokens produced by the lexical level can be affected by what the top level syntax has recognized. Those tokens will affect subsequent recognition by the top level. Both processes operate on the characters and the tokens in the order they are produced. The term “context” refers to the progress of the recognition at some point, without consideration of unprocessed characters and tokens. If a token which is ‘+’, ‘-’, ' or ‘(’ appears in a lexical level context (other than after the keyword ‘PARSE’) where the keyword ‘VALUE’ could appear in the corresponding top level context,

then ‘VALUE’ is passed to the top level before the token is passed. If an ‘=’ operator_char appears in a lexical level context where it could be the ‘=’ of an assignment or message_instruction in the corresponding top level context then it is recognized as the ‘=“’ of that instruction. (It will be outside of brackets and parentheses, and any Var_symbo/ immediately preceding it is passed as a VAR_SYMBOL). If an operand is followed by a colon token in the lexical level context then the operand only is passed to the top level syntax as a LABEL, provided the context permits a LABEL. Except where the rules above determine the token passed, a Var_symbol is passed as a terminal (a keyword) rather than as a VAR_SYMBOL under the following circumstances:

- if the symbol is spelled ‘WHILE’ or ‘UNTIL’ it is a keyword wherever a VAR_-SYMBOL would be part of an expression within a do_specification,
- if the symbol is spelled ‘TO’ , ‘BY’, or ‘FOR’ it is a keyword wherever a VAR_SYMBOL would be part of an expression within a do_rep;
- if the symbol is spelled ‘WITH’ it is a keyword wherever a VAR_SYMBOL would be part of a parsevalue, or part of an expression or taken_constant within address;
- if the symbol is spelled ‘THEN’ it is keyword wherever a VAR_SYMBOL would be part of an expression immediately following the keyword ‘IF’ or ‘WHEN’. Except where the rules above determine the token passed, a Var_symbol is passed as a keyword if the spelling of it matches a keyword which the top level syntax recognizes in its current context, otherwise the Var_symbol is passed as a VAR_SYMBOL token. In a context where the top level syntax could accept a ‘||’ token as the next token, a’||’ operator ora’ ’ operator may be inferred and passed to the top level provided that the next token from the lexical level is a left parenthesis or an operand that is not a keyword. If the blank action has recorded the presence of one or more blanks to the left of the next token then the ’ ’ operator is inferred. Otherwise, a’||’ operator is inferred, except if the next token is a left parenthesis following an operand (see nnn); in this case no operator is inferred. When any of the keywords ‘OTHERWISE’, ‘THEN’, or ‘ELSE’ is recognized, a semicolon token is supplied as the following token. A semicolon token is supplied as the previous token when the ‘THEN’ keyword is recognized. A semicolon token is supplied as the token following a LABEL.

Reserved symbols

A Const_symbol which starts with a period and is not a Number shall be spelled .MN, .RESULT, .RC, .RS, or .SIGL otherwise Msg50.1 is issued.

Function name syntax

A symbol which is the leftmost component of a function shall not end with a period, otherwise Msg51.1 is issued.

8.3 Syntax

8.3.1 Syntax elements

The tokens generated by the actions described in nnn form the basis for recognizing larger constructs.

8.3.2 Syntax level

starter:=x3j18

x3j18:=program Eos | Msg35.1

program := [label list] [ncl] [requires+] [prolog_instruction+] (class definition [requires+])+ requires := ‘REQUIRES’ (taken constant | Mgg19.8) “‘;’+ prolog instruction:= (package | import | options) nel package = ‘PACKAGE’(NAME | Msgnn) import = ‘IMPORT’ (NAME | Msgnn) [‘.’] options := ‘OPTIONS’ (symbol+ | Msgnn) nel := null _clause+ | Msg21.1 null clause = ‘;’ [label list] label list = (LABEL ‘;’)+ class definition = class [property info] [method definition+] class = ‘CLASS’ (taken constant | Msg19.12) [class_option+]

[‘INHERIT’ (taken constant | Msg19.13)+] nel visibility | modifier | ‘BINARY’ | ‘DEPRECATED’ ‘EXTENDS’ (NAME | Msgnn) ‘USES’ (NAMElist | Msgnn) | ‘IMPLEMENTS’ (NAMElist | Msgnn)

class option

numeric digits:= ‘DIGITS’ [expression]

external | metaclass | submix /* | ’PUBLIC! */

external = ‘EXTERNAL’ (STRING | Msg19.14) metaclass = ‘METACLASS’ (taken constant | Msg19.15) submix = ‘MIXINCLASS’ (taken constant | Msg19.16) | ‘SUBCLASS’ (taken constant | Msg19.17) visibility = ‘PUBLIC’ | ‘PRIVATE’ modifier = ‘ABSTRACT’ | ‘FINAL’ | ‘INTERFACE’ | ‘ADAPTER’! NAMElist = NAME [(‘,’ (NAME | Msgnn))+] property _info = numeric | property assignment | properties | trace numeric = ‘NUMERIC’ (numeric digits | numeric form | Msg25.15) numeric form = ‘FORM’ [‘ENGINEERING’ | ‘SCIENTIFIC’] property agsignment := NAME | assignment properties := ‘PROPERTIES’ (properties option+ | Msgnn) properties option := properties vigibility | properties modifier properties visibility := ‘INHERITABLE’ | ‘PRIVATE’ | ‘PUBLIC’ | ‘INDIRECT’ properties modifier := ‘CONSTANT’ | ‘STATIC’ | ‘VOLATILE’ | ‘TRANSIENT’ trace := ‘TRACE’ [‘ALL’ | ‘METHODS’ | ‘OFF’ | ‘RESULTS’] method definition = (method [expose ncl]| routine) balanced expose = ‘EXPOSE’ variable list method = ‘METHOD’ (taken constant | Msg19.9) [‘(’ assigncommalist | Msgnn (‘)’ | Msgnn)] [method option+] nel assigncommalist assignment [(‘,’ (assignment | Msgnn))+]

method visibility | method modifier | ‘PROTECT’ | ‘RETURNS’ (term | Msgnn) | ‘SIGNAL’ (termcommalist | Msgnn)

method option

‘DEPRECATED’! ‘CLASS’ | ‘ATTRIBUTE’ | /*‘PRIVATE’* | / guarded guarded := ‘GUARDED’ | ‘UNGUARDED! method visibility := ’INHERITABLE’ | ‘PRIVATE’ | ‘PUBLIC’ | ‘SHARED!’ method modifier := ‘ABSTRACT’ | ‘CONSTANT’ | ‘FINAL’ | ‘NATIVE’ | ‘STATIC’ termcommalist := term [(‘,’ (term | Msgnn))+] routine := ‘ROUTINE’ (taken constant | Msg19.11) [‘PUBLIC’] nel balanced:= instruction list [‘END’ Msgl0.1] instruction list:= instruction+

/* The second part ig about groups */

instruction = group | gingle instruction nel group = do ncl | if | loop nel | select nel do = do specification nel [instruction+] [group_handler] (‘END’ [NAME] | Eos Msg14.1 | Msg35.1) group option := ‘LABEL’ (NAME | Msgnn) | ‘PROTECT’ (term | Msgnn)

group handler catch | finally catch finally

‘CATCH’ [NAME ‘=’] (NAME | Msgnn) nel [instruction+]

catch = /* FINALLY implies a semicolon. */ finally = ‘FINALLY’ nel (instruction+ | Msgnn) if := ‘IF’ expression [ncl] (then | Msg18.1) [else] then := ‘THEN’ nel (instruction | EOS Msg14.3 | ‘END’ Msg10.5) else := ‘ELSE’ nel (instruction | EOS Msg14.4 | ‘END’ Msg10.6) loop := ‘LOOP’ [group_option+] [repetitor] [conditional] nel

instruction+ [group_handler] loop ending

loop ending ‘END’ [VAR SYMBOL] | EOS Msg14.n | Msg35.1

conditional = ‘WHILE’ whileexpr | ‘UNTIL’ untilexpr untilexpr = expression whileexpr = expression repetitor = assignment [count option+] | expression | over ‘FOREVER’ count option = loopt | loopb | loopf loopt = ‘TO’ expression loopb = ‘BY’ expression loopf = ‘FOR’ expression over = VAR SYMBOL ‘OVER’ expression

NUMBER ‘OVER’ Msg31.1
CONST SYMBOL ‘OVER’ (Msg31.2 | Msg31.3)

select := ‘SELECT’ [group option+] ncl select _ body [group_handler] (‘END’ [NAME Msg10.4] | EOS Msgl14.2 | Msg7.2)

(when | Msg7.1) [when+] [otherwise]

‘WHEN’ expresgion [ncl] (then | Msg18.2)

‘OTHERWISE’ ncl [instruction+]

select body when otherwise

/* Third part is for single instructions. */ single instruction:= assignment | message instruction | keyword instruction | command assignment := VAR SYMBOL ‘#’ expression NUMBER ‘#’ Msg31.1 CONST SYMBOL ‘#!’ (Msg31.2 | Mgg31.3)

message instruction := message term | message term ‘#’ expression keyword instruction:= address | arg | call | drop | exit interpret | iterate | leave

nop | numeric | options parse | procedure | pull | push | queue

raise | reply | return | say | signal | trace | use ‘THEN’ Msg8.1 ‘ELSE’ Msg8.2 ‘WHEN’ Msg9.1 | ‘OTHERWISE’ Msg9.2 command = expression address = ‘ADDRESS’ [(taken constant [expression]]

taken constant symbol | STRING

valueexp = ‘VALUE’ expression connection = ad option+ ad option = error | input | output | Msg25.5 error = ‘ERROR!’ (resourceo | Msg25.14) input = ‘INPUT’ (resourcei | Msg25.6) resourcei := resources | ‘NORMAL’ output = ‘OUTPUT’ (resourceo | Mgg25.7) resourceo := ‘APPEND’ (resources | Msg25.8) | ‘REPLACE’ (resources | Msg25.9) | resources | ‘NORMAL’ resources := ‘STREAM’ (VAR_SYMBOL | Msg53.1) | ‘STEM’ (VAR_SYMBOL | Msg53.2) vref := ‘(’ var symbol (‘)’ | Msg46.1) var symbol = VAR_SYMBOL | Msg20.1

arg := ‘ARG’ [template list]

eall := ‘CALL’ (callon_spec | (taken constant | vref | Msg19.2) [expression list]) callon spec := ‘ON’ (callable condition | Msg25.1)

[‘NAME’ (symbol constant term | Msg19.3)] | ‘OFF’ (callable condition | Msg25.2)

symbol constant term := term

callable condition:= ‘ANY’ | ‘ERROR’ | ‘FAILURE’ | ‘HALT’ | ‘NOTREADY’ | ‘USER’ (symbol constant term | Msg19.18)

condition := callable condition | ‘LOSTDIGITS’ | ‘NOMETHOD’ | ‘NOSTRING’ | ‘NOVALUE’ | ’SYNTAX!

expression list do specification do simple

expr | [expr] ‘,’ [expression list] do simple | do repetitive ‘DO’ [group_option+]

do repetitive = do simple (dorep | conditional | dorep conditional) dorep = ‘FOREVER’ | repetitor drop = ‘DROP’ variable list variable list = (vref | var_-symbol)+ exit = ‘EXIT’ [expression] forward = ‘FORWARD’ [forward _option+ | Msg25.18] forward option = ‘CONTINUE’ | ArrayArgOption | MessageOption | ClassOption | ToOption ArrayArgOption:=‘ARRAY’ arguments | ‘ARGUMENTS’ term MessageOption :=‘MESSAGE’ term ClassOption =‘CLASS’ term ToOption =‘TO’ term guard = ‘GUARD’ (‘ON’ | Msg25.22) [(‘WHEN’ | Msg25.21) expression] | (‘OFF’ | Msg25.19) [(‘WHEN’ | Msg25.21) expression] interpret = ‘INTERPRET’ expression iterate = ‘ITERATE’ [VAR SYMBOL | Msg20.2] leave = ‘LEAVE’ [VAR SYMBOL | Msg20.2] nop = ‘NOP!’ numeric = ‘NUMERIC’ (numeric digits | numeric form

numeric fuzz | Msg25.15)

numeric digits ‘DIGITS’ [expression] numeric form ‘FORM’ [numeric form suffix] numeric form suffix:=(‘ENGINEERING’ |‘SCIENTIFIC’|valueexp | Msg25.11) numeric fuzz ‘FUZZ’ [expression]

options = ‘OPTIONS’ expression parse = ‘PARSE’ [translations] (parse type |Msg25.12) [template list] translations := ‘CASELESS’ [‘UPPER’ | ‘LOWER’] | (‘UPPER’ | ‘LOWER’) [‘CASELESS’] parse type = parse key | parse value | parse var | term parse key = ‘ARG’ | ‘PULL’ | ‘SOURCE’ | ‘LINEIN’ | ‘VERSION! parse value = ’VALUE’ [expression] (‘WITH’ | Mgg38.3) parse var = ‘VAR’ var symbol template := NAME [([pattern] NAME) +] pattern:= STRING | [indicator] NUMBER | [indicator] ‘(’ symbol ‘)’ indicator := ‘+’ | ‘-’ | ‘s! procedure = ’PROCEDURE’ [expose | Msg25.17] pull = ‘PULL’ [template list] push = ‘PUSH’ [expression] queue = ‘QUEUE’ [expression] raise = ‘RAISE’ conditions (raise option | Msg25.24)

conditions = ‘ANY’ | ‘ERROR’ term | ‘FAILURE’ term | ‘HALT’| ‘LOSTDIGITS’ | ‘NOMETHOD’ | ‘NOSTRING’ | “NOTREADY’ | ‘NOVALUE’ | ‘PROPAGATE’ | ‘SYNTAX’ term | ‘USER’ (symbol constant term | Msg19.18) | Msg25.23 raise option := ExitRetOption | Description | ArrayOption

ExitRetOption := ‘EXIT! [term] | ’RETURN’ [term] Description =‘DESCRIPTION’ term ArrayOption = ‘ADDITIONAL’ term | “‘ARRAY’ arguments reply = ‘REPLY’ [expression] return = ‘RETURN’ [expression] say = ‘SAY’ [expression] signal = ‘SIGNAL’ (signal spec | valueexp

symbol constant term | Msg19.4)

signal spec := ‘ON’ (condition | Msg25.3) [‘NAME’ (symbol constant term | Msg19.3)] | ‘OFF’ (condition | Msg25.4)

trace = ‘TRACE’ [(taken_constant | Msgl19.6) | valueexp] use = ‘USE’ (‘ARG’ | Msg25.26) [use list] use list = VAR_SYMBOL | [VAR SYMBOL] ‘,’ [use list]

/* Note: The next part describes templates. */ template list template | [template] ‘,’ [template list]

template = (trigger | target | Msg38.1)+ target = VAR_SYMBOL | ‘.!’ trigger = pattern | positional pattern = STRING | vrefp vrefp = ‘(’ (VAR_SYMBOL | Msgl19.7) (‘)’ | Msg46.1) positional = absolute positional | relative positional absolute positional:= NUMBER ‘=’ position pogition := NUMBER | vrefp | Msg38.2 relative posgitional:= (‘+’ | ‘-“’) position

/* Note: The final part specifies the various forms of symbol, and expression. */

symbol = VAR_SYMBOL | CONST SYMBOL | NUMBER expression = expr [(‘,’ Msg37.1) | (‘)’ Msg37.2)] expr = expr alias

expr alias and expression

expr alias or operator and expression

or operator := ‘|’ | ‘&&! and expression := comparison | and expression’&’ comparison comparison := concatenation | comparison comparison operator concatenation comparison _operator:= normal compare | strict compare normal compare:= ‘=! | “! | hep! | tact | tot | tet | toet len! W>! <! strict_compare:=’==’ | ‘=’ | ‘»!’ | ‘«!’ | ‘»e! |’«s! | “‘>t |’<c! concatenation := addition | concatenation (’ ’ | ‘||’) addition addition := multiplication | addition additive operator multiplication additive operator:= ‘+’ | ’-! multiplication t= power expression

multiplication multiplicative operator

power expression multiplicative operator:= ’*’ | ‘/’ | ‘//’ | ‘%! power expression := prefix expression | power expression’**’ prefix expression prefix expression := (‘+’ | ‘-’ | ’ ’) prefix expression term | Msg35.1 /* “Stub” has to be identified semantically? */

term = simple term [‘.’ (term | Msgnn)] simple term := symbol | STRING | invoke | indexed ‘(’ expression (‘)’ | Msg36) initializer | message term ‘##! message term:= term (’~’ | ‘~~’) method name [arguments] term ‘[’[expression list] (’]’ | Msg36.2)

method name:=(taken constant | Msg19.19) [‘:’ (VAR_SYMBOL | Msg19.21)] /* Method-call without arguments ig syntactically like symbol. / / Editor - not sure

of my notes about here. */

invoke := (symbol | STRING) arguments arguments := ‘#(’ [expression list] (‘)’ | Meg36) expression list := expregsion | [expression] ‘,’ [expression list] indexed = (symbol | STRING) indices indices = ‘#[’ [expression list] (’]’ | Msg36.n) initializer = ‘[’expression list (’]’ | Msg36.n)

8.4 Syntactic information

8.4.1 VAR_SYMBOL matching

Any VAR_SYMBOL in a do_ending must be matched by the same VAR_-SYMBOL occurring at the start of an assignment contained in the do_specification of the do that contains both the do_specification and the do_ending, as described in nnn. If there is a VAR_SYMBOL in a do_ending for which there is no assignment in the corresponding do_specification then message Msg10.3 is produced and no further activity is defined. If there is a VAR_SYMBOL in a do_ending which does not match the one occurring in the assignment then message Msg10.2 is produced and no further activity is defined. An iterate or leave must be contained in the instruction_list of some do with a do_specification which is do_repetitive, otherwise a message (Msg28.2 or Msg28.1 respectively) is produced and no further activity is defined. If an iterate or leave contains a VAR_SYMBOL there must be a matching VAR_SYMBOL ina do_specification, otherwise a message (Msg28.1, Msg28.2, Msg28.3 or Msg28.4 appropriately) is produced and no further activity is defined. The matching VAR_SYMBOL will occur at the start of an assignment in the do_specification. Tne do_specification will be associated with a do by nnn. The /ferafe or leave will be a single instruction in an instruction_list associated with a do by nnn. These two dos shall be the same, or the latter nested one or more levels within the former. The number of levels is called the nesting_correction and influences the semantics of the iterafe or leave. It is zero if the two dos are the same. The nesting_correction for /ferates or leaves that do not contain VAR_SYMBOL is zero.

8.4.2 Trace-only labels

Instances of LABEL which occur within a grouping_instruction and are not in a nc/ at the end of that grouping_instruction are instances of trace-only labels.

8.4.3 Clauses and line numbers

The activity of tracing execution is defined in terms of clauses. A program consists of clauses, each clause ended by a semicolon special token. The semicolon may be explicit in the program or inferred. The line number of a clause is one more than the number of EOL events recognized before the first token of the clause was recognized.

8.4.4 Nested IF instructions

The syntax specification nnn allows ‘IF’ instructions to be nested and does not fully specify the association of an ‘ELSE’ keyword with an ‘IF’ keyword. An ‘ELSE’ associates with the closest prior ‘IF’ that it can associate with in conformance with the syntax.

8.4.5 Choice of messages

The specifications nnn and nnn permit two alternative messages in some circumstances. The following rules apply:

- Msg15.1 shall be preferred to Msg15.3 if the choice of Msg15.3 would result in the replacement for the insertion being a blank character;
- Msg15.2 shall be preferred to Msg15.4 if the choice of Msg15.4 would result in the replacement for the insertion being a blank character;
- Msg31.3 shall be preferred to Msg31.2 if the replacement for the insertion in the message starts with a period;
- Preference is given to the message that appears later in the list: Msg21.1, Msg27.1, Msg25.16, Msg36, Msg38.3, Msg35.1, other messages.

8.4.6 Creation of messages

The message_identifiers in clause 6 correlate with the tails of stem #ErrorText., which is initialized in nnn to identify particular messages. The action of producing an error message will replace any insertions in the message text and present the resulting text, together with information on the origin of the error, to the configuration by writing on the default error stream. Further activity by the language processor is permitted, but not defined by this standard. The effect of an error during the writing of an error message is not defined.

Error message prefix

The error message selected by the message number is preceded by a prefix. The text of the prefix is #ErrorText.0.1 except when the error is in source that execution of an interactive trace interpret instruction (see nnn) is processing, in which case the text is #ErrorText.0.2. The insert called in these texts is the message number. The insert called is the line number of the error. The line number of the error is one more than the number of EOL events encountered before the error was detectable, except for messages Msg6.1, Msg14, Msg14.1, Msg14.2, Msg14.3, and Msg14.4. For Msg6.1 it is one more than the number of EOL events encountered before the line containing the unmatched ’/*’. For the others, it is the line number of the clause containing the keyword referenced in the message text. The insert called is the value provided on the API_ Start function which started processing of the program, see nnn.

8.5 Replacement of insertions

Within the text of error messages, an insertion consists of the characters ‘<’, ‘>’, and what is between those characters. There will be a word in the insertion that specifies the replacement text, with the following meaning:

- if the word is ‘hex-encoding’ and the message is not Msg23.1 then the replacement text is the value of the leftmost character which caused the source to be syntactically incorrect. The value is in hexadecimal notation;
- if the word is ‘token’ then the replacement text is the part of the source program which was recognized as the detection token, or in the case of Msg31.1 and Msg31.2, the token before the detection token. The detection token is the leftmost token for which the program up to and including the token could not be parsed as the left part of a program without causing a message. If the detection token is a semicolon that was not present in the source but was supplied during recognition then the replacement is the previous token;
- if the word is ‘position’ then the replacement text is a number identifying the detection character. The detection character is the leftmost character in the hex_string or binary_string which did not match the required syntax. The number is a count of the characters in the string which preceded the detection character, including the initial quote or apostrophe. In deciding the leftmost blank in a quoted string of radix ‘X’ or ‘B’ that is erroneous not that: A blank as the first character of the quoted string is an error. The leftmost embedded sequence of blanks can validly follow any number of non-blank characters. Otherwise a blank run that follows an odd numbered sequence of non-blanks (or a number not a multiple of four in the case of radix ‘B’) is not valid. If the string is invalid for a reason not described above, the leftmost blank of the rightmost sequence of blanks is the invalid blank to be referenced in the message;
- if the word is ‘char’ then the replacement text is the detection character;
- if the word is ‘linenumber’ then the replacement text is the line number of a clause associated with the error. The wording of the message text specifies which clause that is;
- if the word is ‘keywords’ then the replacement text is a list of the keywords that the syntax would allow at the context where the error occurred. If there are two keywords they shall be separated by the four characters ’ or ‘. If more, the last shall be preceded by the three characters ’or’ and the others shall be followed by the two characters ‘,’. The keywords will be uppercased and in alphabetical order.

Replacement text is truncated to #Limit_MessageInsert characters if it would otherwise be longer than that, except for a keywords replacement. When an insert is both truncated and appears within quotes in the message, the three characters ‘…’ are inserted in the message after the trailing quote.

8.6 Syntactic equivalence

If a message_term contains a ‘[’ it is regarded as an equivalent message_term without a ‘[’, for execution. The equivalent is term~‘[]’(expression_list). See nnn.
If a message_instruction has the construction message_term ‘=’ expression it is regarded as equivalent to a message_term with the same components as the message_term left of the ‘=’, except that the taken_constant has an ‘=’ character appended and arguments has the expression from the right of the ‘=’ as an extra first argument. See nnn.

9

Evaluation

The syntax section describes how expressions and the components of expressions are written in a program. It also describes how operators can be associated with the strings, symbols and function results which are their operands.

This evaluation section describes what values these components have in execution, or how they have no value because a condition is raised.

This section refers to the DATATYPE built-in function when checking operands, see nnn. Except for considerations of limits on the values of exponents, the test:

datatype (Subject) == ‘NUM’ is equivalent to testing whether the subject matches the syntax: num := [blank+] [‘+’ | ‘-’] [blank+] number [blank+]

For the syntax of number see nnn.

When the matching subject does not include a ‘-’ the value is the value of the number in the match, otherwise the value is the value of the expression (0 - number).

The test:

datatype (Subject , ‘W’)

is a test that the Subject matches that syntax and also has a value that is “whole”, that is has no non-zero fractional part.

When these two tests are made and the Subject matches the constraints but has an exponent that is not

in the correct range of values then a condition is raised: call #Raise ‘SYNTAX’, 41.7, Subject

This possibility is implied by the uses of DATATYPE and not shown explicitly in the rest of this section nnn.

9.1 Variables

The values of variables are held in variable pools. The capabilities of variable pools are listed here, together with the way each function will be referenced in this definition.

The notation used here is the same as that defined in sections nnn and nnn, including the fact that the Var_ routines may return an indicator of ‘N’, ‘S’ or ‘X’.

Each possible name in a variable pool is qualified as tailed or non-tailed name; names with different qualification and the same spelling are different items in

the pool. For those Var_ functions with a third argument this argument indicates the qualification; it is ‘1’ when addressing tailed names or ‘0’ when addressing non-tailed names.

Each item in a variable pool is associated with three attributes and a value. The attributes are ‘dropped’ or ‘not-dropped’, ‘exposed’ or ‘not-exposed’ and ‘implicit’ or ‘not-implicit’.

A variable pool is associated with a reference denoted by the first argument, with name Pool. The value of Pool may alter during execution. The same name, in conjunction with different values of Pool, can correspond to different values.

9.1.1 Var_Empty

Var_Empty (Pool)

The function sets the variable pool associated with the specified reference to the state where every name is associated with attributes ‘dropped’, ‘implicit’ and ‘not-exposed’.

9.1.2 Var Set

Var _Set(Pool, Name, ‘0’, Value)

The function operates on the variable pool with the specified reference. The name is a non-tailed name. If the specified name has the ‘exposed’ attribute then Var_Set operates on the variable pool referenced by #Upper in this pool and this rule is applied to that pool. When the pool with attribute ‘not-exposed’ for this name is determined the specified value is associated with the specified name. It also associates the attributes ‘not-dropped’ and ‘not-implicit’. If that attribute was previously ‘not-dropped’ then the indicator returned is ‘R’. The name is a stem if it contains just one period, as its rightmost character. When the name is a stem Var_Set(Pool, TailedName, ‘1’,Value) is executed for all possible valid tailed names which

have Name as their stem, and then those tailed-names are given the attribute ‘implicit’. Var _Set(Pool, Name, ‘1’, Value)

The function operates on the variable pool with the specified reference. The name is a tailed name. The left part of the name, up to and including the first period, is the stem. The stem is a non-tailed name. If the specified stem has the ‘exposed’ attribute then Var_Set operates on the variable pool referenced by #Upper in this pool and this rule is applied to that pool. When the pool with attribute ‘not-exposed’ for the stem is determined the name is considered in that pool. If the name has the ‘exposed’ attribute then the

variable pool referenced by #Upper in the pool is considered and this rule applied to that pool. When the pool with attribute ‘not-exposed’ is determined the specified value is associated with the specified name. It also associates the attributes ‘not-dropped’ and ‘not-implicit’ . If that attribute was previously ‘not-dropped’ then the indicator returned is ‘R’.

9.1.3 Var_Value

Var _Value(Pool, Name, ‘0’)

The function operates on the variable pool with the specified reference. The name is a non-tailed name. If the specified name has the ‘exposed’ attribute then Var_Value operates on the variable pool referenced by #Upper in this pool and this rule is applied to that pool. When the pool with attribute ‘not-exposed’ for this name is determined the indicator returned is ‘D’ if the name has ‘dropped’ associated, ‘N’ otherwise. In the former case #Outcome is set equal to Name, in the latter case #Outcome is set to the value most

recently associated with the name by Var_Set. Var _Value(Pool, Name, ‘1’)

The function operates on the variable pool with the specified reference. The name is a tailed name. The left part of the name, up to and including the first period, is the stem. The stem is a non-tailed name. If the specified stem has the ‘exposed’ attribute then Var_Value operates on the variable pool referenced by #Upper in this pool and this rule is applied to that pool. When the pool with attribute ‘not-exposed’ for the stem is determined the name is considered in that pool. If the name has the ‘exposed’ attribute then the variable pool referenced by #Upper in the pool is considered and this rule applied to that pool. When the pool with attribute ‘not-exposed’ is determined the indicator returned is ‘D’ if the name has ‘dropped’ associated, ‘N’ otherwise. In the former case #Outcome is set equal to Name, in the latter case #Outcome is set to the value most recently associated with the name by Var_Set.

9.1.4 Var_Drop

Var _Drop(Pool, Name, ‘0’)

The function operates on the variable pool with the specified reference. The name is a non-tailed name. If the specified name has the ‘exposed’ attribute then Var_Drop operates on the variable pool referenced by #Upper in this pool and this rule is applied to that pool. When the pool with attribute ‘not-exposed’ for this name is determined the attribute ‘dropped’ is associated with the specified name. Also, when the name is a stem, Var_Drop(Pool,TailedName,‘1’) is executed for all possible valid tailed names which have Name as astem.

Var _Drop(Pool, Name, ‘1’)

The function operates on the variable pool with the specified reference. The name is a tailed name. The left part of the name, up to and including the first period, is the stem. The stem is a non-tailed name. If the specified stem has the ‘exposed’ attribute then Var_Drop operates on the variable pool referenced by #Upper in this pool and this rule is applied to that pool. When the pool with attribute ‘not-exposed’ for the stem is determined the name is considered in that pool. If the name has the ‘exposed’ attribute then the variable pool referenced by #Upper in the pool is considered and this rule applied to that pool. When the pool with attribute ‘not-exposed’ is determined the attribute ‘dropped’ is associated with the specified name.

9.1.5 Var_Expose

Var_Expose (Pool, Name, ‘0’)

The function operates on the variable pool with the specified reference. The name is a non-tailed name. The attribute ‘exposed’ is associated with the specified name. Also, when the name is a stem, Var_Expose(Pool, TailedName,‘1’) is executed for all possible valid tailed names which have Name as a stem.

Var_Expose (Pool, Name, ‘1’)

The function operates on the variable pool with the specified reference. The name is a tailed name. The attribute ‘exposed’ is associated with the specified name.

9.1.6 Var_Reset

Var_ Reset (Pool)

The function operates on the variable pool with the specified reference. It establishes the effect of subsequent API_Next and API_NextVariable functions (see sections nnn and nnn). A Var_Reset is implied by any API_ operation other than API_Next and API_NextVariable.

9.2 Symbols

For the syntax of a symbol see nnn.

The value of a symbol which is a NUMBER or a CONST_SYMBOL which is not a reserved symbol is the content of the appropriate token.

The value of a VAR_SYMBOL which is “taken as a constant” is the VAR_-SYMBOL itself, otherwise the VAR_SYMBOL identifies a variable and its value may vary during execution.

Accessing the value of a symbol which is not “taken as a constant” shall result in trace output, see nnn: if #Tracing.#Level == ‘I’ then call #Trace Tag

where Tag is ‘>L>’ unless the symbol is a VAR_SYMBOL which, when used as an argument to Var_Value, does not yield an indicator ‘D’. In that case, the Tag is ’>V>“.

9.3 Value of a variable

If VAR_SYMBOL does not contain a period, or contains only one period as its last character, the value of

the variable is the value associated with VAR_SYMBOL in the variable pool, that is #Outcome after Var_ Value (Pool, VAR SYMBOL, ‘0’)

If the indicator is ‘D’, indicating the variable has the ‘dropped’ attribute, the NOVALUE condition is raised; see nnn and nnn for exceptions to this. #Response = Var Value(Pool, VAR SYMBOL, ‘0’) if left(#Response,1) == ‘D’ then call #Raise ‘NOVALUE’, VAR_SYMBOL, ’ ’ If VAR_SYMBOL contains a period which is not its last character, the value of the variable is the value associated with the derived name. 7.3.1 Derived names A derived name is derived from a VAR_-SYMBOL as follows:

VAR SYMBOL := Stem Tail

Stem := PlainSymbol ‘.’!

Tail i= (PlainSymbol | ‘.’ [PlainSymbol]) [‘.’ [PlainSymbol1]]+ PlainSymbol := (general letter | digit)+

The derived name is the concatenation of: - the Stem, without further evaluation; - the Tail, with the PlainSymbols replaced by the values of the symbols. The value of a PlainSymbol which does not start with a digit is #Outcome after Var_ Value (Pool, PlainSymbol,‘0’) These values are obtained without raising the NOVALUE condition.

If the indicator from the Var_Value was not ‘D’ then: if #Tracing.#Level == ‘I’ then call #Trace ‘>C>’

The value associated with a derived name is obtained from the variable pool, that is #Outcome after: Var_Value(Pool,Derived Name,‘1’)

If the indicator is ‘D’, indicating the variable has the ‘dropped’ attribute, the NOVALUE condition is raised; see nnn for an exception.

9.3.1 Value of a reserved symbol

The value of a reserved symbol is the value of a variable with the corresponding name in the reserved pool, see nnn.

9.4 Expressions and operators

Add a load of string coercions. Equality can operate on non-strings. What if one operand non-string?

9.4.1 The value of a term

See nnn for the syntax of a ferm.

The value of a STRING is the content of the token; see nnn.

The value of a function is the value it returns, see nnn.

If a termis a symbol or STRING then the value of the term is the value of that symbol or STRING.

If a term contains an expr_alias the value of the term is the value of the expr_-alias, see nnn.

9.4.2 The value of a prefix_expression

If the prefix_expression is a term then the value of the prefix_expression is the value of the ferm, otherwise let rhs be the value of the prefix_expression within it__ see nnn

If the prefix_expression has the form '+' prefix_expression then a check is made: if datatype(rhs)=='NUM' then call #Raise 'SYNTAX',41.3, rhs, '+'

and the value is the value of (0 + rhs).

If the prefix_expression has the form '-' prefix_expression then a check is made: if datatype(rhs)=='NUM' then call #Raise 'SYNTAX',41.3,rhs, '-'

and the value is the value of (0 - rhs).

If a prefix_expression has the form not prefix_expression then if rhs == '0' then if rhs =='1' then call #Raise 'SYNTAX', 34.6, not, rhs

See nnn for the value of the third argument to that #Raise. If the value of rhs is '0' then the value of the prefix_expression value is '1', otherwise it is '0'.

If the prefix_expression is not a term then: if #Tracing.#Level == 'I' then call #Trace '>P>'

9.4.3 The value of a power_expression

See nnn for the syntax of a power_expression.

If the power_expression is a prefix_expression then the value of the power_-expression is the value of the prefix_expression.

Otherwise, let Ihs be the value of power_expression within it, and rhs be the value of prefix_expression within it.

```
if datatype(lhs)\=='NUM' then call #Raise 'SYNTAX',41.1,lhs,'**'

if \datatype(rhs,'W') then call #Raise 'SYNTAX',26.1,rhs,'**'
```

The value of the power_expression is

ArithOp(lhs,'**',rhs)

If the power_expression is not a prefix_expression then: if #Tracing.#Level == 'I' then call #Trace '>0O>'

9.4.4 The value of a multiplication

See nnn for the syntax of a multiplication. If the multiplication is a power_-expression then the value of the multiplication is the value of the power_-expression. Otherwise, let Ihs be the value of multiplication within it, and rns

be the value of power_expression within it. if datatype(lhs)==‘NUM’ then call #Raise ‘SYNTAX’,41.1,lhs,multiplicative operation if datatype(rhs)==‘NUM’ then call #Raise ‘SYNTAX’,41.2,rhs,multiplicative operation

The value of the multiplication is ArithOp(lhs,multiplicative operation, rhs)

If the multiplication is not a power_expression then:

if #Tracing.#Level == ‘I’ then call #Trace ‘>0O>’

9.4.5 The value of an addition

See nnn for the syntax of addition.

If the addition is a multiplication then the value of the addition is the value of the multiplication. Otherwise, let Ihs be the value of ad¢difion within it, and rhs be the value of the multiplication within it. Let

operation be the adaltive_operator. if datatype(lhs)==‘NUM’ then

call #Raise ‘SYNTAX’, 41.1, lhs, operation if datatype(rhs)==‘NUM’ then

call #Raise ‘SYNTAX’, 41.2, rhs, operation

If either of rhs or Ihs is not an integer then the value of the addition is ArithOp(lhs, operation, rhs) Otherwise if the operation is ‘+’ and the length of the integer Ihs+rhs is not greater than #Digits.#Level

then the value of addition is lhs+rhs

Otherwise if the operation is ‘-’ and the length of the integer Ihs-rhs is not greater than #Digits.#Level then

the value of addition is lhs-rhs

Otherwise the value of the addition is ArithOp(lhs, operation, rhs)

If the addition is not a multiplication then: if #Tracing.#Level == ‘I’ then call #Trace ‘>0O>’

9.4.6 The value of a concatenation

See nnn for the syntax of a concatenation. If the concatenation is an addition then the value of the concatenation is the value of the addition. Otherwise, let Ihs be the value of concatenation within it, and rhs be the value of the additive_expression within it. If the concatenation contains ‘||’ then the value of the concatenation will have the following characteristics:

- Config_Length(Value) will be equal to Config_Length(Ihs)+Config_Length(rhs).
- #Outcome will be ‘equal’ after each of:
- Config_Compare(Config_Subsir(Ihs,n)},Config_Subsitr(Value,n)) for values of n not less than 1 and not more than Config_Length(Ihs);
- Config_Compare(Config_Subsir(rhs,n),Config_Substr(Value,Config_Length(Ihs)+n)) for values of n not less than 1 and not more than Config_Length(rhs). Otherwise the value of the concatenation will have the following characteristics:

- Config_Length(Value) will be equal to Config_Length(Ihs)+1+Config_-Length(rhs).
- #Outcome will be ‘equal’ after each of:
- Config_Compare(Config_Subsir(Ihs,n)},Config_Subsitr(Value,n)) for values of n not less than 1 and not more than Config_Length(Ihs);
- Config_Compare(’ ’,Config_Substr(Value,Config_Length(Ihs)}+1));
- Config_Compare(Config_Subsitr(rhs,n),Config_Substr(Value,Config_Length(Ins)+1+n)) for values of n not less than 1 and not more than Config_Length(rhs).

If the concatenation is not an addition then: if #Tracing.#Level == ‘I’ then call #Trace ‘>0O>’

9.4.7 The value of a comparison

See nnn for the syntax of a comparison.

If the comparison is a concatenation then the value of the comparison is the value of the concatenation. Otherwise, let Ihs be the value of the comparison within it, and rns be the value of the concatenation within it.

If the comparison has a comparison_operator that is a strict_compare then the variable #Test is set as follows:

#Test is set to ‘E’. Let Length be the smaller of Config_Length(Ihs) and Config_-Length(rhs). For values of n greater than O and not greater than Length, if any, in ascending order, #Test is set to the uppercased first character of #Outcome after:

Config_Compare(Config_Subsir(Ihs),Contfig_Subsir(rhs)).

If at any stage this sets #Test to a value other than ‘E’ then the setting of #Test is complete. Otherwise, if Config_Length(Ihs) is greater than Config_Length(rhs) then #Test is set to ‘G’ or if Config_Length(Ihs) is less than Config_Length(rhs) then #Test is set to ‘L’.

If the comparison has a comparison_operator that is a normal_compare then the variable #Test is set as follows:

```
if datatype(lhs)\== 'NUM' | datatype(rhs)\== 'NUM' then do
/* Non-numeric non-strict comparison */
lhs=strip(lhs, 'B', ' ') /* ExtraBlanks not stripped */
rhs=strip(rhs, 'B', ' ')

if length(lhs)>length(rhs) then rhs=left (rhs, length (lhs) )
else lhs=left (lhs, length (rhs) )
if lhs>>rhs then #Test='G'
else if lhs<<rhs then #Test='L'
else #Test='E'

end
else do /* Numeric comparison */
```

```
if left(-lhs,1) == ’-’ & left(+rhs,1) \== ’-’ then #Test=’G!
else if left(-rhs,1) == ’-’ & left(+lhs,1) \== ’-’ then #Test=’L’
else do
Difference=lhs - rhs /* Will never raise an arithmetic condition. */
if Difference > 0 then #Test=’G’
else if Difference < 0 then #Test=’L’
else #Test=’E’
end
end
The value of #Test, in conjunction with the operator in the comparison, determines the
comparison.
The value of the comparison is ’1’ if
- #Test is ’E’ and the operator is one of ’=””, ’==””, ’>=’, <=”, ‘\>’, ’\<’, ’p>=’, ’

- #Test is ’G’ and the operator is one of ’>’, ’>=”, ‘\<’, ‘\=’, ’<>’, ’><’, Nes”, ‘>>
- #Test is ’L’ and the operator is one of ’<’, <=”, \>’, \=’, ’<>’, ’><’, \==’, ’<<’,
In all other cases the value of the comparison is ’0’.

If the comparison is not a concatenation then:
if #Tracing.#Level == ’I’ then call #Trace ’>0O>’
```

9.4.8 The value of an and_expression

See nnn for the syntax of an and_expression.

If the and_expression is a comparison then the value of the and_expression is the value of the comparison.

Otherwise, let lhs be the value of the and_expression within it, and rhs be the value of the comparison within it.

if lhs == ‘0’ then if lhs == ‘1’ then call #Raise ‘SYNTAX’,34.5,lhs,‘&’

if rhs == ‘0’ then if rhs == ‘1’ then call #Raise ‘SYNTAX’,34.6,rhs,‘&’

Value=‘0’

if lhs == ‘1’ then if rhs == ‘1’ then Value=‘1’

If the and_expression is not a comparison then:

if #Tracing.#Level == ‘I’ then call #Trace ‘>0O>’

9.4.9 The value of an expression

See nnn for the syntax of an expression.

The value of an expression, or an expr, is the value of the expr_alias within it.

If the expr_alias is an and_expression then the value of the expr_alias is the value of the and_expression. Otherwise, let lhs be the value of the expr_alias within it, and rhs be the value of the and_expression

within it. if lhs == ‘0’ then if lhs == ‘1’ then

call #Raise ‘SYNTAX’,34.5,lhs,or operator if rhs == ‘0’ then if rhs == ‘1’ then

call #Raise ‘SYNTAX’,34.6,rhs,or operator Value=‘1’ if lhs == ‘0’ then if rhs == ‘0’ then Value=‘0’ If the or_operator is ‘&&’ then if lhs == ‘1’ then if rhs == ‘1’ then Value=‘0’ If the expr_alias is not an and_expression then: if #Tracing.#Level == ‘I’ then call #Trace ‘>0O>’

The value of an expression or expr shall be traced when #Tracing.#Level is ‘R’. The tag is ‘>=>’ when

the value is used by an assignment and ‘»>’ when it is not. if #Tracing.#Level == ‘R’ then call #Trace Tag

9.4.10 Arithmetic operations

The user of this standard is assumed to know the results of the binary operators ‘+’ and ‘-’ applied to signed or unsigned integers.

The code of ArithOp itself is assumed to operate under a sufficiently high setting of numeric digits to avoid exponential notation.

```
ArithoOp:

arg Numberl, Operator, Number2
/* The Operator will be applied to Numberl and Number2 under the numeric
settings #Digits.#Level, #Form.#Level, #Fuzz.#Level */

/* The result is the result of the operation, or the raising of a 'SYNTAX' or
'LOSTDIGITS' condition. */

/* Variables with digit 1 in their names refer to the first argument of the
operation. Variables with digit 2 refer to the second argument. Variables
with digit 3 refer to the result. */

/* The quotations and page numbers are from the first reference in
Annex C of this standard. */

/* The operands are prepared first. (Page 130) Function Prepare does this,
separating sign, mantissa and exponent. */

v = Prepare (Numberl, #Digits.#Level)
parse var v Signl Mantissal Exponentl

v = Prepare (Number2, #Digits.#Level)
parse var v Sign2 Mantissa2 Exponent2

/* The calculation depends on the operator. The routines set Sign3
Mantissa3 and Exponent3. */

Comparator = ''
```

```
select
when Operator == '*' then call Multiply

when Operator
when Operator

' then call DivType
*' then call Power
when Operator %' then call DivType
when Operator '//" then call DivType
otherwise call AddSubComp

end

call PostOp /* Assembles Number3 */

if Comparator \== '' then do
/* Comparison requires the result of subtraction made into a logical */
/* value. */
t = '0'
select
when left (Number3,1) == '-' then
if wordpos(Comparator,'< <= <> >< \= \>') > 0 then t = '1'
when Number3 \== '0' then
if wordpos(Comparator,'> >= <> >< \= \<') > 0 then t = '1'
otherwise
if wordpos(Comparator,'>= = =< \< \>') > 0 then t = '1!
end
Number3 = t
end
return Number3 /* From ArithOp */
/* Activity before every operation: */
Prepare: /* Returns Sign Mantissa and Exponent */
/* Preparation of operands, Page 130 */
/* ",...terms being operated upon have leading zeros removed (noting the

position of any decimal point, and leaving just one zero if all the digits in

the number are zeros) and are then truncated to DIGITS+1 significant digits
(if necessary)..." */

arg Number, Digits

/* Blanks are not significant. */

/* The exponent is separated */
```

```
parse upper value space(Number,0) with Mantissa 'E' Exponent
if Exponent == '' then Exponent = '0'

/* The sign is separated and made explicit. */

Sign = '+' /* By default */
if left(Mantissa,1) == '-' then Sign = '-'
if verify (left (Mantissa,1),'+-') = 0 then Mantissa = substr(Mantissa,2)

/* Make the decimal point implicit; remove any actual Point from the
Mantissa. */
Pp = pos('.',Mantissa)
if p > 0 then Mantissa = delstr(Mantissa,p,1)
else p = 1+length (Mantissa)

/* Drop the leading zeros */
do q = 1 to length(Mantissa) - 1

if substr(Mantissa,g,1) \== '0' then leave
p=p-l
end q

Mantissa = substr(Mantissa,q)

/* Detect if Mantissa suggests more significant digits than DIGITS
caters for. */
do j = Digits+1 to length (Mantissa)
if substr(Mantissa,j,1) \== '0' then call #Raise 'LOSTDIGITS', Number
end j

/* Combine exponent with decimal point position, Page 127 */
/* "Exponential notation means that the number includes a power of ten

following an 'E' that indicates how the decimal point will be shifted. Thus

4E9 is just a shorthand way of writing 4000000000 " */

/* Adjust the exponent so that decimal point would be at right of
the Mantissa. */
Exponent = Exponent - (length(Mantissa) - p + 1)

/* Truncate if necessary */

t = length(Mantissa) - (Digits+1)

if t > 0 then do
```

```
Exponent = Exponent + t
Mantissa = left (Mantissa,Digits+1)
end

if Mantissa == '0' then Exponent = 0

return Sign Mantissa Exponent

/* Activity after every operation. */
/* The parts of the value are composed into a single string, Number3. */
PostOp:

/* Page 130 */
/* 'traditional' rounding */
t = length(Mantissa3) - #Digits.#Level
if t > 0 then do
/* 'traditional' rounding */
Mantissa3 = left (Mantissa3,#Digits.#Level+1) + 5
if length(Mantissa3) > #Digits.#Level+1 then
/* There was 'carry' */

Exponent3 = Exponent3 + 1

Mantissa3 = left (Mantissa3,#Digits.#Level)

Exponent3 = Exponent3 + t

end
/* "A result of zero is always expressed as a single character '0' "*/
if verify (Mantissa3,'0') = 0 then Number3 = '0'
else do

if Operator == '/' | Operator == '**' then do

/* Page 130 "For division, insignificant trailing zeros are removed
after rounding." */

/* Page 133 "... insignificant trailing zeros are removed." */
do q = length(Mantissa3) by -1 to 2
if substr(Mantissa3,q,1) \== '0' then leave
Exponent3 = Exponent3 + 1
end q
Mantissa3 = substr(Mantissa3,1,q)
end
if Floating() == 'E' then do /* Exponential format */

Exponent3 = Exponent3 + (length(Mantissa3)-1)
```

```
/* Page 136 "Engineering notation causes powers of ten to be expressed as a

multiple of 3 - the integer part may therefore range from 1 through
999." */

g=l

if #Form.#Level == 'E' then do

/* Adjustment to make exponent a multiple of 3 */
g = Exponent3//3 /* Recursively using ArithOp as
an external routine. */
if g < 0 then g =g+ 3
Exponent3 = Exponent3 - g
geaqgitl
if length(Mantissa3) < g then
Mantissa3 = left (Mantissa3,g,'0')
end /* Engineering */

/* Exact check on the exponent. */
if Exponent3 > #Limit ExponentDigits then

call #Raise 'SYNTAX', 42.1, Numberl, Operator, Number2
if -#Limit ExponentDigits > Exponent3 then

call #Raise 'SYNTAX', 42.2, Numberl, Operator, Number2

/* Insert any decimal [point. */

if length(Mantissa3) \= g then Mantissa3 = insert('.',Mantissa3,g)
/* Insert the E */
if Exponent3 >= 0 then Number3
else Number3
end /* Exponent format */
else do /* 'pure number' notation */
p = length(Mantissa3) + Exponent3 /* Position of the point within
Mantissa */
/* Add extra zeros needed on the left of the point. */
if p < 1 then do
Mantissa3 = copies('0',1 - p)| |Mantissa3
p=il
end
/* Add needed zeros on the right. */
if p > length(Mantissa3) then
Mantissa3 = Mantissa3||copies('0',p-length (Mantissa3) )
/* Format with decimal point. */
```

```
Number3 = Mantissa3

Mantissa3'E+'Exponent3
Mantissa3'E'Exponent3

if p < length(Number3) then Number3 = insert('.',Mantissa3,p)
else Number3 = Mantissa3
end /* pure */
if Sign3 == '-' then Number3 = '-'Number3
end /* Non-Zero */
return
/* This tests whether exponential notation is needed. */
Floating:
/* The rule in the reference has been improved upon. */
Ct = ter
if Exponent3+length(Mantissa3) > #Digits.#Level then t = 'E'
if length(Mantissa3) + Exponent3 < -5 then t = 'E'
return t
/* Add, Subtract and Compare. */

AddSubComp: /* Page 130 */
/* This routine is used for comparisons since comparison is
defined in terms of subtraction. Page 134 */
/* "Numeric comparison is affected by subtracting the two numbers (calculating

the difference) and then comparing the result with '0'." */
NowDigits = #Digits.#Level
if Operator \=='+' & Operator \== '-' then do

Comparator = Operator

/* Page 135 "The effect of NUMERIC FUZZ is to temporarily reduce the value
of NUMERIC DIGITS by the NUMERIC FUZZ value for each numeric comparison" */
NowDigits = NowDigits - #Fuzz.#Level

end
/* Page 130 "If either number is zero then the other number ... is used as
the result (with sign adjustment as appropriate). */
if Mantissa2 == '0' then do /* Result is the lst operand */
Sign3=Signl; Mantissa3 = Mantissal; Exponent3 = Exponentl
return ''
end

if Mantissal
Sign3 = Sign

== '0' then do /* Result is the 2nd operand */
```

```
2
if Operator \

; Mantissa3 = Mantissa2; Exponent3 = Exponent2
== '+' then if Sign3 = '+' then Sign3 rat
else Sign3

bat
return ''
end

/* The numbers may need to be shifted into alignment. */

/* Change to make the exponent to reflect a decimal point on the left,

so that right truncation/extension of mantissa doesn't alter exponent. */
Exponentl = Exponentl + length (Mantissal1)
Exponent2 = Exponent2 + length (Mantissa2)

/* Deduce the implied zeros on the left to provide alignment. */

Aligni = 0

Align2 = Exponentl - Exponent2

if Align2 > 0 then do /* Arg 1 provides a more significant digit */
Align2 = min(Align2,NowDigits+1) /* No point in shifting further. */
/* Shift to give Arg2 the same exponent as Argl */

Mantissa2 = copies('0',Align2) || Mantissa2
Exponent2 = Exponentl
end

if Align2 < 0 then do /* Arg 2 provides a more significant digit */
/* Shift to give Argl the same exponent as Arg2 */

Aligni = -Align2

Alignl = min(Align1l,NowDigits+1) /* No point in shifting further. */
Align2 = 0

Mantissal = copies('0',Alignl) || Mantissal

Exponentl = Exponent2

end
```

```
/* Maximum working digits is NowDigits+1. Footnote 41. */

SigDigits
SigDigits

max (length (Mantissal) , length (Mantissaz2) )
min (SigDigits,NowDigits+1)

/* Extend a mantissa with right zeros, if necessary. */
Mantissal = left (Mantissal,SigDigits,'0')

Mantissa2 = left (Mantissa2,SigDigits,'0')

/* The exponents are adjusted so that

the working numbers are integers, ie decimal point on the right. */

Exponent3 = Exponentl-SigDigits
Exponentl = Exponent3
Exponent2 = Exponent3
if Operator = '+' then
Mantissa3 = (Signl || Mantissal) + (Sign2 || Mantissa2)

else Mantissa3 (Signl || Mantigsal) - (Sign2 || Mantissa2)

/* Separate the sign */
if Mantissa3 < 0 then do

Sign3 = '-'
Mantissa3 = substr (Mantissa3,2)
end

else Sign3 = '+'

/* "The result is then rounded to NUMERIC DIGITS digits if necessary,
taking into account any extra (carry) digit on the left after addition,
but otherwise counting from the position corresponding to the most
significant digit of the terms being added or subtracted." */

if length(Mantissa3) > SigDigits then SigDigits = SigDigits+l
d = SigDigits - NowDigits /* Digits to drop. */
if d <= 0 then return
t = length(Mantissa3) - d /* Digits to keep. */
/* Page 130. "values of 5 through 9 are rounded up, values of 0 through 4 are
rounded down." */
if t > 0 then do
/* 'traditional' rounding */
```

```
Mantissa3 = left(Mantissa3, t +1) +5
if length(Mantissa3) > t+1 then
/* There was 'carry' */
/* Keep the extra digit unless it takes us over the limit. */
if t < NowDigits then t = t+l
else Exponent3 = Exponent3+1
Mantissa3 = left (Mantissa3,t)
Exponent3 = Exponent3 + d
end /* Rounding */
else Mantissa3 = '0'
return /* From AddSubComp */

/* Multiply operation: */

Multiply: /* p 131 */

/* Note the sign of the result */

if Signl == Sign2 then Sign3 = '+'
else Sign3 = '-'
/* Note the exponent */
Exponent3 = Exponentl + Exponent2
if Mantissal == '0' then do
Mantissa3 = '0'
return
end
/* Multiply the Mantissas */
Mantissa3 = ''

do q=1 to length (Mantissa2)
Mantissa3 = Mantissa3'0'
do substr(Mantissa2,q,1)
Mantissa3 = Mantissa3 + Mantissal
end
end q
return /* From Multiply */

/* Types of Division: */

DivType: /* p 131 */
/* Check for divide-by-zero */

if Mantissa2 == '0' then call #Raise 'SYNTAX',

/* Note the exponent of the result */
Exponent3 = Exponentl - Exponent2
```

```
/* Compute (one less than) how many digits will be in the integer

part of the result. */

IntDigits = length(Mantissal) - Length(Mantissa2) + Exponent3
/* In some cases, the result is known to be zero.
if Mantigsal = 0 | (IntDigits < 0 & Operator

Mantissa3 = 0
Sign3 = '+'
Exponent3 = 0
return
end
/* In some cases, the result is known to be to be the first argument.
if IntDigits < 0 & Operator == '//' then do
Mantissa3 = Mantissal
Sign3 = Signl
Exponent3 = Exponentl
return
end
/* Note the sign of the result. */
if Signl == Sign2 then Sign3 = '+'
else Sign3 = '-'

/* Make Mantissal at least as large as Mantissa2 so Mantissa2 can be
subtracted without causing leading zero to result. Page 131 */

az 0
do while Mantissa2 > Mantissal

Mantissal = Mantissal'0'
Exponent3 = Exponent3 - 1
aztadtl
end
/* Traditional divide */
Mantissa3 = ''

/* Subtract from part of Mantissal that has length of Mantissa2 */

left (Mantissal,length(Mantissa2) )
substr (Mantissal, length (Mantissa2)+1)
o forever

x
Y
d
```

```
/* Develop a single digit in z by repeated subtraction.

ze=O0
do forever
xX = kK - Mantissa2
if left(x,1) == '-' then leave
Zeze¢t+tl
end

x = x + Mantissa2 /* Recover from over-subtraction */
/* The digit becomes part of the result */

Mantissa3 = Mantissa3 || z

if Mantissa3 == '0' then Mantissa3 = '' /* A single leading


##

*f
'%') then do

*/
zero can happen. */
/* x||y is the current residue */
if y == '' then if x = 0 then leave /* Remainder is zero */

if length(Mantissa3) > #Digits.#Level then leave /* Enough digits
in the result */

/* Check type of division */
if Operator \== '/' then do
if IntDigits = 0 then leave
IntDigits = IntDigits - 1
end
/* Prepare for next digit */
/* Digits come from y, until that is exhausted. */
/* When y is exhausted an extra zero is added to Mantissal */
if y == '' then do
y = ror
Exponent3 = Exponent3 - 1
aztadtl
end
xX = xX | | left (y,1)
y = substr(y,2)
end /* Iterate for next digit. */
```

```
Remainder = x || y
Exponent3 = Exponent3 + length(y) /* The loop may have been left early.
/* Leading zeros are taken off the Remainder. */
do while length(Remainder) > 1 & Left (Remainder,1) == '0'
Remainder = substr (Remainder, 2)
end
if Operator \== '/' then do
/* Check whether % would fail, even if operation is // */

if Floating() 'E' then do
if Operator '%' then MsgNum
else MsgNum

/* Page 133. % could fail by needing exponential notation */

26.11
26.12

call #Raise 'SYNTAX', MsgNum, Numberl , Number2, #Digits.#Level

end
end
if Operator == '//' then do
/* We need the remainder */
Sign3 = Signl

Mantissa3 = Remainder
Exponent3 = Exponentl - a
end

return /* From DivType */

/* The Power operation: */

Power: /* page 132 */
/* The second argument should be an integer */
if \WholeNumber2() then call #Raise 'SYNTAX', 26.8, Number2
/* Lhs to power zero is always 1 */
if Mantissa2 == '0' then do
Sign3 = '+'
Mantissa3
Exponent3
return
end

i
ror
```

```
/* Pages 132-133 The Power algorithm */
Rhs = left (Mantissa2,length(Mantissa2)+Exponent2,'0')/* Explicit
integer form */
L length (Rhs)
b X2B(D2X(Rhs)) /* Makes Rhs in binary notation */
/* Ignore initial zeros */
do q=l1byl
if substr(b,q,1) \== '0' then leave
end q
ael
do forever
/* Page 133 "Using a precision of DIGITS+L+1" */
if substr(b,q,1) == '1' then do
a = Recursion('*',Signl || Mantissal'E'Exponent1)


*/
if left(a,2) == 'MN' then signal PowerFailed
end

/* Check for finished */

if q = length(b) then leave

/* Square a */

a = Recursion('*',a)
if left(a,2) == 'MN' then signal PowerFailed
q=qrtil
end
/* Divide into one for negative power */
if Sign2 == '-' then do
Sign2 = '+'
a = Recursion('/')
if left(a,2) == 'MN' then signal PowerFailed
end

/* Split the value up so that PostOp can put it together with rounding */
Parse value Prepare(a,#Digits.#Level+L+1) with Sign3 Mantissa3 Exponent3
return

PowerFailed:
/* Distinquish overflow and underflow */
ReWas = substr(a,4)
if Sign2 = '-' then if ReWas == '42.1' then RcWas
else RcWas
```

```
call #Raise 'SYNTAX', RcWas, Number1, '**', Number2
/* No return */

"42.2!
"42.1!

WholeNumber2:
numeric digits Digits
if #Form.#Level == 'S' then numeric form scientific

else numeric form engineering
return datatype (Number2, 'W')

Recursion: /* Called only from '**! */
numeric digits #Digits.#Level + L + 1
signal on syntax name Overflowed
/* Uses ArithOp again under new numeric settings. */
if arg(1) == '/' then return 1l/a
else return a * arg(2)
Over flowed:
return 'MN '.MN
```

9.5 Functions

9.5.1 Invocation

Invocation occurs when a function or a message_term or a callis evaluated. Invocation of a function may result in a value, in which case:

if #Tracing.#Level == 'I' then call #Trace '>F>' Invocation of a message_term may result in a value, in which case: if #Tracing.#Level == 'I' then call #Trace '>M>'

9.5.2 Evaluation of arguments

The argument positions are the positions in the exoression_list where syntactically an expression occurs or could have occurred. Let ArgNumber be the number of an argument position, counting from 1 at the left; the range of ArgNumber is all whole numbers greater than zero.

For each value of ArgNumber, #ArgExists.#NewLevel.ArgNumber is set '1' if there is an expression present, 'O' if not.

From the left, if #ArgExists.#NewLevel.ArgNumber is '1' then #Arg.#NewLevel.ArgNumber is set to the value of the corresponding expression. If #ArgExists.#NewLevel.ArgNumber is '0' then #Arg.#NewLevel.ArgNumber is set to the null string.

#ArgExists.#NewLevel.0 is set to the largest ArgNumber for which #ArgExists.#NewLevel.ArgNum is ‘1’, or to zero if there is no such value of ArgNumber.

9.5.3 The value of a label

The value of a LABEL, or of the taken_constant in the function or call_instruction, is taken as a constant, see nnn. If the taken_constant is not a string_literal it is a reference to the first LABEL in the program which has the same value. The comparison is made with the ‘==’ operator.

If there is such a matching label and the label is trace-only (see nnn) then a condition is raised: call #Raise ‘SYNTAX’, 16.3, taken constant

If there is such a matching label, and the label is not trace-only, execution continues at the label with routine initialization (see nnn). This is execution of an internal routine.

If there is no such matching label, or if the taken_constant is a string_literal, further comparisons are made.

If the value of the taken_constant matches the name of some built-in function then that built-in function is invoked. The names of the built-in functions are defined in section nnn and are in uppercase.

If the value does not match any built-in function name, Config_ExternalRoutine is used to invoke an external routine.

Whenever a matching label is found, the variables SIGL and .SIGL are assigned the value of the line number of the clause which caused the search for the label. In the case of an invocation resulting from a

condition occurring that shall be the clause in which the condition occurred. Var _ Set(#Pool, ‘SIGL’, ‘0’, #LineNumber) var Set(0 , ‘.SIGL’, ‘0’, #LineNumber)

The name used in the invocation is held in #Name.#Level for possible use in an error message from the RETURN clause, see nnn

9.5.4 The value of a function

A built-in function completes when it returns from the activity defined in section nnn. The value of a built-in function is defined in section nnn.

An internal routine completes when #Level returns to the value it had when the routine was invoked. The value of the internal function is the value of the expression on the return which completed the routine. The value of an external function is determined by Config_ExternalRoutine.

9.5.5 The value of a method

A built-in method completes when it returns from the activity defined in section n. The value of a built-in method is defined in section n.

An internal method completes when #Level returns to the value it had when the routine was invoked. The value of the internal method is the value of the expression on the return which completed the method. The value of an external method is determined by Config_ExternalMethod.

9.5.6 The value of a message term

See nnn for the syntax of a message_term. The value of the ferm within a message_term is called the receiver.

The receiver and any arguments of the term are evaluated, in left to right order. r= #evaluate(message term, term) If the message term contains '~~' the value of the message term is the receiver. Any effect on .Result? Otherwise the value of a message_term is the value of the method it invokes. The method invoked is determined by the receiver and the taken_constant and symbol. t = #Instance(message term, taken constant) If there is a symbol, it is subject to a constraints. if #contains (message term, symbol) then do if r <> #Self then

call #Raise 'SYNTAX', nn.n

/* OOI: "Message search overrides can only be used from methods of the target object." */

The search will progress from the object to its class and superclasses. /* This is going to be circular because it describes the message lookup algorithm and also uses messages. However for the messages in this code the message names are chosen to be unique to a method so there is no need to use this algorithm in deciding which method is intended. */

/* message term ::= receiver '~' taken constant ':' VAR_SYMBOL arguments */

/* This code reflects OOI - the arguments on the message don't affect the method choice. */

/* This code selects a method based on its arguments, receiver, taken_constant, and symbol. */

/* This code is used in a context where #Self is the receiver of the method invocation which the subject message term is running under. */

```
SelectMethod:

/* If symbol given, receiver must be self. */
if arg(3,'E') then if arg(1)\==#Self then signal error /* syntax number? */

t arg(2) /* Will have been uppercased, unless a literal. */

x arg(1) /* Cursor through places to look for the method. */

Mixing 1 /* Off for potential mixins ignored because symbol given. */
Mixins -array~new /* to note any Mixins involved. */
```

```
/* Look in the method table of the object, if no 'symbol' given. */
if arg(3,'E') then do
Mixing = 0

end
else do
m = x~#MethodTable[t]
if m \== .nil then return m
end

do until x==.object
/* Follow the class hierarchy. */
x = x-class
/* Note any mixins for later reference. */
Mix = x~Inherited /* An array, ordered as the directive left-to-right. */

if Mix \== .nil then /* Append to the record. */
do j=1 to Mix~dimension (1)
Mixins [Mixins~dimension(1)+1] = Mix[j]
end

if Mixing do
/* Consider mixins only for superclasses of 'symbol'. */
do j=1 to Mixins~dimension (1)
/* Look at the baseclass of each. */
/* That is closest superclass not a mixin. */
s = Mixins[j]~class
do while s~Mixin /* Assert stop at .object if not before. */
s=s~class
end
if s==x then do
m=Mixins [j]~#MethodTable[t]
if m \== .nil then return m
end
end j
end /* Mixing */
if arg(3,'E') then if arg(3)==x then do
Mixing=1
end
if Mixing do
/* Consider non-Mixins */
m= x-#InstanceMethodTable[t]

if m \== .nil then return m
end

x=x~superclass
```

```
end

/* Try for UNKNOWN instead */
if t == 'UNKNOWN' then return .nil
if \arg(3,'E') then return SelectMethod arg(1),'UNKNOWN'
else return SelectMethod arg(1),'UNKNOWN',arg(3)
```

9.5.7 Use of Config_ExternalRoutine

The values of the arguments to the use of Config_ExternalRoutine, in order, are:

The argument How is 'SUBROUTINE' if the invocation is from a call, "'FUNCTION' if the invocation is from a function.

The argument NameType is '1' if the taken_constant is a string_literal, '0' otherwise.

The argument Name is the value of the faken_constant.

The argument Environment is the value of this argument on the API_ Start which started this execution. The argument Arguments is the #Arg. and #ArgExists. data.

The argument Streams is the value of this argument on the API_Start which started this execution.

The argument Traps is the value of this argument on the API_Start which started this execution. Var_Reset is invoked and #API_Enabled set to '1' before use of Config_ExternalRoutine. #API_Enabled is set to 'O' after.

The response from Config_ExternalRoutine is processed. If no conditions are (implicitly) raised, #Outcome is the value of the function.

10

Directives

The syntax constructs which are introduced by the optional '::' token are known as directives.

10.1 Notation

Notation functions are functions which are not directly accessible as functions in a program but are used in this standard as a notation for defining semantics.

Some notation functions allow reference to syntax constructs defined in nnn. Which instance of the syntax construct in the program is being referred to is implied; it is the one for which the semantics are being specified.

The BNF_primary referenced may be directly in the production or in some component referenced in the

production, recursively. The components are considered in left to right order. #Contains (Identifier, BNF primary)

where:

Identifier is an identifier in a production (see nnn) defined in nnn.

BNF_primary is a bnf_primary (see nnn) in a production defined in nnn. Return '1' if the production identitied by identifier contained a bnf_primary identified by BNF_primary, otherwise return '0'.

#Instance (Identifier, BNF primary) where: Identifier is an identifier in a production defined in nnn. BNF_primary is a Onf_primary in a production defined in nnn. Returns the content of the particular instance of the BNF_primary. If the BNF_-primary is a VAR_SYMBOL this is referred to as the symbol "taken as a constant."

#Evaluate (Identifier, BNF primary) where: Identifier is an identifier in a production defined in nnn. BNF_primary is a Onf_primary in a production defined in nnn. Return the value of the BNF_primary in the production identified by Identifier.

#Execute (Identifier, BNF primary) where: Identifier is an identifier in a production defined in nnn. BNF_primary is a Onf_primary in a production defined in nnn. Perform the instructions identified by the BNF_primary in the production identified by Identifier.

#Parses (Value, BNF primary) where: Value is a string BNF_primary is a Onf_-primary in a production defined in nnn. Return '1' if Value matches the definition of the BNF_primary, by the rules of clause 6, '0' otherwise.

#Clause (Label) where: Label is a label in code used by this standard to describe processing. Return an identification of that label. The value of this identification is used only by the #Goto notation function.

#Goto (Value) where:

Value identifies a label in code used by this standard to describe processing. The description of processing continues at the identified label.

#Retry ()

This notation is used in the description of interactive tracing to specify re-execution of the clause just previously executed. It has the effect of transferring execution to the beginning of that clause, with state variable #Loop set to the value it had when that clause was previously executed.

10.2 Initializing

Some of the initializing, now grouped in classic section 8.2.1 will have to come here so that we have picked up anything from the START_API that needs to be passed on to the execution of REQUIRES subject.

We will be using some operations that are forward reference to what was section nnn.

10.2.1 Program initialization and message texts

Processing of a program begins when API_Start is executed. A pool becomes current for the reserved

variables. call Config ObjectNew #ReservedPool = #Outcome #Pool = #ReservedPool Is it correct to make the reserved variables and the builtin objects in the same pool?

Some of the values which affect processing of the program are parameters of API_ Start:

#HowInvoked is set to ‘COMMAND’, ‘FUNCTION’ or ‘SUBROUTINE’ according to the first parameter of APL Start.

#Source is set to the value of the second parameter of API_ Start.

The third parameter of API_Start is used to determine the initial active environment.

The fourth parameter of API Start is used to determine the arguments. For each argument position #ArgExists.1.ArgNumber is set ‘1’ if there is an argument present, ‘0’ if not. ArgNumber is the number of the argument position, counting from 1. If #ArgExists.1.ArgNumber is ‘1’ then #Arg.1.ArgNumber is set to the value of the corresponding argument. If #ArgExists.1.ArgNumber is ‘0’ then #Arg.1.Arg is set to the null string. #ArgExists.1.0 is set to the largest n for which #ArgExists.1.n is ‘1’, or to zero if there is no such value of n.

Some of the values which affect processing of the program are provided by the

configuration: call Config OtherBlankCharacters

#A11Blanks<Index “#Al1Blanks” # “ ” > = ’ ’#Outcome /* “Real” blank concatenated with others */

#Bif Digits. = 9

call Config Constants

-true = ‘1’

-false = ‘0O’

Objects in our model are only distinquished by the values within their pool so we can construct the builtin classes incomplete and then complete them with directives.

Can we initialize the methods of .nil by directives?

call Config ObjectNew

-List = #Outcome

call var_set .List, #IsClass, ‘0’, ‘1’

call var_set .List, #ID, ‘0’, ‘List’

Some of the state variables set by this call are limits, and appear in the text of error messages. The relation between message numbers and message text is defined by the following list, where the message

number appears immediately before an ‘=’ and the message text follows in quotes.

#ErrorText. stl

#ErrorText.0.1 = ‘Error running , line :’

#ErrorText.0.2 = ‘Error in interactive trace:’

#ErrorText.0.3 = ‘Interactive trace. “Trace Off” to end debug.’, “ENTER to continue.’

#ErrorText.2 = ‘Failure during finalization’

#ErrorText.2.1 = ‘Failure during finalization: ’

#ErrorText.3 #ErrorText.3.1

‘Failure during initialization’ ‘Failure during initialization: ’

#ErrorText.4 #ErrorText.4.1

‘Program interrupted’ ‘Program interrupted with HALT condition: ’

#ErrorText.5 #ErrorText.5.1

‘System resources exhausted’ ‘System resources exhausted: ’

#ErrorText.6 #ErrorText.6.1

’Unmatched “/*” or quote! ’Unmatched comment delimiter (”/*”)!

#ErrorText.6.2 #ErrorText.6.3

#ErrorText.7 = #ErrorText.7.1 =

#ErrorText.7.2 =

#ErrorText.7.3 =

#ErrorText.8 = #ErrorText.8.1 #ErrorText.8.2

#ErrorText.9 = #ErrorText.9.1 #ErrorText.9.2

#ErrorText.10 #ErrorText.10.1= #ErrorText.10.2=

#ErrorText.10.3=

#ErrorText.10.4= #ErrorText.10.5= #ErrorText.10.6=

#ErrorText.13 = #ErrorText.13.1=

#ErrorText.14 = #ErrorText.14.1= #ErrorText.14.2= #ErrorText.14.3= #ErrorText.14.4=

#ErrorText.15 = #ErrorText.15.1=

#ErrorText.15.2= #ErrorText.15.3= #ErrorText.15.4= #ErrorText.16 = #ErrorText.16.1= #ErrorText.16.2= #ErrorText.16.3=

#ErrorText.17 = #ErrorText.17.1=

#ErrorText.17.2=

#ErrorText.18 =

#ErrorText.18.1=

#ErrorText.18.2=

“Unmatched single quote (‘)” ‘Unmatched double quote (“)’

‘WHEN or OTHERWISE expected’ “SELECT on line ‘found “” ’

“SELECT on line ‘or END; found “” ’!

requires WHEN;’,

‘All WHEN expressions of SELECT on line are’,

‘false; OTHERWISE expected’

“Unexpected THEN or ELSE’

‘THEN has no corresponding IF or WHEN clause’ ‘ELSE has no corresponding THEN clause’

“Unexpected WHEN or OTHERWISE’ ‘WHEN has no corresponding SELECT’ ‘OTHERWISE has no corresponding SELECT’

‘Unexpected or unmatched END’

‘END has no corresponding DO or SELECT’

‘END corresponding to DO on line ’, ‘must have a symbol following that matches’, ‘the control variable (or no symbol);’,

‘found “” ’!

‘END corresponding to DO on line ’, ‘must not have a symbol following it because’, ‘there is no control variable;’,

‘found “” ’!

‘END corresponding to SELECT on line ’, ‘must not have a symbol following;’,

‘found “” ’!

‘END must not immediately follow THEN’

‘END must not immediately follow ELSE’

‘Invalid character in program’ ‘Invalid character “(’‘X)” in program’

“Incomplete DO/SELECT/IF’

‘DO instruction requires a matching END’ ‘SELECT instruction requires a matching END’ ‘THEN requires a following instruction’ ‘ELSE requires a following instruction’

‘Invalid hexadecimal or binary string’

‘Invalid location of blank in position’, ‘ in hexadecimal string’

‘Invalid location of blank in position’, ‘ in binary string’

‘Only 0-9, a-f, A-F, and blank are valid in a’, “hexadecimal string; found”“ ’

‘Only 0, 1, and blank are valid in a’,

‘binary string; found “” ’

‘Label not found’

‘Label “” not found’

‘Cannot SIGNAL to label “” because it is’, ‘inside an IF, SELECT or DO group’

‘Cannot invoke label “” because it is’, ‘inside an IF, SELECT or DO group’

‘Unexpected PROCEDURE’

“PROCEDURE is valid only when it is the first’, ‘instruction executed after an internal CALL’, ‘or function invocation’

‘The EXPOSE ‘instruction executed after a method invocation!’ ‘THEN expected’

‘IF keyword on line requires’, ‘matching THEN clause; found “” ’

‘WHEN keyword on line requires’,

requires WHEN, OTHERWISE,’,

instruction is valid only when it is the first’, #ErrorText.19 = #ErrorText.19.1=

#ErrorText.19.2= #ErrorText.19.3= #ErrorText.19.4= #ErrorText.19.6= #ErrorText.19.7= #ErrorText.19.8=

#ErrorText.19.9=

#ErrorText.19.11=’String or symbol

#ErrorText.19.12=’String or symbol

#ErrorText.19.13=

#ErrorText.19.15=’String or symbol

#ErrorText.19.16=’String or symbol

#ErrorText.19.17=

Unsound now we are using “term’?

65

#ErrorText.20 = #ErrorText.20.1= #ErrorText.20.2= #ErrorText.20.3=

#ErrorText.21 #ErrorText.21.1

#ErrorText.22 = #ErrorText.22.1=

#ErrorText.23 = #ErrorText.23.1=

#ErrorText.24 = #ErrorText.24.1= #ErrorText.25 = #ErrorText.25.1= #ErrorText.25.2= #ErrorText.25.3= #ErrorText.25.4= #ErrorText.25.5= #ErrorText.25.6= #ErrorText.25.7=

#ErrorText.25.8=

‘matching THEN clause; found “” ’ ‘String or symbol expected’ ‘String or symbol expected after ADDRESS keyword;’, “found”“! ‘String or symbol expected after CALL keyword;’,”found “”! ‘String or symbol expected after NAME keyword;’, “found”“! ‘String or symbol expected after SIGNAL keyword;’,”found “”! ‘String or symbol expected after TRACE keyword;’, “found”“! ‘Symbol expected in parsing pattern;’,”found “”! ‘String or symbol expected after REQUIRES;’, “found”“! ‘String or symbol expected after METHOD;’,”found “”!

expected after ROUTINE;’, “found”“!

expected after CLASS;’, “found”“! ‘String or symbol expected after INHERIT;’,”found “”!

expected after METACLASS;’, “found”“!

expected after MIXINCLASS;‘, “found”“! ‘String or symbol expected after SUBCLASS;’,”found “”! ’Name expected’ ‘Name required; found “” ’

‘Found “” where only a name is valid’

’Found “” where only a name or

‘Invalid data on end of clause’ ‘The clause ended at an unexpected token;’,

‘found “” ’

’Invalid “Invalid

’Invalid “Invalid

’Invalid

character string’ character string

data string’ data string

TRACE request’

is valid’

”xX”

‘’! xX”

‘TRACE request letter must be one of’,

’ “ACEFILNOR”;

found

“evalue>”’

‘Invalid sub-keyword found’ ‘CALL ON must be followed by one of the’,

’keywords ;

found

“etoken>”!

‘CALL OFF must be followed by one of the’,

’keywords ;

found

“etoken>”!

“SIGNAL ON must be followed by one of the’,

’keywords ;

found

“etoken>”!

‘SIGNAL OFF must be followed by one of the’,

’keywords ;

found

“etoken>”!

“ADDRESS WITH must be followed by one of the’,

’keywords ;

found

“etoken>”!

“INPUT must be followed by one of the’,

’keywords ; ’OUTPUT must be followed by ’keywords ; “APPEND must be followed by ’keywords ;

found

found

found

“etoken>”! one of the’, “etoken>”! one of the’, “etoken>”!

#ErrorText.25.9=

#ErrorText.25.11= #ErrorText.25.12= #ErrorText.25.13= #ErrorText.25.14= #ErrorText.25.15= #ErrorText.25.16= #ErrorText.25.17=

#ErrorText.25.18=

#ErrorText.26 #ErrorText.26.1=

#ErrorText.26.2=

#ErrorText.26.3=

#ErrorText.26.4= #ErrorText.26.5=

#ErrorText.26.6=

#ErrorText.26.7= #ErrorText.26.8=

#ErrorText.26.11

#ErrorText.26.12

#ErrorText.27 #ErrorText.27.1=

#ErrorText.28 #ErrorText.28.1= #ErrorText.28.2= #ErrorText.28.3=

#ErrorText.28.4=

#ErrorText.29 #ErrorText.29.1=

#ErrorText.30 #ErrorText.30.1= #ErrorText.30.2=

#ErrorText.31 #ErrorText.31.1=

#ErrorText.31.2=

#ErrorText.31.3=

“REPLACE must be followed by one of the’, ‘keywords ; found “”’ ”NUMERIC FORM must be followed by one of the’, ‘keywords ; found “”’

“PARSE must be followed by one of the’, ‘keywords ; found “”’

“UPPER must be followed by one of the’, ‘keywords ; found “”’

“ERROR must be followed by one of the’, ‘keywords ; found “”’ ”NUMERIC must be followed by one of the’, ‘keywords ; found “”’ ‘FOREVER must be followed by one of the’, “keywords or nothing; found”“”’PROCEDURE must be followed by the keyword’, “EXPOSE or nothing; found”“ ’

“FORWARD must be followed by one of the the keywords’,

‘; found “”’

‘Invalid whole number’

‘Whole numbers must fit within current DIGITS’, ‘setting(); found “”’

‘Value of repetition count expression in DO instruction’,

‘must be zero or a positive whole number;’, ‘found “”’

‘Value of FOR expression in DO instruction’, ‘must be zero or a positive whole number;’, ‘found “”’

“Positional pattern of parsing template’, ‘must be a whole number; found “”’ ”NUMERIC DIGITS value’,

‘must be a positive whole number; “NUMERIC FUZZ value’,

‘must be zero or a positive whole number;’,

‘found “”’

‘Number used in TRACE setting’,

‘must be a whole number; found “”’

’Operand to right of the power operator (“**“)‘, ’must be a whole number; found “”’

“Result of % operation would need’,

found “”’

‘exponential notation at current NUMERIC DIGITS ’

“Result of % operation used for // ‘, ‘operation would need’,

‘exponential notation at current NUMERIC DIGITS ’

‘Invalid DO syntax’ ‘Invalid use of keyword “” in DO clause’

‘Invalid LEAVE or ITERATE’

‘LEAVE is valid only within a repetitive DO loop’ ‘ITERATE is valid only within a

repetitive DO loop’ ‘Symbol following LEAVE (“”) must’,

‘either match control variable of a current’,

‘DO loop or be omitted’

‘Symbol following ITERATE (“”) must’, ‘either match control variable of a current’,

‘DO loop or be omitted’

‘Environment name too long’ ‘Environment name exceeds’,

#Limit EnvironmentName ‘characters; found “”’ ‘Name or string too long’

‘Name exceeds’ #Limit Name ‘characters’ “Literal string exceeds’ #Limit Literal ‘characters’ ‘Name starts with number or “.”’!

‘A value cannot be assigned to a number;’,

‘found “”’

‘Variable symbol must not start with a number;’, ‘found “”’

‘Variable symbol must not start with a “.”;’,

‘found “”’ #ErrorText.33 = #ErrorText.33.1=

#ErrorText.33.2=

#ErrorText.33.3=

#ErrorText.34 = #ErrorText.34.1=

‘Invalid expression result’ ‘Value of NUMERIC DIGITS (“”)’,

’must exceed value of NUMERIC FUZZ

“(<¢value>)”’!

‘Value of NUMERIC DIGITS (“”)’,

‘must not exceed’

#Limit Digits

“Result of expression following NUMERIC FORM’,

’must start with “E”

or “S”; found “”’

“Logical value not”0” or “1”!

’Value of expression

following IF keyword’,

‘must be exactly “0” or “1”; found “”’ #ErrorText.34.2= ‘Value of expression following WHEN keyword’,

‘must be exactly “0” or “1”; found “”’ #ErrorText.34.3= ‘Value of expression following WHILE keyword’,

‘must be exactly “0” or “1”; found “”’ #ErrorText.34.4= ‘Value of expression following UNTIL keyword’,

‘must be exactly “0” or “1”; found “”’ #ErrorText.34.5= ‘Value of expression to left’,

‘of logical operator “”’,

‘must be exactly “0” or “1”; found “” ’ #ErrorText.34.6= ‘Value of expression to right’,

‘of logical operator “” ’,

‘must be exactly “0” or “1”; found “” ’ #ErrorText.35 = ‘Invalid expression’ #ErrorText.35.1= ‘Invalid expression detected at “” ’

#ErrorText.36

‘Unmatched “(” in expression’

#ErrorText.37 = #ErrorText.37.1= #ErrorText.37.2=

’Unexpected n a n ny nt

‘Unexpected “,”! ‘Unmatched “)” in expression’

or

#ErrorText.38 = #ErrorText.38.1= #ErrorText.38.2= #ErrorText.38.3=

‘Invalid template or pattern’

‘Invalid parsing template detected at “” ’ ‘Invalid parsing position detected at “” ’ “PARSE VALUE instruction requires WITH keyword’

“Incorrect call to routine’

‘External routine “” failed’

‘Not enough arguments in invocation of ;’, ‘minimum expected is ’

‘Too many arguments in invocation of ;’, ‘maximum expected is ’

‘Missing argument in invocation of ;’, ‘argument is required’ ‘ebif> argument ’,

‘exponent exceeds’ #Limit ExponentDigits “found”“ ’

‘ argument ’,

‘must be a number; found “” ’! ‘ argument ’,

#ErrorText.40 = #ErrorText.40.1= #ErrorText.40.3= #ErrorText.40.4= #ErrorText.40.5=

#ErrorText.40.9= ‘digits;’,

#ErrorText.40.11=

#ErrorText.40.12=

‘must be a whole number; found “” ’ #ErrorText.40.13=‘ argument ’,

‘must be zero or positive; found “” ’ #ErrorText.40.14=‘ argument ’,

‘must be positive; found “” ’

#ErrorText.40.17=‘ argument 1’, ‘must have an integer part in the range 0:90 and a’, ‘decimal part no larger than .9; found “” ’ #ErrorText.40.18=‘ conversion must’,

“have a year in the range 0001 to 9999! #ErrorText.40.19=‘ argument 2, “”, is not in the format’,

‘described by argument 3, “” ’

#ErrorText.40.21=‘ argument must not be null’ #ErrorText.40.23=‘ argument ’,

‘must be a single character; found “” ’ #ErrorText.40.24=‘ argument 1’,

‘must be a binary string; found “” ’ #ErrorText.40.25=‘ argument 1’,

‘must be a hexadecimal string; found “” ’ #ErrorText.40.26=‘ argument 1’,

‘must be a valid symbol; found “” ’ #ErrorText.40.27=‘ argument 1’,

‘must be a valid stream name; found “” ’ #ErrorText.40.28=‘ argument ,’,

‘option must start with one of “”;’,

‘found “” ’

#ErrorText.40.29=‘ conversion to format “” is not allowed’ #ErrorText.40.31=‘ argument 1 (“”) must not exceed 100000’ #ErrorText.40.32=‘ the difference between argument 1 (“”) and’,

‘argument 2 (“”) must not exceed 100000’ #ErrorText.40.33=‘ argument 1 (“”) must be less than’,

‘or equal to argument 2 (“”)’ #ErrorText.40.34=‘ argument 1 (“”) must be less than’,

‘or equal to the number of lines’,

‘in the program (<sourceline()>)’ #ErrorText.40.35=‘ argument 1 cannot be expressed as a whole number;’,

‘found “” ’

#ErrorText.40.36=‘ argument 1’, ‘must be the name of a variable in the pool;’, ‘found “” ’

#ErrorText.40.37=‘ argument 3’,

‘must be the name of a pool; found “” ’ #ErrorText.40.38=‘ argument ’,

‘is not large enough to format “” ’ #ErrorText.40.39=‘ argument 3 is not zero or one; found “” ’ #ErrorText.40.41=‘ argument ’,

‘must be within the bounds of the stream;’,

‘found “” ’

#ErrorText.40.42=‘ argument 1; cannot position on this stream;’, ‘found “” ’ #ErrorText.40.45=‘ argument must be a single’,

‘non-alphanumeric character or the null string;’,

’ “found ”’

#ErrorText.40.46=‘ argument 3, “”, is a format incompatible’,

‘with separator specified in argument ’

#ErrorText.41 = ‘Bad arithmetic conversion’ #ErrorText.41.1= ‘Non-numeric value (“”)’, ‘to left of arithmetic operation “” ’

#ErrorText.41.2= ‘Non-numeric value (“”)’, ‘to right of arithmetic operation “” ’ #ErrorText.41.3= ‘Non-numeric value (“”)’,

‘used with prefix operator “” ’

#ErrorText.41.4= ‘Value of TO expression in DO instruction’, ‘must be numeric; found “” ’!

#ErrorText.41.5= ‘Value of BY expression in DO instruction’, ‘must be numeric;

found “” ’!

#ErrorText.41.6= ‘Value of control variable expression of DO instruction’, ‘must be numeric; found “” ’!

#ErrorText.41.7= ‘Exponent exceeds’ #Limit ExponentDigits ‘digits;’, ‘found “” ’

#ErrorText.42 = ‘Arithmetic overflow/underflow’ #ErrorText.42.1l= ‘Arithmetic overflow detected at’, ‘Nevalue> “;’, ‘exponent of result requires more than’, #Limit ExponentDigits ‘digits’ #ErrorText.42.2= ‘Arithmetic underflow detected at’, ‘Nevalue> “;‘, ‘exponent of result requires more than’, #Limit ExponentDigits ‘digits’ ”Arithmetic overflow; divisor must not be zero’

#ErrorText.42.3

“Routine not found’ ‘Could not find routine “” ’

#ErrorText.43 = #ErrorText.43.1= #ErrorText.44 = ‘Function did not return data’

#ErrorText.44.1= ‘No data returned from function “” ’

#ErrorText.45 = ‘No data specified on function RETURN’ #ErrorText.45.1= ‘Data expected on RETURN instruction because’, ‘routine “” was called as a function’

#ErrorText.46 = ‘Invalid variable reference’ #ErrorText.46.1= ‘Extra token (“”) found in variable’, ‘reference; “)” expected’

#ErrorText.47 = ‘Unexpected label’ #ErrorText.47.1l= ‘INTERPRET data must not contain labels;’, ’found “”!

#ErrorText.48 = ‘Failure in system service’ #ErrorText.48.1= ‘Failure in system service: ’

#ErrorText.49 = ‘Interpretation Error’ #ErrorText.49.1= ‘Interpretation Error: ’ #ErrorText.50 = ‘Unrecognized reserved symbol’

#ErrorText.50.1= ‘Unrecognized reserved symbol “” ’

#ErrorText.51 = ‘Invalid function name’

#ErrorText.51.1= ‘Unquoted function names must not end with a period;’, ‘found “” ’

#ErrorText.52

“Result returned by”” is longer than’, #Limit String ‘characters’

#ErrorText.53 = ‘Invalid option’ #ErrorText.53.1l= ‘Variable reference expected’, ‘after STREAM keyword; found “” ’ #ErrorText.53.2= ‘Variable reference expected’, ‘after STEM keyword; found “” ’ #ErrorText.53.3= ‘Argument to STEM must have one period,’, ‘as its last character; found “” ’ #ErrorText.54 = ‘Invalid STEM value’ #ErrorText.54.1= ‘For this use of STEM, the value of “” must be a’, ‘count of lines; found: “” ’

If the activity defined by clause 6 does not produce any error message, execution of the program continues.

call Config NoSource

If Config_NoSource has set #NoSource to ‘0’ the lines of source processed by clause 6 are copied to #SourceLine. , with #SourceLine.O being a count of the lines and #SourceLine.n for n=1 to #SourceLine.0 being the source lines in

order.

If Config_NoSource has set #NoSource to ‘1’ then #SourceLine.0 is set to 0. The following state variables affect tracing:

#InhibitPauses = 0

#InhibitTrace = 0

#AtPause = 0 /* Off until interactive input being received. */

#Trace QueryPrior = ‘No’ An initial variable pool is established:

call Config ObjectNew

#Pool = #Outcome

#P0011 = #Pool

call Var_Empty #Pool

call Var_Reset #Pool

#Level = 1 /* Level of invocation */ #NewLevel = 2 #IsFunction.#Level = (#HowInvoked == ‘FUNCTION’)

For this first level, there is no previous level from which values are inherited. The relevant fields are initialized.

#Digits.#Level = 9 /* Numeric Digits */ *#Form.#Level = ‘SCIENTIFIC’* / Numeric Form / *#Fuzz.#Level = 0* / Numeric Fuzz / *#StartTime.#Level = ’ ’* / Elapsed time boundary */ #LineNumber = ’ ’

#Tracing.#Level = ‘N’

#Interactive.#Level = ‘0’

69 An environment is provided by the API_ Start to become the initial active environment to which commands will be addressed. The alternate environment is made the same:

/* Call the environments ACTIVE, ALTERNATE, TRANSIENT where these are never-initialized state variables.

Similarly call the redirections I O and E */

call EnvAssign ALTERNATE, #Level, ACTIVE, #Level

Conditions are initially disabled:

#Enabling.SYNTAX.#Level = ‘OFF’ #Enabling.HALT.#Level = ‘OFF’ #Enabling.ERROR.#Level = ‘OFF’ #Enabling.FAILURE.#Level = ‘OFF’ #Enabling.NOTREADY.#Level = ‘OFF’ #Enabling.NOVALUE.#Level = ‘OFF’ #Enabling.LOSTDIGITS.#Level = ‘OFF’

#PendingNow.HALT.#Level = 0 #PendingNow.ERROR.#Level = 0 #PendingNow.FAILURE.#Level = 0 #PendingNow.NOTREADY.#Level = 0 /* The following field corresponds to the results from the CONDITION built-in function. */ #Condition.#Level = ’ ’ The opportunity is provided for a trap to initialize the pool. #API Enabled = ‘1’ call Var_Reset #Pool call Config Initialization #API Enabled = ‘0’ ## REQUIRES For each requires in order of appearence: A use of Start_API with #instance(requires, taken_constant). Msg40.1 or a new if completion ‘E’. Add Provides to an ordered collection. Not cyclic because .LIST can be defined

without defining REQUIRES but a fairly profound forward reference. ## CLASS For each class in order of appearence: #ClassName = #Instance(class, taken constant) call var_value #ReservedPool, '#CLASSES.'ClassName, '1' if #Indicator == 'D' then do call Config ObjectNew #Class = #Outcome call var_set #ReservedPool, '#CLASSES.'ClassName, '1', #Class end else call #Raise 'SYNTAX', nn.nn, #ClassName

New instance of CLASS class added to list. Msg "Duplicate ::CLASS directive instruction"(?) ## METHOD

For each method in order of appearence: call Config ObjectNew #Po00ol = #Outcome call Config ObjectSource (#Pool) #MethodName = #Instance(method, taken constant) call var_value #Class, '#METHODS.'#MethodName, '1' if #Indicator == 'D' then call var set #Class, '#METHODS.'#MethodName, '1', #Pool else call #Raise 'SYNTAX', nn.nn, #MethodName, #ClassName

GUARDED & public is default. if #contains(method, 'PRIVATE') then m~setprivate; if #contains(method, 'UNGUARDED)) then m~setunguarded

Why is there a keyword for GUARDED but not for PUBLIC here?

Does CLASS option mean ENHANCE with Class class methods?

#CurrentClass ~class(#instance(method, taken_constant), m)

For ATTRIBUTE, should we actually construct source for two methods? ATTRIBUTE case needs test of null body. OO! doesn't have source (because it actually traps UNKNOWN?).

For EXTERNAL test for null body. Simon Nash doc says "Accessibility to external methods … is implementation-defined". Left like that it doesn't even tell us about search order. We will need a Config_ExternalClass to import the public names of the class.

10.3 ROUTINE

For each routine in order of appearence:

Add name (with duplicate check) to list for this file.

Extra step needed in the invocation search order. Although this is nominally EXTERNAL we presumably wont use the external call mechanism. (Except perhaps when the routine was made available by a REQUIRES; in that case the PARSE SOURCE answer has to change.)

have the builtins-defined-by-directives elsewhere; it would make sense if they wound up about here.

11

Instructions

This clause describes the execution of instructions, and how the sequence of execution can vary from the normal execution in order of appearance in the program.

Execution of the program begins with its first clause.

If we left Routine initialization to here we can leave method initialization. ## Method initialization

There is a pool for local variables.

call Config ObjectNew

#Po00ol = #Outcome

Set self and super

11.1 Routine initialization

If the routine is invoked as a function, #lsFunction.#NewLevel shall be set to ‘1’, otherwise to ‘0O’; this affects the processing of a subsequent RETURN instruction.

#AllowProcedure.#NewLevel = ‘1’

Many of the initial values for a new invocation are inherited from the caller’s values. #Digits.#NewLevel = #Digits.#Level

#Form.#NewLevel = #Form.#Level

#Fuzz.#NewLevel = #Fuzz.#Level

#StartTime.#NewLevel = #StartTime.#Level

#Tracing.#NewLevel = #Tracing.#Level #Interactive.#NewLevel = #Interactive.#Level

call EnvAssign ACTIVE, #NewLevel, ACTIVE, #Level call EnvAssign ALTERNATE, #NewLevel, ALTERNATE, #Level

do t=1 to 7 Condition = word(‘SYNTAX HALT ERROR FAILURE NOTREADY NOVALUE LOSTDIGITS’,t) #Enabling.Condition.#NewLevel = #Enabling.Condition.#Level

#Instruction.Condition.#NewLevel = #Instruction.Condition.#Level #TrapName.Condition.#NewL = #TrapName.Condition.#Level #EventLevel.Condition.#NewLevel = #EventLevel.Condition.#Lev end t

If this invocation is not caused by a condition occurring, see nnn, the state variables for the CONDITION

built-in function are copied. #Condition.#NewLevel = #Condition.#Level #ConditionDescription.#N
= #ConditionDescription.#Level #ConditionExtra.#NewLevel = #ConditionExtra.#Level
#ConditionInstruction.#NewLevel = #ConditionInstruction.#Level Execution of
the initialized routine continues at the new level of invocation. #Level = #NewLevel
#NewLevel = #Level + 1

11.2 Clause initialization

The clause is traced before execution: if pos(#Tracing.#Level, 'AIR') > 0 then call #TraceSource

The time of the first use of DATE or TIME will be retained throughout the clause.

#ClauseTime.#Level = ' ' The state variable #LineNumber is set to the line number of the clause, see nnn. A clause other than a null clause or label or procedure instruction sets:

#AllowProcedure.#Level = '0' /* See message 17.1 */

11.3 Clause termination

if #InhibitTrace > 0 then #InhibitTrace = #InhibitTrace - 1 Polling for a HALT condition occurs:

#Response = Config Halt Query ()

if #0utcome == 'Yes' then do call Config Halt Reset

call #Raise 'HALT', substr(#Response,2) /* May return */ end

At the end of each clause there is a check for conditions which occurred and were delayed. It is acted on

if this is the clause in which the condition arose. do t=1 to 4 #Condition=WORD('HALT FAILURE ERROR NOTREADY',t) /* HALT can be established during HALT handling. */ do while #PendingNow.#Condition.#Level #PendingNow.#Condition.#Level = '0' call #Raise end end

Interactive tracing may be turned on via the configuration. Only a change in the setting is significant. call Config Trace Query

if #AtPause = 0 & #Outcome == 'Yes' & #Trace QueryPrior == 'No' then do /* External request for Trace '?R!' */ #Interactive.#Level = '1' #Tracing.#Level = 'R' end

#TraceQueryPrior = #Outcome

Tracing just not the same with NetRexx.

When tracing interactively, pauses occur after the execution of each clause except for CALL, DO the second or subsequent time through the loop, END, ELSE, EXIT, ITERATE, LEAVE, OTHERWISE, RETURN, SIGNAL, THEN and null clauses.

If the character ‘=’ is entered in response to a pause, the prior clause is re-executed.

Anything else entered will be treated as a string of one or more clauses and executed by the language processor. The same rules apply to the contents of the string executed by interactive trace as do for strings executed by the INTERPRET instruction. If the execution of the string generates a syntax error, the standard message is displayed but no condition is raised. All condition traps are disabled during execution of the string. During execution of the string, no tracing takes place other than error or failure return codes from commands. The special variable RC is not set by commands executed within the string, nor is .RC.

If a TRACE instruction is executed within the string, the language processor immediately alters the trace setting according to the TRACE instruction encountered and leaves this pause point. If no TRACE instruction is executed within the string, the language processor simply pauses again at the same point in the program.

At a pause point: if #AtPause = 0 & #Interactive.#Level & #InhibitTrace = 0 then do if #InhibitPauses > 0 then #InhibitPauses = #InhibitPauses-1 else do #TraceInstruction = ‘0’ do forever

call Config Trace Query

if #Outcome == ‘No’ & #Trace QueryPrior == ‘Yes’ then do /* External request to stop tracing. */ #Trace_QueryPrior=#Outcome #Interactive.#Level = ‘0’ #Tracing.#Level = ‘N’ leave end

if #Outcome == ‘Yes’ & #Trace QueryPrior == ‘No’ then do /* External request for Trace ‘?R!’ */ #Trace QueryPrior = #Outcome #Interactive.#Level = ‘1’ #Tracing.#Level = ‘R’ leave end

if #Interactive.#Level | #TraceInstruction then leave

/* Accept input for immediate execution. */

call Config Trace Input

if length(#0utcome) = 0 | #0Outcome == ‘=’ then leave #AtPause = #Level

interpret #Outcome

#AtPause = 0 end /* forever loop */ *if #Outcome == ‘=’ then call #Retry* / With no return */ end end

11.4 Instruction

11.4.1 ADDRESS

For a definition of the syntax of this instruction, see nnn.

An external environment to which commands can be submitted is identified by an environment name. Environment names are specified in the ADDRESS instruction to identify the environment to which a command should be sent.

I/O can be redirected when submitting commands to an external environment. The submitted command's input stream can be taken from an existing stream or from a set of compound variables with a common stem. In the latter case (that is, when a stem is specified as the source for the commands input stream) whole number tails are used to order input for presentation to the submitted command. Stem.0 must contain a whole number indicating the number of compound variables to be presented, and stem. 1 through stem.n (where n=stem.0) are the compound variables to be presented to the submitted command.

Similarly, the submitted command's output stream can be directed to a stream, or to a set of compound variables with a given stem. In the latter case (i.e., when a stem is specified as the destination) compound variables will be created to hold the standard output, using whole number tails as described above. Output redirection can specify a REPLACE or APPEND option, which controls positioning prior to the command's execution. REPLACE is the default.

I/O redirection can be persistently associated with an environment name. The term "environment" is used to refer to an environment name together with the I/O redirections.

At any given time, there will be two environments, the active environment and the alternate environment. When an ADDRESS instruction specifies a command to the environment, any specified I/O redirection applies to that command's execution only, providing a third environment for the duration of the instruction. When an ADDRESS command does not contain a command, that ADDRESS command creates a new active environment, which includes the specified I/O redirection.

The redirections specified on the ADDRESS instruction may not be possible. If the configuration is aware that the command processor named does not perform I/O in a manner compatible with the request, the value of #Env_Type. may be set to 'UNUSED' as an alternative to 'STEM' and 'STREAM' where those values are assigned in the following code.

In the following code the particular use of #Contains(address, expression) refers to an expression immediately contained in the address.

Addrinstr: /* If ADDRESS keyword alone, environments are swapped. / *if #Contains (address, taken constant), & #Contains (address,valueexp), & #Contains (address, 'WITH') then do call EnvAssign TRANSIENT, #Level, ACTIVE, #Level call EnvAssign ACTIVE, #Level, ALTERNATE, #Level call EnvAssign ALTERNATE, #Level, TRANSIENT, #Level return end /* The environment name will be explicitly specified. */ if #Contains(address,taken constant) then Name = #Instance(address, taken _ constant) else Name = #Evaluate(valueexp, expression) if length(Name) > #LimitEnvironmentName then call #Raise 'SYNTAX', 29.1, Name

if #Contains(address,expression) then do /* The command is evaluated (but not issued) at this point. */ Command = #Evaluate (address, expression)

if #Tracing.#Level == 'C' | #Tracing.#Level == 'A' then do call #Trace '»>! end

end

call AddressSetup /* Note what is specified on the ADDRESS instruction. / / If

there is no command, the persistent environment is being set. */ if #Contains(address,expression) then do

call EnvAssign ACTIVE, #Level, TRANSIENT, #Level

return

end

call CommandIssue Command /* See nnn / *return* / From Addrinstr */

AddressSetup: /* Note what is specified on the ADDRESS instruction, into the TRANSIENT environment. / *EnvTail = 'TRANSIENT.'#Level* / Initialize with defaults. */ #Env_Name.EnvTail = ' '

#Env_ Type.I.EnvTail = 'NORMAL' #Env_ Type.O.EnvTail = 'NORMAL' #Env_-Type.E.EnvTail = 'NORMAL'

#Env_Resource.I.EnvTail = ' ' #Env_Resource.O.EnvTail = '! #Env_Resource.E.EnvTail =" /* APPEND / REPLACE does not apply to input. */

#Env_Position.I.EnvTail = 'INPUT' #Env_Position.O.EnvTail = 'REPLACE' #Env_-Position.E.EnvTail = 'REPLACE'

/* If anything follows ADDRESS, it will include the command processor name.*/ #Env_Name.EnvTail = Name

/* Connections may be explicitly specified. */ if #Contains (address, connection) then do

if #Contains(connection,input) then do /* input redirection */ if #Contains (resourcei, 'STREAM') then do #Env_Type.I.EnvTail = 'STREAM' #Env_Resource.1.EnvTail=#Evaluate(reso VAR_SYMBOL) end if #Contains (resourcei, 'STEM') then do #Env_Type.I.EnvTail = 'STEM'

Temp=#Instance (resourcei,VAR_SYMBOL) if #Parses(Temp, stem /* See nnn */) then call #Raise 'SYNTAX', 53.3, Temp #Env_Resource.I.EnvTail=Temp end end / Input */

if #Contains(connection,output) then /* output redirection */ call NoteTarget O

if #Contains(connection,error) then /* error redirection / / The prose on the description of #Contains specifies that the relevant resourceo is used in NoteTarget. */ call NoteTarget E

end /* Connection */

return /* from AddressSetup */

NoteTarget:

/* Note the characteristics of an output resource. */

arg Which /* O or E */

if #Contains (resourceo, 'STREAM') then do #Env_Type.Which.EnvTail='STREAM' #Env_Resource.Which.EnvTail=#Evaluate(resourceo, VAR_SYMBOL) end

if #Contains(resourceo,'STEM') then do #Env_Type.Which.EnvTail='STEM' Temp=#Instance (resourceo, VAR_SYMBOL) if #Parses(Temp, stem /* See nnn */) then

call #Raise 'SYNTAX', 53.3, Temp

#Env_Resource.Which.EnvTail=Temp end

if #Contains (resourceo,append) then #Env_Position.Which.EnvTail='APPEND' return /* From NoteTarget */

EnvAssign: /* Copy the values that name an environment and describe its redirections. */ arg Lhs, LhsLevel, Rhs, RhsLevel #Env_Name.Lhs.LhsLevel = #Env_Name.Rhs.RhsLevel #Env_ Type.I.Lhs.LhsLevel = #Env_Type.I.Rhs.RhsLevel #Env_ Resource.I.Lhs.LhsLevel = #Env_Resource.I.Rhs.RhsLevel #Env_Position.I.Lhs.LhsLeve = #Env_Position.I.Rhs.RhsLevel #Env_ Type.O.Lhs.LhsLevel = #Env_Type.O.Rhs.RhsLevel #Env_Resource.O.Lhs.LhsLevel = #Env_Resource.O.Rhs.RhsLevel #Env_-Position.O.Lhs.LhsLevel = #Env_Position.O.Rhs.RhsLevel #Env_ Type.E.Lhs.LhsLevel = #Env_Type.E.Rhs.RhsLevel #Env_Resource.E.Lhs.LhsLevel #Env_Resource.E.Rhs.RhsLeve #Env_Position.E.Lhs.LhsLevel #Env_Position.E.Rhs.RhsLevel return

11.4.2 ARG

For a definition of the syntax of this instruction, see nnn.

The ARG instruction is a shorter form of the equivalent instruction: PARSE UPPER ARG template list

11.4.3 Assignment

Assignment can occur as the result of executing a clause containing an assignment (see nnn and nnn), or as a result of executing the VALUE built-in function, or as part of the execution of a PARSE instruction. Assignment involves an expression and a VAR_SYMBOL. The value of the expression is determined; see nnn.

If the VAR_SYMBOL does not contain a period, or contains only one period as its last character, the

value is associated with the VAR_SYMBOL: call Var Set #Pool,VAR SYMBOL, '0',Value

Otherwise, a name is derived, see nnn. The value is associated with the derived name: call Var Set #Pool,Derived Name,'1',Value

11.4.4 CALL

For a definition of the syntax of this instruction, see nnn.

The CALL instruction is used to invoke a routine, or is used to control trapping of conditions. lf a vrefis specified that value is the name of the routine to invoke:

if #Contains (call, vref) then Name = #Evaluate(vref, var_symbol)

If a taken_constant is specified, that name is used. if #Contains (call, taken constant) then Name = #Instance(call, taken constant)

The name is used to invoke a routine, see nnn. If that routine does not return a result the RESULT and

-RESULT variables become uninitialized: call Var Drop #Pool, ‘RESULT’, ‘0! call Var Drop #ReservedPool,’.RESULT’, ‘0’

If the routine does return a result that value is assigned to RESULT and .RESULT. See nnn for an exception to assigning results.

If the routine returns a result and the trace setting is ‘R’ or ‘I’ then a trace with that result and a tag ’»>” shall be produced, associated with the call instruction.

If a callon_spec is specified: If #Contains(call,callon spec) then do Condition = #Instance(callon_spec,callable condition)

#Instruction.Condition.#Level = ‘CALL’ If #Contains(callon spec, ‘OFF’) then #Enabling.Condition.#Level = ‘OFF’ else #Enabling.Condition.#Level = ‘ON’

/* Note whether NAME supplied. */ If Contains (callon spec,taken constant) then Name = #Instance (callable condition, taken_constant)

else

Name = Condition #TrapName.Condition.#Level = Name end

11.4.5 Command to the configuration

For a definition of the syntax of a command, see nnn. A command that is not part of an ADDRESS instruction is processed in the ACTIVE environment.

Command = #Evaluate(command, expression)

if #Tracing.#Level == ‘C’ | #Tracing.#Level == ‘A’ then call #Trace ‘»>!’

call EnvAssign TRANSIENT, #Level, ACTIVE, #Level

call CommandiIssue Command

CommandIssue is also used to describe the ADDRESS instruction:

CommandIssue: parse arg Cmd /* Issues the command, requested environment is TRANSIENT / / This description does not require the command processor to understand stems, so it uses an altered environment. */ call EnvAssign PASSED, #Level, TRANSIENT, #Level EnvTail = ‘TRANSIENT.’#Level

/* Note the command input. */ *if #Env_Type.I.EnvTail = ‘STEM’ then do* / Check reasonableness of the stem. */ *Stem = #Env_Resource.I.EnvTail Lines = value(Stem’0’) if (Lines,‘W’) then call #Raise ‘SYNTAX’,54.1,Stem’0’, Lines if Lines<0 then call #Raise ‘SYNTAX’,54.1,Stem’0’, Lines* / Use a stream for the stem */ #Env_-Type.I.PASSED.#Level = ‘STREAM’ call Config Stream Unique InputStream = #Outcome #Env_Resource.1I.PASSED.#Level = InputStream call charout InputStream , vl do j = 1 to Lines call lineout InputStream, value(Stem || j) end j call lineout InputStream end

/* Note the command output. */

if #Env_Type.O.EnvTail = ‘STEM’ then do Stem = #Env_Resource.O.EnvTail if #Env_Position.O.EnvTail == ‘APPEND’ then do

/* Check that Stem.0 will accept incrementing. */ *Lines=value (Stem’0’); if (Lines,‘W’) then call #Raise ‘SYNTAX’,54.1,Stem’0’, Lines if Lines<0 then call #Raise*

‘SYNTAX’,54.1,Stem’0’, Lines end else call value Stem’0’,O / Use a stream for the stem */ #Env_Type.O.PASSED.#Level = ‘STREAM’ call Config Stream Unique #Env_Resource.O.PASSED.#Level = #Outcome end

/* Note the command error stream. */

if #Env_Type.E.EnvTail = ‘STEM’ then do Stem = #Env_Resource.E.EnvTail if #Env_Position.E.EnvTail == ‘APPEND’ then do

/* Check that Stem.0 will accept incrementing. */ Lines=value (Stem’0’); if (Lines,‘W’) then call #Raise ‘SYNTAX’,54.1,Stem’0’, Lines if Lines<0 then call #Raise ‘SYNTAX’,54.1,Stem’0’, Lines

end else call value Stem’0’,0O /* Use a stream for the stem */ #Env_ Type.E.PASSED.#Level = ‘STREAM’ call Config Stream Unique #Env_Resource.E.PASSED.#Level = #Outcome end

#API Enabled = ‘1’

call Var_Reset #Pool

/* Specifying PASSED here implies all the

components of that environment. */

#Response = Config Command (PASSED, Cmd) #Indicator = left (#Response,1) Description = substr (#Response, 2)

#API Enabled = ‘0’

/* Recognize success and failure. */ if #AtPause = 0 then do

call value ‘RC’, #RC

call var Set 0, ‘.RC’, 0, #RC end

select when #Indicator==‘N’ then Temp=0

when #Indicator==‘F’ then Temp=-1 /* Failure */ *when #Indicator==‘E’ then Temp=1* / Error */ end call Var Set 0, ‘.RS’, 0, Temp* / Process the output */ if #Env_Type.O.EnvTail=‘STEM’ then do* / get output into stem. */ Stem = #Env_-Resource.0O.EnvTail OutputStream = #Env_Resource.0O.PASSED.#Level do while lines (OutputStream) > 0 call value Stem’0O’,value(Stem’0’)4+1 call value Stem*| |value(Stem’0’),linein (OutputStream) end end */* Stemmed Output */ if #Env_Type.E.EnvTail=‘STEM’ then do* / get error output into stem. */ Stem = #Env_Resource.E.EnvTail OutputStream = #Env_Resource.E.PASSED.#Level do while lines (OutputStream) > 0 call value Stem’0O’,value(Stem’0’)4+1 call value Stem| |value(Stem’0’),linein (OutputStream)

end end /* Stemmed Error output */ if #Indicator == ‘N’ & pos(#Tracing.#Level, ‘CAIR’) > 0 then call #Trace ‘+++’ if (#Indicator == ‘N’ & #Tracing.#Level==‘E’), | (#Indicator==‘F’ & (#Tracing.#Level==‘F’ | #Tracing.#Level==‘N’)) then do

call #Trace ‘»>!’ call #Trace ‘+++!’

end #Condition=‘FAILURE’ if #Indicator=‘F’ & #Enabling.#Condition.#Level == ‘OFF’ then call #Raise ‘FAILURE’ , Cmd else if #Indicator=‘E’ | #Indicator=‘F’ then call #Raise ‘ERROR’, Cmd

return /* From CommandIssue */

The configuration may choose to perform the test for message 54.1 before or after issuing the command.

11.4.6 DO

For a definition of the syntax of this instruction, see nnn.

The DO instructions is used to group instructions together and optionally to execute them repeatedly. Executing a do_simple has the same effect as executing a nop, except in its trace output. Executing the do_ending associated with a do_simple has the same effect as executing a nop, except in its trace output.

A do_instruction that does not contain a do_simple is equivalent, except for trace output, to a sequence of instructions in the following order.

#Loop = #Loop+1 #Iterate.#Loop = #Clause (IterateLabel) #Once.#Loop = #Clause (OnceLabel) #Leave.#Loop = #Clause (LeaveLabel) if #Contains (do specification,assignment) then #Identity.#Loop = #Instance(assignment, VAR SYMBOL) if #Contains (do specification, repexpr) then if (repexpr,'W') then call #Raise 'SYNTAX', 26.2,repexpr else do #Repeat.#Loop = repexpr+0 if #Repeat.#Loop<0 then call #Raise 'SYNTAX',26.2,#Repeat.#Loop end if #Contains (do specification,assignment) then do #StartValue.#Loop = #Evaluate (assignment, expression) if datatype (#StartValue.#Loop) == 'NUM' then call #Raise 'SYNTAX', 41.6, #StartValue.#Loop #StartValue.#Loop = #StartValue.#Loop + 0 if #Contains (do specification,byexpr) then #By.#Loop = 1 end

The following three assignments are made in the order in which 'TO'", 'BY' and 'FOR' appear in docount; see nnn.

if #Contains (do specification, toexpr) then do if datatype(toexpr) == 'NUM' then call #Raise 'SYNTAX', 41.4, toexpr #To.#LOop = toexpr+0 if #Contains (do specification, byexpr) then do if datatype (byexpr) =='NUM' then call #Raise 'SYNTAX', 41.5, byexpr #By.#Loop = byexpr+0 if #Contains (do specification, forexpr) then do if (forexpr, 'W') then call #Raise 'SYNTAX', 26.3, forexpr #For.#Loop = forexpr+0 if #For.#Loop <0 then call #Raise 'SYNTAX', 26.3, #For.#Loop end if #Contains (do specification,assignment) then do call value #Identity.#Loop, #StartValue.#Loop end if #Contains (do specification, 'OVER') then do Value = #Evaluate(dorep, expression) #OverArray.#Loop = Value ~ makearray

#Repeat.#Loop = #OverArray~items /* Count this downwards as if repexpr. */ #Iidentity.#Loop = #Instance(dorep, VAR SYMBOL) end

call #Goto #Once.#Loop /* to OnceLabel */

IterateLabel:

if #Contains (do specification, untilexpr) then do Value = #Evaluate(untilexp, expression)

if Value == '1' then leave if Value == '0' then call #Raise 'SYNTAX', 34.4, Value end

if #Contains (do specification, assignment) then do t = value (#Identity. #Loop)

if #Indicator == ‘D’ then call #Raise ‘NOVALUE’, #Identity.#Loop call value #Identity.#Loop, t + #By.#Loop end

OnceLabel:

if #Contains (do specification, toexpr) then do if #By.#Loop>=0 then do if value(#Identity.#Loop) > #To.#Loop then leave end else do if value(#Identity.#Loop) < #To.#Loop then leave end end

if #Contains(dorep, repexpr) | #Contains(dorep, ‘OVER’) then do if #Repeat.#Loop = 0 then leave #Repeat.#Loop = #Repeat.#Loop-1 if #Contains(dorep, ‘OVER’) then call value #Identity.#Loop, #OverArray[#OverArray~items - #Repeat.#Loop] end if #Contains (do specification, forexpr) then do if #For.#Loop = 0 then leave #For.#Loop = #For.#Loop - 1 end if #Contains (do specification, whileexpr) then do Value = #Evaluate(whileexp, expression)

if Value == ‘0’ then leave if Value == ‘1’ then call #Raise ‘SYNTAX’, 34.3, Value end #Execute (do instruction, instruction list) TraceOfEnd: call #Goto #Iterate.#Loop /* to IterateLabel */ LeaveLabel:

#Loop = #Loop - 1

11.4.7 DO loop tracing

When clauses are being traced by #TraceSource, due to pos(#Tracing.#Level, ‘AIR’) > 0, the DO instruction shall be traced when it is encountered and again each time the IterateLabel (see nnn) is encountered. The END instruction shall be traced when the TraceOfEnd label is encountered.

When expressions or intermediates are being traced they shall be traced in the order specified by nnn. Hence, in the absence of conditions arising, those executed prior to the first execution of OnceLabel shall be shown once per execution of the DO instruction; others shall be shown depending on the outcome of the tests.

The code in the DO description: t = value (#Identity. #Loop) if #Indicator == ‘D’ then call #Raise ‘NOVALUE’, #Identity.#Loop call value #Identity.#Loop, t + #By.#Loop

represents updating the control variable of the loop. That assignment is subject to tracing, and other expressions involving state variables are not. When tracing intermediates, the BY value will have a tag of ‘>+>’,

11.4.8 DROP

For a definition of the syntax of this instruction, see nnn.

The DROP instruction restores variables to an uninitialized state.

The words of the variable_list are processed from left to right.

A word which is a VAR_SYMBOL, not contained in parentheses, specifies a variable to be dropped. If VAR_SYMBOL does not contain a period, or has only a single period as its last character, the variable

associated with VAR_SYMBOL by the variable pool is dropped:

#Response = Var Drop (#Pool,VAR_ SYMBOL, ‘0’)

If VAR_SYMBOL has a period other than as the last character, the variable associated with VAR_SYMBOL by the variable pool is dropped by:

#Response = Var Drop (#Pool,VAR SYMBOL, ‘1’)

If the word of the variable_list is a VAR_SYMBOL enclosed in parentheses then the value of the

VAR_SYMBOL is processed. The value is considered in uppercase: #Value = Config Upper (#Value)

Each word in that value found by the WORD built-in function, from left to right, is subjected to this process:

If the word does not have the syntax of VAR_SYMBOL a condition is raised: call #Raise ‘SYNTAX’, 20.1, word

Otherwise the VAR_SYMBOL indicated by the word is dropped, as if that VAR_-SYMBOL were a word of the variable_list.

11.4.9 EXIT

For a definition of the syntax of this instruction, see nnn.

The EXIT instruction is used to unconditionally complete execution of a program.

Any expression is evaluated:

if #Contains(exit, expression) then Value = #Evaluate(exit, expression) #Level = 1

#Pool = #Pooll

The opportunity is provided for a final trap. #API Enabled = ‘1’ call Var_Reset #Pool call Config Termination #API Enabled = ‘0’

The processing of the program is complete. See nnn for what API Start returns as the result.

If the normal sequence of execution “falls through” the end of the program; that is, would execute a further statement if one were appended to the program, then the program is terminated in the same manner as an EXIT instruction with no argument.

11.4.10 EXPOSE

The expose instruction identifies variables that are not local to the method.

We need a check that this starts method; similarities with PROCEDURE.

For a definition of the syntax of this instruction, see nnn.

It is used at the start of a method, after method initialization, to make variables in the receiver’s pool

accessible: if #AllowExpose then call #Raise ‘SYNTAX’, 17.2

The words of the variable_list are processed from left to right.

A word which is a VAR_SYMBOL, not contained in parentheses, specifies a variable to be made accessible. If VAR_SYMBOL does not contain a period, or has only a single period as its last character, the variable associated with VAR_SYMBOL by the variable pool (as a non-tailed name) is given the

attribute ‘exposed’. call Var_ Expose #Pool, VAR SYMBOL, ‘0’

If VAR_SYMBOL has a period other than as last character, the variable associated with VAR_SYMBOL

in the variable pool (by the name derived from VAR_SYMBOL, see nnn) is given the attribute ‘exposed’. call Var_ Expose #Pool, Derived Name, ‘1’

If the word from the variable_list is a VAR_SYMBOL enclosed in parentheses then the VAR_SYMBOL is exposed, as if that VAR_SYMBOL was a word in the variable_list. The value of the VAR_SYMBOL is

processed. The value is considered in uppercase: #Value = Config Upper (#Value)

Each word in that value found by the WORD built-in function, from left to right, is subjected to this process:

If the word does not have the syntax of VAR_SYMBOL a condition is raised: call #Raise ‘SYNTAX’, 20.1, word

Otherwise the VAR_SYMBOL indicated by the word is exposed, as if that VAR_-SYMBOL were a word of the variable_list.

11.4.11 FORWARD

For a definition of the syntax of this instruction, see nnn.

The FORWARD instruction is used to send a message based on the current message. if #Contains (forward, ‘ARRAY’) & #Contains(forward, ‘ARGUMENTS’) then call #Raise ‘SYNTAX’, nn.n

11.4.12 GUARD

For a definition of the syntax of this instruction, see nnn.

The GUARD instruction is used to conditionally delay the execution of a method. do forever Value = #Evaluate(guard, expression)

if Value == ‘1’ then leave

if Value == ‘0’ then call #Raise ‘SYNTAX’, 34.7, Value Drop exclusive access and wait for change

end ### IF

For a definition of the syntax of this instruction, see nnn.

The IF instruction is used to conditionally execute an instruction, or to select

between two alternatives. The expression is evaluated. If the value is neither ‘0’ nor ‘1’ error 34.1 occurs. If the value is ‘1’, the instruction in the then is executed. If the value is ‘0’ and e/se is specified, the instruction in the else is executed.

In the former case, if tracing clauses, the clause consisting of the THEN keyword shall be traced in addition to the instructions.

In the latter case, if tracing clauses, the clause consisting of the ELSE keyword shall be traced in addition to the instructions.

11.4.13 INTERPRET

For a definition of the syntax of this instruction, see nnn.

The INTERPRET instruction is used to execute instructions that have been built dynamically by evaluating an expression.

The expression is evaluated.

The HALT condition is tested for, and may be raised, in the same way it is tested at clause termination, see nnn.

The process of syntactic recognition described in clause 6 is applied, with Config_SourceChar obtaining its results from the characters of the value, in left-to-right order, without producing any EOL or EOS events. When the characters are exhausted, the event EOL occurs, followed by the event EOS.

If that recognition would produce any message then the interpret raises the corresponding ‘SYNTAX’ condition.

If the program recognized contains any LABELs then the interpret raises a condition: call #Raise ‘SYNTAX’,47.1,Label

where Label is the first LABEL in the program.

Otherwise the instruction_list in the program is executed.

11.4.14 ITERATE

For a definition of the syntax of this instruction, see nnn.

The ITERATE instruction is used to alter the flow of control within a repetitive DO. For a definition of the nesting correction, see nnn.

#Loop = #Loop - NestingCorrection call #Goto #Iterate.#Loop

11.4.15 Execution of labels

The execution of a label has no effect, other than clause termination activity and any tracing. if #Tracing.#Level==‘L’ then call #TraceSource

11.4.16 LEAVE

For a definition of the syntax of this instruction, see nnn.

The LEAVE instruction is used to immediately exit one or more repetitive DOs. For a definition of the nesting correction, see nnn.

#Loop = #Loop - NestingCorrection call #Goto #Leave.#Loop

11.4.17 Message term

We can do this by reference to method invokation, just as we do CALL by reference to invoking a function.

11.4.18 LOOP

Shares most of it's definition with repetitive DO.

11.4.19 NOP

For a definition of the syntax of this instruction, see nnn. The NOP instruction has no effect other than the effects associated with all instructions.

11.4.20 NUMERIC

For a definition of the syntax of this instruction, see nnn. The NUMERIC instruction is used to change the way in which arithmetic operations are carried out.

NUMERIC DIGITS

For a definition of the syntax of this instruction, see nnn.

NUMERIC DIGITS controls the precision under which arithmetic operations and arithmetic built-in functions will be evaluated.

if #Contains(numericdigits, expression) then

Value = #Evaluate(numericdigits, expression) else Value = 9

if (Value,'W') then

call #Raise 'SYNTAX',26.5,Value Value = Value + 0 if Value<=#Fuzz.#Level then

call #Raise 'SYNTAX',33.1,Value if Value>#Limit Digits then

call #Raise 'SYNTAX',33.2,Value #Digits.#Level = Value

NUMERIC FORM

For a definition of the syntax of this instruction, see nnn.

NUMERIC FORM controls which form of exponential notation is to be used for the results of operations and arithmetic built-in functions.

The value of form is either taken directly from the SCIENTIFIC or ENGINEERING keywords, or by

evaluating valueexp . if #Contains (numeric,numericsuffix) then

Value = 'SCIENTIFIC' else if #Contains (numericformsuffix, 'SCIENTIFIC') then Value = 'SCIENTIFIC' else if #Contains (numericformsuffix, 'ENGINEERING') then Value = 'ENGINEERING' else do Value = #Evaluate (numericformsuffix,valueexp) Value = translate (left (Value,1)) select when Value == 'S' then Value = 'SCIENTIFIC' when Value == 'E' then Value = 'ENGINEERING' otherwise call #Raise 'SYNTAX',33.3,Value end end #Form.#Level = Value

NUMERIC FUZZ

For a definition of the syntax of this instruction, see nnn. NUMERIC FUZZ controls how many digits, at full precision, will be ignored during a numeric comparison.

If #Contains (numericfuzz,expression) then Value = #Evaluate (numericfuzz,expression) else Value = 0 If (Value,'W') then call #Raise 'SYNTAX',26.6,Value Value = Value+0 If Value < 0 then call #Raise 'SYNTAX',26.6,Value If Value >= #Digits.#Level then call #Raise 'SYNTAX',33.1,#Digits.#Level,Value #Fuzz.#Level = Value

OPTIONS

For a definition of the syntax of this instruction, see nnn.

The OPTIONS instruction is used to pass special requests to the language processor.

The expression is evaluated and the value is passed to the language processor. The language processor treats the value as a series of blank delimited words. Any words in the value that are not recognized by

the language processor are ignored without producing an error. call Config Options (Expression)

PARSE

For a definition of the syntax of this instruction, see nnn.

The PARSE instruction is used to assign data from various sources to variables.

The purpose of the PARSE instruction is to select substrings of the parse_fype under control of the template_list. |f the template_list is omitted, or a template

in the list is omitted, then a template which is the null string is implied.

Processing for the PARSE instruction begins by constructing a value, the source to be parsed.

ArgNum = 0 select when #Contains (parse type, ‘ARG’) then do ArgNum = 1 ToParse = #Arg.#Level.ArgNum end when #Contains (parse type, ‘LINEIN’) then ToParse = linein(‘‘) when #Contains (parse type, ’PULL’) then do /* Acquire from external queue or default input. */ #Response = Config Pull() if left(#Response, 1) == ‘F’ then call Config Default Input ToParse = #Outcome end when #Contains (parse type, ‘SOURCE’) then ToParse = #Configuration #HowInvoked #Source when #Contains (parse type, ‘VALUE’) then if #Contains(parse value, expression) then ToParse = ’ ’ else ToParse = #Evaluate(parse value, expression) when #Contains (parse type, ‘VAR’) then ToParse = #Evaluate (parse var, VAR_-SYMBOL) when #Contains (parse type, ‘VERSION’) then ToParse = #Version end Uppering = #Contains(parse, ‘UPPER’) The first template is associated with this source. If there are further templates, they are matched against null strings unless ‘ARG’ is specified, when they are matched against further arguments. The parsing process is defined by the following routine, ParseData. The template_list is accessed by ParseData as a stemmed variable. This variable Template. has elements which are null strings except

for any elements with tails 1,2,3,… corresponding to the tokens of the template_-list from left to right.

ParseData: /* Targets will be flagged as the template is examined. / *Target.=‘0’* / Token is a cursor on the components of the template, moved by FindNextBreak. / *Token = 1* / Tok ig a cursor on the components of the template moved through the target variables by routine WordParse. */ Tok = 1

do forever /* Until commas dealt with. / / BreakStart and BreakEnd indicate the position in the source string where there is a break that divides the source. When the break is a pattern they are the start of the pattern and the position just beyond it. */ BreakStart = BreakEnd = 1 SourceEnd = length(ToParse) + 1 If Uppering then ToParse = translate (ToParse)

do while Template.Tok == ’ ’ & Template.Tok == ‘,’

/* Isolate the data to be processed on this iteration. / *call FindNextBreak* / Also marks targets. */

/* Results have been set in DataStart which indicates the start of the isolated data and BreakStart and BreakEnd which are ready for the next iteration. Tok has not changed. */

/* If a positional takes the break leftwards from the end of the previous selection, the source selected is the rest of the string, */

if BreakEnd <= DataStart then DataEnd = SourceEnd

else DataEnd = BreakStart

/* Isolated data, to be assigned from: */

Data=substr (ToParse,DataStart, DataEnd-DataStart) call WordParse /* Does the assignments. */

end /* while / *if Template.Tok == ‘,’ then leave* / Continue with next source. */ Token=Token+1 Tok=Token if ArgNum <> 0 then do ArgNum = ArgNum+1 ToParse = #Arg.ArgNum end else ToParse=’ ’ end

return /* from ParseData */

FindNextBreak: do while Template.Token == ’ ’ & Template.Token == ‘,’

Type=left (Template.Token,1) /* The source data to be processed next will normally start at the end of the break that ended the previous piece. (However, the relative positionals alter this.) */ DataStart = BreakEnd select

when Type=‘ “’ | Type=” ”’ | Type=‘(’ then do if Type=‘(’ then do /* A parenthesis introduces a pattern which is not a constant. */ Token = Token+1 Pattern = value(Template.Token)

if #Indicator == ‘D’ then call #Raise ‘NOVALUE’, Template.Token Token = Token+1 end

else

/* The following removes the outer quotes from the literal pattern / *interpret “Pattern=”Template.Token Token = Token+1* / Is that pattern in the remaining source? / *PatternPos=pos (Pattern, ToParse,DataStart) if PatternPos>0 then do* / Selected source runs up to the pattern. / *BreakStart=PatternPos BreakEnd=PatternPos+length (Pattern) return end leave* / The rest of the source is selected. */ end

when datatype(Template.Token,‘W’) | pos(Type,‘+-=’) > 0 then do /* A positional specifies where the relevant piece of the subject ends. / *if pos (Type, ‘+-=’) = 0 then do* / Whole number positional / *BreakStart = Template.Token Token = Token+1 end else do* / Other forms of positional. / *Direction=Template.Token Token = Token + 1* / For a relative positional, the position is relative to the start of the previous trigger, and the source segment starts there. / *if Direction == ‘=’ then DataStart = BreakStart* / The adjustment can be given as a number or a variable in parentheses. */ if Template.Token =‘(’ then do Token=Token + 1 BreakStart = value(Template. Token) if #Indicator == ‘D’ then call #Raise ‘NOVALUE’, Template.Token Token=Token + 1 end else BreakStart = Template.Token

if

if

el

(BreakStart,‘W’)

then call #Raise ‘SYNTAX’, 26.4,BreakStart Token = Token+1

Direction=‘+’

then BreakStart=DataStart+BreakStart se if Direction=‘-’

then BreakStart=DataStart-BreakStart

end /* Adjustment should remain within the ToParse / *BreakStart = max(1, BreakStart) BreakStart = min(SourceEnd, BreakStart) BreakEnd = BreakStart* / No actual literal marks the boundary. */ return

end

when Template.Token == ‘.’ & pos(Type, ‘0123456789.’)>0 then /* A number

that isn’t a whole number. */

call

#Raise ‘SYNTAX’, 26.4, Template.Token

/* Raise will not return */

otherwise do /* It is a target, not a pattern */ Target.Token=‘1’ Token = Token+1

end

end /*

select */

end /* while / / When no more explicit breaks, break is at the end of the source. */ DataStart=BreakEnd BreakStart=SourceEnd BreakEnd=SourceEnd

return

WordParse: /* The names in the template are assigned blank-delimited values from the source string. */

/* From FindNextBreak */

do while Target.Tok /* Until no more targets for this data. */

/* Last target gets all the residue of the Data.

Next Tok

= Tok + 1

if .NextTok then do call Assign (Data)

leave end /* Not 1 Data = s if Data else do Word =

ast target; assign a word. */ trip (Data, ‘L’) == ’ ’ then call Assign(’ ’)

word (Data,1)

call Assign Word

Data =

substr(Data,length(Word) + 1)

*/

/* The word terminator is not part of the residual data: */ if Data == ’ ’ then Data = substr (Data, 2)

end

Tok = Tok + 1

*/

end Tok=Token /* Next time start on new part of template. / *return Assign: if Template.Tok==‘.’ then Tag=‘>.>’ else do Tag=‘>=>’ call value Template.Tok,arg(1) end* / Arg(1) is an implied argument of the tracing. if #Tracing.#Level == ‘R’ | #Tracing.#Level == ‘I’

return

11.4.21 PROCEDURE

For a definition of the syntax of this instruction, see nnn.

then call #Trace Tag The PROCEDURE instruction is used within an internal routine to protect all the existing variables by making them unknown to following instructions. Selected variables may be exposed.

It is used at the start of a routine, after routine initialization:

if #AllowProcedure.#Level then call #Raise ‘SYNTAX’, 17.1

#AllowProcedure.#Level = 0

/* It introduces a new variable pool: */

call #Config ObjectNew

call var_set (#Outcome,‘#UPPER’, ‘0’, #Pool) /* Previous #Pool is upper from the new #Pool. */

#Pool=#O0Outcome

IsProcedure.#Level=‘1’

call Var_Empty #Pool

If there is a variable_list, it provides access to a previous variable pool.

The words of the variable_list are processed from left to right.

A word which is a VAR_SYMBOL, not contained in parentheses, specifies a variable to be made accessible. If VAR_SYMBOL does not contain a period, or has only a single period as its last character, the variable associated with VAR_SYMBOL by the variable pool (as a non-tailed name) is given the

attribute ‘exposed’. call Var_ Expose #Pool, VAR SYMBOL, ‘0’

If VAR_SYMBOL has a period other than as last character, the variable associated with VAR_SYMBOL

in the variable pool (by the name derived from VAR_SYMBOL, see nnn) is given the attribute ‘exposed’. call Var_ Expose #Pool, Derived Name, ‘1’

If the word from the variable_list is a VAR_SYMBOL enclosed in parentheses then the VAR_SYMBOL is exposed, as if that VAR_SYMBOL was a word in the variable_list. Tne value of the VAR_SYMBOL is

processed. The value is considered in uppercase: #Value = Config Upper (#Value)

Each word in that value found by the WORD built-in function, from left to right, is subjected to this process:

If the word does not have the syntax of VAR_SYMBOL a condition is raised: call #Raise ‘SYNTAX’, 20.1, word

Otherwise the VAR_SYMBOL indicated by the word is exposed, as if that VAR_-SYMBOL were a word of the variable_list.

11.4.22 PULL

For a definition of the syntax of this instruction, see nnn.

A PULL instruction is a shorter form of the equivalent instruction: PARSE UPPER PULL template list

11.4.23 PUSH

For a definition of the syntax of this instruction, see nnn. The PUSH instruction is used to place a value on top of the stack.

If #Contains(push,expression) then Value = #Evaluate (push, expression) else Value = ' ' call Config Push Value

11.4.24 QUEUE

For a definition of the syntax of this instruction, see nnn. The QUEUE instruction is used to place a value on the bottom of the stack.

If #Contains (queue,expression) then Value = #Evaluate (queue, expression) else Value = ' ' call Config Queue Value

11.4.25 RAISE

The RAISE instruction returns from the current method or routine and raises a condition.

11.4.26 REPLY

The REPLY instruction is used to allow both the invoker of a method, and the replying method, to continue executing.

Must set up for error of expression on subsequent RETURN. ### RETURN For a definition of the syntax of this instruction, see nnn. The RETURN instruction is used to return control and possibly a result from a program or internal routine to the point of its invocation. The RETURN keyword may be followed by an optional expression, which will be evaluated and returned as a result to the caller of the routine. Any expression is evaluated: if #Contains(return,expression) then #Outcome = #Evaluate(return, expression)

else if #IsFunction.#Level then call #Raise 'SYNTAX', 45.1, #Name.#Level

At this point the clause termination occurs and then the following:

If the routine started with a PROCEDURE instruction then the associated pool is taken out of use: if #IsProcedure.#Level then #Pool = #Upper A RETURN instruction which is interactively entered at a pause point leaves the pause point. if #Level = #AtPause then #AtPause = 0 The activity at this level is complete:

#Level = #Level-1 #NewLevel = #Level+1 If #Level is not zero, the processing of the RETURN instruction and the invocation is complete. Otherwise processing of the program is completed:

The opportunity is provided for a final trap. #API Enabled = ‘1’

call Var_Reset #Pool

call Config Termination

#API Enabled = ‘0’

The processing of the program is complete. See nnn for what API Start returns as the result. ### SAY

For a definition of the syntax of this instruction, see nnn.

The SAY instruction is used to write a line to the default output stream.

If #Contains(say,expression) then Value = Evaluate (say, expression) else Value = ’ ’ call Config Default Output Value

11.4.27 SELECT

For a definition of the syntax of this instruction, see nnn.

The SELECT instruction is used to conditionally execute one of several alternative instructions. When tracing, the clause containing the keyword SELECT is traced at this point.

The #Contains(select_body, when) test in the following description refers to the items of the optional when repetition in order:

LineNum = #LineNumber Ending = #Clause (EndLabel) Value=#Evaluate (select body, expression) /* In the required WHEN */ *if Value == ‘1’ & Value == ‘0’ then call #Raise ‘SYNTAX’,34.2,Value If Value==‘1’ then call #Execute when, instruction else do do while #Contains (select body, when) Value = #Evaluate (when, expression) If Value==‘1’ then do call #Execute when, instruction call #Goto Ending end if Value == ‘0’ then call #Raise ‘SYNTAX’, 34.2, Value end /* Of each when */

If #Contains(select body, ‘OTHERWISE’) then call #Raise ‘SYNTAX’, 7.3, LineNum If #Contains (select body, instruction list) then call #Execute select body, instruction list end EndLabel:

When tracing, the clause containing the END keyword is traced at this point.

11.4.28 SIGNAL

For a definition of the syntax of this instruction, see nnn.

The SIGNAL instruction is used to cause a change in the flow of control or is used with the ON and OFF keywords to control the trapping of conditions.

If #Contains (signal,signal spec) then do Condition = #Instance(signal spec,condition)

#Instruction.Condition.#Level = ‘SIGNAL’ If #Contains (signal spec, ‘OFF’) then

#Enabling.Condition.#Level = ‘OFF’ else #Enabling.Condition.#Level = ‘ON’

If Contains (signal spec,taken constant) then Name = #Instance (condition, taken constant)

else

Name = Condition #TrapName.Condition.#Level = Name end

If there was a signal_spec this complete the processing of the signal instruction. Otherwise: if #Contains (signal, valueexp)

then Name #Evaluate(valueexp, expression)

else Name #Instance(signal,taken constant)

The Name matches the first LABEL in the program which has that value. The comparison is made with the ‘==“’ operator.

If no label matches then a condition is raised:

call #Raise ‘SYNTAX’,16.1, Name

If the name is a trace-only label then a condition is raised: call #Raise ‘SYNTAX’, 16.2, Name

If the name matches a label, execution continues at that label after these settings: #Loop.#Level = 0

/* A SIGNAL interactively entered leaves the pause point. */

if #Level = #AtPause then #AtPause = 0

11.4.29 TRACE

For a definition of the syntax of this instruction, see nnn.

The TRACE instruction is used to control the trace setting which in turn controls the tracing of execution of the program.

The TRACE instruction is ignored if it occurs within the program (as opposed to source obtained by

Config_Trace_Input) and interactive trace is requested (#Interactive.#Level = ‘1’). Otherwise: #TraceInstruction = ‘1’ value = ’ ’ if #Contains(trace, valueexp) then Value = #Evaluate(valueexp, expression) if #Contains (trace, taken constant) then Value = #Instance (trace, taken constant) if datatype(Value) == ‘NUM’ & (Value,‘W’) then call #Raise ‘SYNTAX’, 26.7, Value if datatype(Value,‘W’) then do /* Numbers are used for skipping. */ *if Value>=0 then #InhibitPauses = Value else #InhibitTrace = -Value end else do if length(Value) = 0 then do #Interactive.#Level = ‘0’ Value = ‘N’ end* / Each question mark toggles the interacting. */ do while left(Value,1)==‘?’ #Interactive.#Level = #Interactive.#Level Value = substr(Value,2) end if length(Value) = 0 then do Value = translate(left(Value,1)) if verify(Value, ‘ACEFILNOR’) > 0 then call #Raise ‘SYNTAX’, 24.1, Value if Value==‘0O’ then #Interactive.#Level=‘0’ end #Tracing.#Level = Value end

11.4.30 Trace output

If #NoSource is ‘1’ there is no trace output.

The routines #TraceSource and #Trace specify the output that results from the trace settings. That output is presented to the configuration by Config_Trace_-Output as lines. Each line has a clause identifier at the left, followed by a blank, followed by a three character tag, followed by a blank, followed by the trace data.

The width of the clause identifier shall be large enough to hold the line number of the last line in the program, and no larger. The clause identifier is the source program line number, or all blank if the line number is the same as the previous line number indicated and no execution with trace Off has occurred since. The line number is right-aligned with leading zeros replaced by blank characters.

When input at a pause is being executed (#AtPause = 0), #Trace does nothing when the tag is not ‘+++’.

When input at a pause is being executed, #TraceSource does nothing. If #InhibitTrace is greater than zero, #TraceSource does nothing except decrement #InhibitTrace. Otherwise, unless the current clause is a null clause, #TraceSource outputs all lines of the source program which contain any part of the current clause, with any characters in those lines which are not part of the current clause and not other_blank_characters replaced by blank characters. The possible replacement of other_blank_characters is defined by the configuration. The tag is ’-”, or if the line is not the first line of the clause. ™,*’. #Trace output also has a clause identifier and has a tag which is the argument to the #Trace invocation. The data is truncated, if necessary, to #Limit_TraceData characters. The data is enclosed by quotation marks and the quoted data preceded by two blanks. If the data is truncated, the trailing quote has the three characters ‘…’ appended. _ when #Tracing.#Level is ‘C’ or ‘E’ or ‘F’ or ‘N’ or ‘A’ and the tag is ‘»>’ then the data is the value of the command passed to the environment; _ when the tag is ‘+++’ then the data is the four characters ‘RC concatenated with the character “ “; _ when #Tracing.#Level is ’I’ or ‘R’ the data is the most recently evaluated value. Trace output can also appear as the result of a’SYNTAX’ condition occurring, irrespective of the trace setting. If a’SYNTAX’ condition occurs and it is not trapped by SIGNAL ON SYNTAX, then the clause in error shall be traced, along with a traceback. A traceback is a display of each active CALL and INTERPRET instruction, and function invocation, displayed in reverse order of execution, each with a tag of ‘+4+’. ### USE For a definition of the syntax of this instruction, see nnn. The USE instruction assigns the values of arguments to variables. Better not say copies since COPY method has different semantics. The optional VAR_SYMBOL positions, positions 1, 2, …, of the instruction are considered from left to right. If the position has a VAR_SYMBOL then its value is assigned to:

if #ArgExists.Position then call Value VAR_SYMBOL, #Arg.Position else

Messy because VALUE bif won’t DROP and var_drop needs to know if compound.

11.5 Conditions and Messages

When an error occurs during execution of a program, an error number and message are associated with it. The error number has two parts, the error code and the error subcode. These are the integer and decimal parts of the error number. Subcodes beginning or ending in zero are not used.

Error codes in the range 1 to 90 and error subcodes up to .9 are reserved for errors described here and for future extensions of this standard.

concatenated with #RC

Error number 3 is available to report error conditions occuring during the initialization phase; error number 2 is available to report error conditions during the termination phase. These are error conditions recognized by the language processor, but the circumstances of their detection is outside of the scope of this standard.

The ERRORTEXT built-in function returns the text as initialized in nnn when called with the 'Standard' option. When the 'Standard' option is omitted, implementation-dependent text may be returned.

When messages are issued any message inserts are replaced by actual values.

The notation for detection of a condition is:

call #Raise Condition, Arg2, Arg3, Arg4, Arg5, Arg6é

Some of the arguments may be omitted. In the case of condition 'SYNTAX' the arguments are the message number and the inserts for the message. In other cases the argument is a further description of the condition.

The action of the program as a result of a condition is dependent on any signa/_-spec and callon_spec in the program.

11.5.1 Raising of conditions

The routine #Raise corresponds to raising a condition. In the following definition, the instructions containing SIGNAL VALUE and INTERPRET denote transfers of control in the program being processed. The instruction EXIT denotes termination. If not at an interactive pause, this will be termination of the program, see nnn, and there will be output by Config_Trace_Output of the message (with prefix _ see nnn) and tracing (see nnn). If at an interactive pause (#AtPause = 0), this will be termination of the interpretation of the interactive input; there will be output by Config_Trace_Output of the message (without traceback) before continuing. The description of the continuation is in nnn after the "interpret #Outcome" instruction.

The instruction "interpret 'CALL' #TrapName.#Condition.#Level" below does not set the variables RESULT and .RESULT; any result returned is discarded.

#Raise: /* If there is no argument, this is an action which has been delayed from the time the condition occurred until an appropriate clause boundary. */ if arg$(1, `E')thendoDescription = \#PendingDescription.\#Condition.\#LevelExtra =$

$\#PendingExtra.\#Condition.\#Level$

end else do #Condition = arg(1) if #Condition == ‘SYNTAX’ then do

Description = arg(2) Extra = arg(3) end else do Description = #Message(arg(2),arg(3),arg(4),arg(5
call Var Set #ReservedPool, ‘.MN’, 0, arg(2) Extra = ’! end end

/* The events for disabled conditions are ignored or cause termination. */

if #Enabling.#Condition.#Level == ‘OFF’ | #AtPause = 0 then do if #Condition == ‘SYNTAX’ & #Condition == ‘HALT’ then return /* To after use of #Raise. / *if #Condition == ‘HALT’ then Description = #Message(4.1, Description) exit /* Terminate with Description as the message. */ end

/* SIGNAL actions occur as soon as the condition is raised. */

if #Instruction.#Condition.#Level == “‘SIGNAL’ then do #ConditionDescription.#Level = Description #ConditionExtra.#Level = Extra #ConditionInstruction.#Level = ‘SIGNAL’ #Enabling.#Condition.#Level = ‘OFF’ signal value #TrapName.#Condition.#Level end

/* All CALL actions are initially delayed until a clause boundary. */

if arg(1,‘E’) then do /* Events within the handler are not stacked up, except for one extra HALT while a first is being handled. / *EventLevel = #Level if #Enabling.#Condition.#Level == ‘DELAYED’ then do if #Condition == ‘HALT’ then return EventLevel = #EventLevel.#Condition. #Level if #PendingNow.#Condition.EventLevel then return* / Setup a HALT to come after the one being handled. */ end /* Record a delayed event. / #PendingNow.#Condition.EventLevel = ‘1’ #PendingDescription.#Condition.Ever = Description #PendingExtra.#Condition.EventLevel = Extra #Enabling.#Condition.EventLevel = ‘DELAYED’ return end* / Here for CALL action after delay. / / Values for the CONDITION built-in function. */ #Condition.#NewLevel = #Condition #ConditionDescription.#NewL = #PendingDescription. #Condition. #Level #ConditionExtra.#NewLevel = #PendingExtra.#Condi #Level #ConditionInstruction.#NewLevel = ‘CALL’ interpret ‘CALL’ #TrapName.#Condition.#Level #Enabling.#Condition.#Level = ‘ON’ return* / To clause termination */

11.5.2 Messages during execution

The state function #Message corresponds to constructing a message.

This definition is for the message text in nnn. Translations in which the message inserts are in a different order are permitted.

In addition to the result defined below, the values of MsgNumber and #LineNumber shall be shown when a message is output. Also there shall be an indication of whether the error occurred in code executed at an interactive pause, see nnn.

Messages are shown by writing them to the default error stream.

#Message: MsgNumber = arg(1) if #NoSource then MsgNumber = MsgNumber % 1 /* And hence no inserts */ Text = #ErrorText.MsgNumber Expanded = ’ ’

do Index = 2 parse var Text Begin ‘<’ Insert ‘>’ +1 Text

if Insert = ’ ’ then leave Insert = arg(Index) if length(Insert) > #Limit MessageInsert then Insert = left(Insert,#Limit MessageInsert)‘…’

```
Expanded = Expanded || Begin || Insert
end
Expanded = Expanded || Begin
say Expanded return
```

12

Built-in functions

12.1 Notation

The built-in functions are defined mainly through code. The code refers to state variables. This is solely a notation used in this standard.

The code refers to functions with names that start with ‘Config_’; these are the functions described in section nnn.

The code is specified as an external routine that produces a result from the values #Bif (which is the name of the built-in function), #Bif_Arg.0 (the number of arguments), #Bif_Arg.i and #Bif_ArgExists.i (which are the argument data.)

The value of #Level is the value for the clause which invoked the built-in function.

The code either returns the result of the built-in or exits with an indication of a condition that the invocation of the built-in raises.

The code below uses built-in functions. Such a use invokes another use of this code with a new value of #Level. On these invocations, the CheckArgs function is not relevant.

Numeric settings as follows are used in the code. When an argument is being checked as a number by ‘NUM’ or ‘WHOLENUM’ the settings are those current in the caller. When an argument is being checked as an integer by an item containing ‘WHOLE’ the settings are those for the particular built-in function. Elsewhere the settings have sufficient numeric digits to avoid values which would require exponential notation.

12.2 Routines used by built-in functions

The routine CheckArgs is concerned with checking the arguments to the built-in. The routines Time2Date and Leap are for date calculations. ReRadix is used for radix conversion. The routine Raise raises a condition and does not return.

12.2.1 Argument checking

/* Check arguments. Some further checks will be made in particular built-ins./ / The argument to CheckArgs is a checklist for the allowable arguments. */

/* NUM, WHOLENUM and WHOLE have a side-effect, ‘normalizing’ the number.

```
*/

/* Calls to raise syntax conditions will not return. */

CheckArgs: CheckList = arg(1) /* This refers to the argument of CheckArgs. */

/* Move the checklist information from a string to individual variables / ArgType. = ' ' ' ArgPos = 0 / To count arguments / MinArgs = 0 do j = 1 to length (CheckList) ArgPos = ArgPos+1 / Count the required arguments. / if substr(CheckList,j,1) == 'r' then MinArgs = MinArgs + 1 / Collect type information. / do while j < length(CheckList) j=eajae1 t = substr(CheckList,j,1) if t==' ' then leave ArgType.ArgPos = ArgType.ArgPos || t end / A single space delimits parts. */ end j MaxArgs = ArgPos

/* Check the number of arguments to the built-in, in this instance. */ NumArgs = #Bif Arg.0

if NumArgs < MinArgs then call Raise 40.3, MinArgs

if NumArgs > MaxArgs then call Raise 40.4, MaxArgs

/* Check the type(s) of the arguments to the built-in. */ do ArgPos = 1 to NumArgs if #Bif ArgExists.ArgPos then call CheckType else if ArgPos <= MinArgs then call Raise 40.5, ArgPos end ArgPos

/* No errors found by CheckArgs. */

return CheckType: Value = #Bif Arg.ArgPos Type = ArgType.ArgPos select when Type == 'ANY' then nop /* Any string / when Type == 'NUM' then do / Any number */

/* This check is made with the caller's digits setting. / if (Value, 'N') then if #DatatypeResult=='E' then call Raise 40.9, ArgPos, Value else call Raise 40.11, ArgPos, Value #Bif Arg.ArgPos=#DatatypeResult / Update argument copy. */ end

when Type == 'WHOLE' then do /* Whole number / / This check is made with digits setting for the built-in. */ if (Value,'W') then call Raise 40.12, ArgPos, Value #Bif_ Arg.ArgPos=#DatatypeResult end

when Type == 'WHOLE>=0' then do /* Non-negative whole number */ if (Value,'W') then call Raise 40.12, ArgPos, Value if #DatatypeResult < 0 then call Raise 40.13, ArgPos, Value #Bif_ Arg.ArgPos=#DatatypeResult end

when Type == 'WHOLE>0O' then do /* Positive whole number */ if (Value,'W') then call Raise 40.12, ArgPos, Value if #DatatypeResult <= 0 then call Raise 40.14, ArgPos, Value #Bif_ Arg.ArgPos=#DatatypeResult end

when Type == 'WHOLENUM' then do /* D2X type whole number / / This check is made with digits setting of the caller. */ if (Value,'W') then call Raise 40.12, ArgPos, Value #Bif_ Arg.ArgPos=#DatatypeResult end

when Type == 'WHOLENUM>=0' then do /* D2X Non-negative whole number */ if (Value,'W') then call Raise 40.12, ArgPos, Value if #DatatypeResult < 0 then call Raise 40.13, ArgPos, Value #Bif_ Arg.ArgPos=#DatatypeResult end

when Type == '0 90' then do /* Errortext */

if (Value,'N') then call Raise 40.11, ArgPos, Value
```

Value=#DatatypeResult

#Bif_ Arg.ArgPos=Value

Major=Value % 1

Minor=Value - Major

if Major < 0 | Major > 90 | Minor > .9 | pos('E',Value)>0 then call Raise 40.17, Value /* ArgPos will be 1 */

end

when Type == 'PAD' then do /* Single character, usually a pad. */

if length(Value) = 1 then call Raise 40.23, ArgPos, Value end

when Type == 'HEX' then /* Hexadecimal string / *if (Value, 'X') then call Raise 40.25, Value* / ArgPos will be 1 / *when Type == 'BIN' then* / Binary string / *if (Value,'B') then call Raise 40.24, Value* / ArgPos will be 1 / *when Type == 'SYM' then* / Symbol */

if (Value, 'S') then call Raise 40.26, Value /* ArgPos will be 1 */

when Type == 'STREAM' then do call Config Stream Qualify Value if left (#Response, 1) == 'B' then call Raise 40.27, Value /* ArgPos will be 1 */ end

when Type = 'ACEFILNOR' then do /* Trace / *Val* = *Value* / Allow '?' alone */

if val == '?' then do /* Allow leading '?' */ if left(Val,1) == '?' then Val = substr(Val,2)

if pos(translate(left(Val, 1)), 'ACEFILNOR') = 0 then call Raise 40.28, ArgPos, Type, Val

end end otherwise do /* Options / / The checklist item is a list of allowed characters */ if Value == ' ' then

call Raise 40.21, ArgPos #Bif Arg.ArgPos = translate(left(Value, 1)) if pos(#Bif Arg.ArgPos, Type) = 0 then call Raise 40.28, ArgPos, Type, Value end

end /* Select */ return

Cdatatype: /* This check is made with the digits setting of the caller. / / #DatatypeResult will be set by use of datatype() */

numeric digits #Digits.#Level

numeric form value #Form.#Level

return datatype(arg(1), arg(2))

Edatatype: /* This check is made with digits setting for the particular built-in. / / #DatatypeResult will be set by use of datatype() */

numeric digits #Bif Digits.#Bif

numeric form scientific

return datatype(arg(1),arg(2))

10.2.2 Date calculations Time2Date: if arg(1) < 0 then call Raise 40.18 if arg(1) >= 315537897600000000 then call Raise 40.18 return Time2Date2 (arg(1))

Time: procedure /* This routine is essentially the code from the standard, put in stand-alone form. The only 'tricky bit' is that there is no Rexx way for it to fail with the same error codes as a "real" implementation would. It can however give

a SYNTAX error, albeit not the desirable one. This causing of an error is done by returning with no value. Since the routine will have been called as a function, this produces an error. */

/* Backslash is avoided as some systems don't handle that negation sign. */ *if argQ)>3 then returm numeric digits 18 if arg(1,'E') then if pos(translate(left(arg(1),1)),"CEHLMNRS then return* / (The standard would also allow 'O' but what this code is running on would not.) */ *if arg(3,'E') then if pos(translate(left(arg(3),1)),"CHLMNS")=0 then return* / Tf the third argument is given then the second is mandatory. */ *if arg(3,'E') & arg(2,'E')=0 then return* / Default the first argument. */ *if arg(1,'E') then Option = translate(left(arg(1),1)) else Option = 'N"* / Tf there is no second argument, the current time is returned. */ *if arg(2,'E') = 0 then if arg(1,'E') then return "TIME'(arg(1)) else return 'TIME'()* / One cannot convert to elapsed times. */ if pos(Option, 'ERO') > 0 then returm InValue = arg(2) if arg(3,'E') then InOption = arg(3) else InOption = 'N' HH =0 MM = 0 SS =0 HourAdjust = 0 select

when InOption == 'C' then do parse var InValue HH ':'. +1 MM +2 XX if HH = 12 then HH =0 if XX == 'pm' then HourAdjust = 12

end when InOption == 'H' then HH = InValue

when InOption == 'L" | InOption == 'N' then parse var InValue HH ':' MM ':' SS when InOption == 'M' then MM = InValue otherwise SS = InValue end if datatype(HH,'W')=0 | datatype(MM,'W')=0 | datatype(SS,'N')=0 then returm HH = HH + HourAdjust /* Convert to microseconds */ *Micro = trunc((((HH* 60) + MM) * 60 + SS) * 1000000) /* There is no special message for time-out-of-range; the bad-format message is used. */ *if Micro<0O | Micro > 24*3600* 1000000 then return /* Reconvert to further check the original. */ if TimeFormat(Micro,InOption) == InValue then return TimeFormat(Micro, Option) return

TimeFormat: procedure /* Convert from microseconds to given format. / / The day will be irrelevant; actually it will be the first day possible. */ x = Time2Date2(arg(1)) parse value x with Year Month Day Hour Minute Second Microsecond Base Days select when arg(2) == 'C' then select when Hour>12 then return Hour-12':'right(Minute,2,'0")'pm' when Hour=12 then return '12:'right(Minute,2,'0')'pm' when Hour>0 then return Hour':'right(Minute,2,'0')'am' when Hour=0 then return '12:'right(Minute,2,'0')'am'

when arg(2) == 'H' then return Hour when arg(2) == 'L' then return right(Hour,?2,'0')':'right(Minute,2 || '.'right(Microsecond,6,'0') when arg(2) == 'M' then return 60*Hour+Minute when arg(2) == 'N' then return right(Hour,?2,'0')':'right(Minute,2,'0')':'right(Second,2,'0') otherwise* / arg(2) =='S' */ *return 3600*Hour+60* Minute+Second end

Time2Date2: Procedure /* Convert a timestamp to a date. Argument is a timestamp (the number of microseconds relative to 0001 01 01 00:00:00.000000) Returns a date in the form: year month day hour minute second microsecond base days */

/* Argument is relative to the virtual date 0001 01 01 00:00:00.000000 */ Time = arg(1)

Second = Time % 1000000 } Microsecond = Time // 1000000 Minute = Second % 60 ; Second = Second // 60 Hour = Minute % 60 ; Minute = Minute // 60 Day

= Hour % 24 ;} Hour = Hour // 24

/* At this point, the days are the days since the 0001 base date. */ BaseDays = Day Day = Day + 1

/* Compute either the fitting year, or some year not too far earlier. Compute the number of days left on the first of January of this year. */

Year = Day % 366 Day = Day - (Year*365 + Year%4 - Year%100 + Year%400) Year = Year +1

/* Now if the number of days left is larger than the number of days in the year we computed, increment the year, and decrement the number of days accordingly. */ do while Day > (365 + Leap (Year))

Day = Day - (365 + Leap(Year))

Year = Year + 1 end

/* At this point, the days left pertain to this year. */ YearDays = Day

/* Now step through the months, increment the number of the month, and decrement the number of days accordingly (taking into consideration that in a leap year February has 29 days), until further reducing the number of days and incrementing the month would lead to a negative number of days */ Days = '31 28 31 30 31 30 31 31 30 31 30 31' do Month = 1 to words (Days)

ThisMonth = Word(Days, Month) + (Month = 2) * Leap (Year)

if Day <= ThisMonth then leave

Day = Day - ThisMonth end

return Year Month Day Hour Minute Second Microsecond BaseDays YearDays

Leap: procedure /* Return 1 if the year given as argument is a leap year, or 0 otherwise. */ return (arg(1)//4 = 0) & ((arg(1)//100 <> 0) | (arg(1)//400 = 0))

10.2.1. Radix conversion

ReRadix: /* Converts Arg(1) from radix Arg(2) to radix Arg(3) */

procedure

Subj ect=arg(1)

FromRadix=arg (2)

ToRadix=arg (3)

/* Radix range is 2-16. Conversion is via decimal */

Integer=0

do j=1 to length (Subject) /* Individual digits have already been checked for range. */ *Integer=Integer*FromRadix+pos (substr (Subject,j,1),'0123456789ABCDEF')-1 end

rete

do while Integer>0

r= substr('0123456789ABCDEF',1 + Integer // ToRadix, 1) || r Integer = Integer % ToRadix end

/* When between 2 and 16, there is no zero suppression. */ *if FromRadix = 2 &*

ToRadix = 16 then r=eright(r, (length(Subject)+3) % 4, '0') else if FromRadix = 16 & ToRadix = 2 then reright(r, length(Subject) 4, '0') return r

12.2.2 Raising the SYNTAX condition

Raise:

/* These 40.nn messages always include the built-in name as an insert./ *call #Raise 'SYNTAX', arg(1), #Bif, arg(2), arg(3), arg(4) /* #Raise does not return. */

12.3 Character built-in functions

These functions process characters or words in strings. Character positions are numbered from one at the left. Words are delimited by blanks and their equivalents, word positions are counted from one at the left.

12.3.1 ABBREV

ABBREV returns '1' if the second argument is equal to the leading characters of the first and the length of the second argument is not less than the third argument.

call CheckArgs 'rANY rANY oWHOLE>=0'

Subject #Bif Arg.1

Subj #Bif Arg.2

if #Bif_ArgExists.3 then Length = #Bif Arg.3 else Length = length (Subj)

Condl = length(Subject) >= length(Subj) Cond2 = length(Subj) >= Length Cond3 = substr(Subject, 1, length(Subj)) == Subj

return Condl & Cond2 & Cond3

12.3.2 CENTER

CENTER returns a string with the first argument centered in it. The length of the result is the second argument and the third argument specifies the character to be used for padding.

call CheckArgs 'rANY rWHOLE>=0 oPAD'

String = #Bif Arg.1

Length = #Bif Arg.2

if #Bif_ArgExists.3 then Pad = #Bif Arg.3 else Pad = ' '!

Trim = length(String) - Length

if Trim > 0 then return substr(String, Trim % 2 + 1, Length)

return overlay(String, copies(Pad, Length), -Trim % 2 + 1)

12.3.3 CENTRE

This is an alternative spelling for the CENTER built-in function.

12.3.4 CHANGESTR

CHANGESTR replaces all occurrences of the first argument within the second argument, replacing them with the third argument.

call CheckArgs “‘rANY rANY rANY’

Output = ’ ’ Position = 1 do forever FoundPos = pos(#Bif Arg.1, #Bif Arg.2, Position) if FoundPos = 0 then leave Output = Output || substr(#Bif_ Arg.2, Position, FoundPos - Position), || #Bif Arg.3 Position = FoundPos + length(#Bif Arg.1) end return Output || substr(#Bif Arg.2, Position)

12.3.5 COMPARE

COMPARE returns ‘0’ if the first and second arguments have the same value. Otherwise, the result is the position of the first character that is not the same in both strings.

call CheckArgs ‘rANY rANY oPAD’

Strl = #Bif_Arg.1

Str2 = #Bif_Arg.2

if #Bif_ArgExists.3 then Pad else Pad

#Bif Arg.3

/* Compare the strings from left to right one character at a time */ if length(Str1) > length(Str2) then do

Length = length(Str1)

Str2=left (Str2, Length, Pad)

end else do

Length = length(Str2)

Strl=left (Str1, Length, Pad)

end

do i= 1 to Length if substr(Stri, i, 1) == substr(Str2, i, 1) then return i end

return 0

12.3.6 COPIES

COPIES returns concatenated copies of the first argument. The second argument is the number of copies.

call CheckArgs “‘rANY rWHOLE>=0’ Output = ’ ’ do #Bif Arg.2 Output = Output || #Bif_Arg.1 end

return Output

12.3.7 COUNTSTR

COUNTSTR counts the appearances of the first argument in the second argument.

call CheckArgs “‘rANY rANY’

Output = 0

Position = pos (#Bif Arg.1,#Bif Arg.2)

do while Position > 0 Output = Output + 1 Position = pos(#Bif Arg.1, #Bif Arg.2, Position + length(#Bif Arg.1)) end

return Output

12.3.8 DATATYPE

DATATYPE tests for characteristics of the first argument. The second argument specifies the particular test.

call CheckArgs ‘rANY oABLMNSUWX’

/* As well as returning the type, the value for a ‘NUM’ is set in #DatatypeResult. This is a convenience when DATATYPE is used

by CHECKARGS. */

String = #Bif Arg.1

/* If no second argument, DATATYPE checks whether the first is a number. */ if #Bif_ArgExists.2 then return DtypeOne()

Type = #Bif_ Arg.2 /* Null strings are a special case. */

if String == ’ ’ then do if Type == “X” then return 1 if Type == “B” then return 1 return 0 end

/* Several of the options are shorthands for VERIFY */ *azl=“abcdefghijklmnopqrstuvwxyz” AZU= “ABCDEFGHIJKLMNOPQRSTUVWXYZ” DO09=“0123456789” if Type == “A” then return verify(String,az1||AzU||D09)=0 if Type == “B” then do /* Check blanks in allowed places. */ if pos (left (String,1),#Al1Blanks)>0 then return 0 if pos (right (String,1),#Al1Blanks)>0 then return 0 BinaryDigits=0 do j = length(String) by -1 tol ce = substr(String,j,1) if pos(c,#Al11Blanks)>0 then do */* Blanks need four BinaryDigits to the right of them. */ if BinaryDigits//4 = 0 then return 0 end else do if verify(c,“01”) = 0 then return 0 BinaryDigits = BinaryDigits + 1

end end j

return 1

end /* B */ if Type == “L” then return (verify (String,azl1)=0) if Type == “M” then return (verify (String, azl1||AZU)=0) if Type == “N” then return (datatype (String) ==“NUM”) if Type == “S” then return(symbol (String) ==‘BAD’) if Type == “U” then return (verify (String, AZU)=0) if Type == “W” then do

/* It may not be a number. */ *if DtypeOne(String) == ‘CHAR’ then return ‘0’* / It can be “Whole” even if originally in exponential notation, provided it can be written as non-exponential. */ *if pos(‘E’,#DatatypeResult)>0 then return ‘0’* / It won’t be “Whole” if there is a non-zero after the decimal point. */ InFraction=‘0’

do j = 1 to length (String) ce = substr(String,j,1) if pos(c,‘Ee’)>0 then leave j if InFraction & pos(c,‘+-’)>0 then leave j if c == ‘.’ then InFraction=‘1’ else if InFraction & a==‘0’ then return 0 end j /* All tests for Whole passed. */ *#DatatypeResult = #DatatypeResult % 1 return 1 end* / W / / Type will be “x” */ *if pos (left (String,1),#Al1Blanks)>0 then return 0 if pos (right (String,1),#Al1Blanks)>0 then return 0 HexDigits=0 do j=length(String) by -1 to 1 c=substr (String,j,1) if pos(c,#Al11Blanks)>0 then do* / Blanks need a pair of HexDigits to the right of them. */ *if HexDigits//2 = 0 then return 0 end else do if verify(c,“abcdefABCDEF”D09) = 0 then return 0 HexDigits=HexDigits+1 end end return 1* / end X */

DtypeOne: /* See section nnn for the syntax of a number. */ *#DatatypeResult = ‘S’* / If not syntactically a number */ *Residue = strip(String)* / Blanks are allowed at both ends. */ if Residue == ’ ’ then return “CHAR” Sign = ’ ’ if left(Residue,1) == ‘+’ | left(Residue,1) == ‘-’ then do

Sign = left(Residue, 1) Residue = strip(substr(Residue,2),‘L’) /* Blanks after sign */

end if Residue == ’ ’ then return “CHAR” /* Now testing Number, section nnn */ if left(Residue,1) == ‘.’ then do Residue = substr (Residue, 2) Before = ’ ’ After = DigitRun() if After == ’ ’ then return “CHAR” end else do Before = DigitRun() if Before == ’ ’ then return “CHAR” if left(Residue,1) == ‘.’ then do Residue = substr (Residue, 2) After = DigitRun() end end Exponent = 0 if Residue == ’ ’ then do if left(Residue, 1) == ‘e’ & left(Residue, 1) == ‘E’ then

return “CHAR” Residue = substr (Residue, 2)

if Residue == ’ ’ then return “CHAR”

Esign = ’ ’

if left(Residue, 1) == ‘+’ | left(Residue, 1) == ‘-’ then do Esign = left(Residue, 1) Residue = substr (Residue, 2) if Residue == ’ ’ then return “CHAR” end

Exponent = DigitRun()

if Exponent == ’ ’ then return “CHAR”

Exponent = Esign || Exponent

end

if Residue == ’ ’ then return “CHAR”

102 */DATATYPE tests for exponent out of range. /*

#DatatypeResult = ‘E’ /* If exponent out of range */ Before = strip(Before,‘L’,‘0’) if Before == ’ ’ then Before = ‘0’

Exponent = Exponent + length(Before) -1 /* For SCIENTIFIC */

/* “Engineering notation causes powers of ten to expressed as a multiple of 3 - the integer part may therefore range from 1 through 9910.” */ *g=1 if #Form.#Level == ‘E’ then do* / Adjustment to make exponent a multiple of 3 */ g = Exponent//3 if g < 0 then g = g + Exponent = Exponent - end

/* Check on the exponent. */ if Exponent > #Limit ExponentDigits then return “CHAR” if -#Limit ExponentDigits > Exponent then return “CHAR”

/* Format to the numeric setting of the caller of DATATYPE */ numeric digits #Digits.#Level

numeric form value #Form.#Level

#DatatypeResult = 0 + #Bif_Arg.1

return “NUM”

DigitRun: Outcome = ’ ’ do while Residue == ’ ’ if pos(left (Residue, 1), ‘0123456789’) = 0 then leave

Outcome = Outcome || left(Residue, 1) Residue = substr(Residue, 2) end

return Outcome

12.3.9 DELSTR

DELSTR deletes the sub-string of the first argument which begins at the position given by the second argument. The third argument is the length of the deletion.

call CheckArgs ‘rANY rWHOLE>0 oWHOLE>=0’ String #Bif Arg.1

Num #Bif Arg.2 if #Bif_ArgExists.3 then Len = #Bif_Arg.3

if Num > length(String) then return String

Output = substr(String, 1, Num - 1) if #Bif_ArgExists.3 then if Num + Len <= length(String) then Output = Output || substr(String, Num + Len) return Output

12.3.10 DELWORD

DELWORD deletes words from the first argument. The second argument specifies position of the first word to be deleted and the third argument specifies the number of words.

call CheckArgs ‘rANY rWHOLE>0 oWHOLE>=0’ String #Bif Arg.1

Num #Bif Arg.2 if #Bif_ArgExists.3 then Len = #Bif_Arg.3

if Num > words(String) then return String

EndLeft = wordindex(String, Num) - 1 Output = left(String, EndLeft) if #Bif_-ArgExists.3 then do BeginRight = wordindex(String, Num + Len)

if BeginRight>0 then Output =

end return Output

12.3.11 INSERT

Output || substr(String, BeginRight)

INSERT insets the first argument into the second. The third argument gives the position of the character before the insert and the fourth gives the length of the insert. The fifth is the padding character.

call CheckArgs

New #Bif_ Arg.1 Target #Bif Arg.2 if #Bif_ArgExists.3

then else then else then else

Num Num Length = #Bif Arg.4 Length = length (New) Pad #Bif Arg.5 Pad ro

#Bif Arg.3 0

if #Bif_ArgExists.4

if #Bif_ArgExists.5

return left(Target, Num, Pad) left (New, Length, Pad), substr(Target, Num + 1)

“

12.3.12 LASTPOS

/* To left of insert /* New string inserted / / To right of insert

‘rANY rANY oWHOLE>=0 oWHOLE>=0 oPAD’

/ /

LASTPOS returns the position of the last occurrence of the first argument within the second. The third

argument is a starting position for the search.

call CheckArgs ‘rANY rANY oWHOLE>0’

Needle = #Bif_Arg.1 Haystack = #Bif Arg.2 if #Bif_ArgExists.3 then Start = #Bif_-Arg.3 else Start = length(Haystack)

NeedleLength = length (Needle) if NeedleLength = 0 then return 0 Start = Start - NeedleLength + 1 do i= Start by -1 while i > 0 if substr(Haystack, i, NeedleLength) end i return 0

12.3.13 LEFT

LEFT returns characters that are on the left of the first argument.

length of the result and the third is the padding character. call CheckArgs ‘rANY rWHOLE>=0 oPAD’

if #Bif_ArgExists.3 then Pad else Pad

#Bif Arg.3

return substr(#Bif Arg.1, 1, #Bif_ Arg.2, Pad)

10.1.14 LENGTH

Needle then return i

The second argument specifies the

Length returns a count of the number of characters in the argument.

call CheckArgs ‘rANY’

String = #Bif Arg.1

#Response = Config Length(String)

Length = #Outcome

call Config Substr #Response, 1

if #Outcome == ‘E’ then return Length

/* Here if argument was not a character string. call Config C2B String call #Raise ‘SYNTAX’,

23.1, b2x(#Outcome)

/ / No return to here */

12.3.14 OVERLAY

OVERLAY overlays the first argument onto the second. The third argument is the starting position of the

overlay. The fourth argument is the length of the overlay and the fifth is the padding character.

call CheckArgs ‘rANY rANY oWHOLE>0 oOWHOLE>=0 oPAD’

New = #Bif_Arg.1

Target = #Bif Arg.2

if #Bif_ArgExists.3 then Num = #Bif_Arg.3 else Num = 1

if #Bif_ArgExists.4 then Length = #Bif_ Arg.4 else Length = length (New)

if #Bif_ArgExists.5 then Pad = #Bif_ Arg.5 else Pad = ’ ’ return left(Target, Num - 1, Pad), /* To left of overlay / || *left (New, Length, Pad),* / New string overlaid / || *substr (Target, Num + Length)* / To right of overlay */

10.1.16 POS POS returns the position of the first argument within the second.

call CheckArgs ‘rANY rANY oWHOLE>0’

Needle #Bif Arg.1

Haystack = #Bif Arg.2

if #Bif_ ArgExists.3 then Start else Start

#Bif Arg.3 1

if length(Needle) = 0 then return 0

do i = Start to length (Haystack) +1-length (Needle) if substr(Haystack, i, length(Needle)) == Needle then return i end i

return 0

12.3.15 REVERSE

REVERSE returns its argument, swapped end for end.

call CheckArgs ‘rANY’

String

#Bif Arg.1 Output = ’ ’ do i= 1 to length (String) Output = substr(String,i,1) || Output end return Output

12.3.16 RIGHT

RIGHT returns characters that are on the right of the first argument. The second argument specifies the

length of the result and the third is the padding character. call CheckArgs ‘rANY rWHOLE>=0 oPAD’

String = #Bif Arg.1 Length = #Bif Arg.2 if #Bif_ArgExists.3 then Pad

#Bif Arg.3 else Pad ’

Trim = length(String) - Length if Trim >= 0 then return substr(String,Trim + 1) return copies(Pad, -Trim) || String /* Pad string on the left */

12.3.17 SPACE

SPACE formats the blank-delimited words in the first argument with pad characters between each word. The second argument is the number of pad characters between each word and the third is the pad character.

call CheckArgs ‘rANY oOWHOLE>=0 oPAD’

String = #Bif Arg.1

if #Bif_ArgExists.2 then Num = #Bif_ Arg.2 else Num = 1 if #Bif ArgExists.3 then Pad = #Bif Arg.3 else Pad = ’ ’! Padding = copies(Pad, Num) Output = subword(String, 1, 1) do i = 2 to words (String) Output = Output || Padding || subword(String, i, 1)

end return Output

12.3.18 STRIP

STRIP removes characters from its first argument. The second argument specifies whether the deletions are leading characters, trailing characters or

both. Each character deleted is equal to the third argument, or equivalent to a blank if the third argument is omitted.

call CheckArgs ‘rANY oLTB oPAD’

String = #Bif Arg.1 if #Bif_ArgExists.2 then Option = #Bif Arg.2 else Option = ‘B’ if #Bif_ ArgExists.3 then Unwanted = #Bif_ Arg.3 else Unwanted = #A11Blanks<Index “#Al11Blanks” # “ ” >

if Option == ‘L’ | Option == ‘B’ then do /* Strip leading characters */ do while String == ’ ’ & pos(left(String, 1), Unwanted) > 0 String = substr(String, 2) end end

if Option == ‘T’ | Option == ‘B’ then do /* Strip trailing characters */ *do while String == ’ ’ & pos(right(String, 1), Unwanted) > 0 String = left(String, length(String) -1) end* / of while */ end return String

12.3.19 SUBSTR

SUBSTR returns a sub-string of the first argument. The second argument specifies the position of the first character and the third specifies the length of the sub-string. The fourth argument is the padding

character.

call CheckArgs ‘rANY rWHOLE>0 oOWHOLE>=0 oPAD’

String = #Bif Arg.1 Num = #Bif_ Arg.2 if #Bif_ArgExists.3 then Length = #Bif Arg.3 else Length = max(length (String) +1-Num, 0) if #Bif_ArgExists.4 then Pad = #Bif_ Arg.4 else Pad = ’ ‘! Output =” do Length #Response Config Substr(String,Num) /* Attempt to fetch character.*/ Character #Outcome

Num = Num + 1 call Config Substr #Response,1 /* Was there such a character? */ if #Outcome == ‘E’ then do

/* Here if argument was not a character string. */

call Config C2B String call #Raise ‘SYNTAX’, 23.1, b2x(#Outcome) /* No return to here */

end if #Outcome == ‘M’ then Character = Pad Output=Output | | Character end

return Output

12.3.20 SUBWORD

SUBWORD returns a sub-string of the first argument, comprised of words. The second argument is the position in the first argument of the first word of the sub-string. The third argument is the number of words in the sub-string.

call CheckArgs ‘rANY rWHOLE>0 oWHOLE>=0’

String #Bif Arg.1

Num #Bif Arg.2

if #Bif_ ArgExists.3 then Length else Length

#Bif_Arg.3 length(String) /* Avoids call / / to WORDS() */

if Length = 0 then return ' '

/* Find position of first included word */

Start = wordindex (String, Num)

if Start = 0 then return ' ' /* Start is beyond end */

/* Find position of first excluded word */ End = wordindex (String, Num+Length) if End = 0 then End = length(String)+1

Output=substr (String, Start, End-Start)

/* Drop trailing blanks */

do while Output == ' ' if pos (right (Output,1),#A11Blanks) = 0 then leave Output = left (Output, length (Output) -1) end

return Output

12.3.21 TRANSLATE

TRANSLATE returns the characters of its first argument with each character either unchanged or translated to another character.

call CheckArgs 'rANY oANY oANY oPAD' String = #Bif Arg.1 /* If neither input nor output tables, uppercase. */ if #Bif ArgExists.2 & #Bif_ArgExists.3 then do Output = ' ' do j=1 to length (String) #Response = Config Upper (substr (String,j,1))

Output = Output || #Outcome end j

return Output

end

/* The input table defaults to all characters. */ *if #Bif ArgExists.3 then do #Response = Config Xrange() Tablei = #Outcome end else Tablei = #Bif_Arg.3* / The output table defaults to null / *if #Bif_ArgExists.2 then Tableo = #Bif Arg.2 else Tableo = ' '* / The tables are made the same length */ if #Bif_ArgExists.4 then Pad = #Bif_ Arg.4 else Pad = ' ' Tableo=left (Tableo, length (Tablei) , Pad)

107 Output=' ' do j=1 to length (String) c=substr (String,j,1) k=pos(c,Tablei) if k=0 then Output=Output ||c else Output=Output | | substr (Tableo,k,1) end j return Output

12.3.22 VERIFY

VERIFY checks that its first argument contains only characters that are in the second argument, or that it contains no characters from the second argument; the third argument specifies which check is made. The result is '0', or the position of the character that failed verification. The fourth argument is a starting position for the check.

call CheckArgs 'rANY rANY oMN oWHOLE>0'

String = #Bif Arg.1

Reference = #Bif_Arg.2

if #Bif_ ArgExists.3 then Option #Bif Arg.3 else Option 'N!

if #Bif_ ArgExists.4 then Start = #Bif_Arg.4 else Start = 1

Last = length(String) if Start > Last then return 0 if Reference == ' ' then if Option == 'N' then return Start else return 0

do i = Start to Last t = pos(substr(String, i, 1), Reference) if Option == 'N' then do if t = 0 then return i /* Return position of NoMatch character. */ *end else if t > 0 then return i* / Return position of Matched character. */ end i return 0

12.3.23 WORD

WORD returns the word from the first argument at the position given by the second argument.

call CheckArgs 'rANY rwWHOLE>0'

return subword(#Bif Arg.1, #Bif_Arg.2, 1)

12.3.24 WORDINDEX

WORDINDEX returns the character position in the first argument of a word in the first argument. The second argument is the word position of that word.

call CheckArgs 'rANY rwWHOLE>0'

String Num

#Bif Arg.1 #Bif Arg.2

/* Find starting position */

Start = 1 Count = 0 do forever Start = verify(String, #Al1Blanks<Index "#A11Blanks" # " " >, 'N', Start) / *Find non-blank* / if Start = 0 then return 0 /* Start is beyond end / *Count = Count + 1* / Words found / *if Count = Num then leave Start = verify(String, #Al1Blanks<Index "#Al11Blanks" # " " >, 'M', Start + 1) /*

Find blank */

if Start = 0 then return 0 /* Start is beyond end */ end return Start

12.3.25 WORDLENGTH

WORDLENGTH returns the number of characters in a word from the first argument. The second argument is the word position of that word.

call CheckArgs 'rANY rwWHOLE>0'

return length(subword(#Bif Arg.1, #Bif_Arg.2, 1))

12.3.26 WORDPOS

WORDPOS finds the leftmost occurrence in the second argument of the sequence of words in the first argument. The result is ‘0’ or the word position in the second argument of the first word of the matched sequence. Third argument is a word position for the start of the search.

call CheckArgs ‘rANY rANY oWHOLE>0’

Phrase = #Bif_Arg.1

String = #Bif Arg.2

if #Bif_ArgExists.3 then Start = #Bif_Arg.3 else Start = 1

Phrase = space (Phrase) PhraseWords = words (Phrase) if PhraseWords = 0 then return 0 String = space (String) StringWords = words (String) do WordNumber = Start to StringWords - PhraseWords + 1 if Phrase == subword(String, WordNumber, PhraseWords) then return WordNumber end WordNumber return 0

12.3.27 WORDS

WORDS counts the number of words in its argument.

call CheckArgs ‘rANY’

do Count = 0 by 1 if subword(#Bif Arg.1, Count + 1) == ’ ’ then return Count end Count

12.3.28 XRANGE

XRANGE returns an ordered string of all valid character encodings in the specified range.

call CheckArgs ‘oPAD oPAD’

if #Bif_ArgExists.1 then #Bif_Arg.1 mr if #Bif_ArgExists.2 then #Bif Arg.2 #Response = Config Xrange(#Bif Arg.1, #Bif Arg.2) return #Outcome

12.4 Arithmetic built-in functions

These functions perform arithmetic at the numeric settings current at the invocation of the built-in function. Note that CheckArgs formats any ‘NUM’ (numeric) argument.

12.4.1 ABS

ABS returns the absolute value of its argument.

call CheckArgs ‘rNUM’ Number=#Bif Arg.1

if left (Number,1) = ‘-’ then Number = substr (Number, 2) return Number

12.4.2 FORMAT

FORMAT formats its first argument. The second argument specifies the number of characters to be used for the integer part and the third specifies the number of characters for the decimal part. The fourth argument specifies the number of characters for the exponent and the fifth determines when exponeniial notation is used.

call CheckArgs, ‘rNUM OWHOLE>=0 OWHOLE>=0 OWHOLE>=0 OWHOLE>=0’

if #Bif_ArgExists.2 then Before #Bif Arg.2 if #Bif_ArgExists.3 then After #Bif Arg.3 if #Bif_ArgExists.4 then Expp #Bif Arg.4

if #Bif_ArgExists.5 then Expt = #Bif Arg.5

/* In the simplest case the first is the only argument. */ Number=#Bif Arg.1

if #Bif_Arg.0 < 2 then return Number

/* Dissect the Number. It is in the normal Rexx format. */ parse var Number Mantissa ‘E’ Exponent

if Exponent == ’ ’ then Exponent = 0 Sign = 0 if left (Mantissa,1) == ‘-’ then do Sign = 1 Mantissa = substr(Mantissa, 2) end parse var Mantissa Befo ‘.’ Afte

/* Count from the left for the decimal point. */

Point = length (Befo)

/* Sign Mantissa and Exponent now reflect the Number. Befo Afte and Point reflect Mantissa. */

/* The fourth and fifth arguments allow for exponential notation. / / Decide whether exponential form to be used, setting ShowExp. */ *ShowExp = 0 if #Bif_-ArgExists.4 #Bif ArgExists.5 then do if #Bif_ArgExists.5 then Expt = #Digits.#Level* / Decide whether exponential form to be used. */ *if (Point + Exponent) > Expt then ShowExp = 1* / Digits before rule. */ *LeftOfPoint = 0 if length(Befo) > 0 then LeftOfPoint = Befo* / Value left of the point */

/* Digits after point rule for exponentiation: */

/* Count zeros to right of point. */

ze=O0

do while substr(Afte,z+1,1) == ‘0’ Zeze+41tl end

if LeftOfPoint = 0 & (z - Exponent) > 5 then ShowExp = 1

/* An extra rule for exponential form: */ if #Bif_ArgExists.4 then if Expp = 0 then ShowExp = 0

/* Construct the exponential part of the result. */ if ShowExp then do

Exponent = Exponent + (Point - 1) Point = 1 /* As required for ’SCIENTIFIC! */ if #Form.#Level == ‘ENGINEERING’ then do while Exponent//3 = 0

Point = Point+1l

Exponent = Exponent-1

end end

if then Point = Point + Exponent end /* Expp or Expt given / *else do* / Even if Expp and Expt are not given, exponential notation will be used if the original Number+0 done by CheckArgs led to it. */ if Exponent = 0 then do ShowExp = 1

110 111

end end

/* ShowExp now indicates whether to show an exponent, Exponent is its value. / / Make this a Number without a point.

Integer = Befo||Afte

*/

/* Make sure Point position isn't disjoint from Integer. / *if Point<1 then do* / Extra zeros on the left. */

Integer = Point = 1 end

copies('0',1 - Point)

if Point > length(Integer) then Integer = left(Integer,Point,'0') /* And maybe on the right.

/* Deal with right of decimal point first since that can affect the

|| Integer

left. Ensure the requested number of digits there. Afters = length(Integer) -Point

if #Bif_ArgExists.3 = 0 then After = /* Make Afters match the requested After */

do while Afters < After Afters = Afters+1 Integer = Integer'0' end

if Afters > After then do

/* Round by adding 5 at the right place.

Afters

r=substr (Integer, Point + After + 1,

Integer = left (Integer,

1)

Point + After)

if r >= '5' then Integer = Integer + 1 /* This can leave the result zero. If Integer = 0 then Sign = 0

/* The case when rounding makes the integer longer is an awkward

*/

*/

one. The exponent will have to be adjusted. */ if length(Integer) > Point + After then do

Point = Point+l end

```
if ShowExp = 1 then do
Exponent=Exponent + (Point - 1)
Point = 1 /* As required for 'SCIENTIFIC! */
if form() = 'ENGINEERING' then do while Exponent//3 = 0
Point = Point+
1
Exponent = Exponent-1
end end
t = Point-length (Integer) if t > 0 then Integer = Integer||copies('0',t)
end /* Rounded */
/* Right part is final if After > 0 then Afte else Afte
now. */ '.'| |substr (Integer, Point+1,After)
/* Now deal with the integer part of the result. Integer = left (Integer, Point)
if #Bif ArgExists.2 =
0 then Before
/* Make Point match Before */ if Point > Before - Sign then call Raise
do while Point<Before Point = Point+1l Integer = '0'Integer end
40.38,
*/
Point + Sign /* Note default.
2,
/* Find the Sign position and blank leading zeroes.
re ter Triggered = 0
do j = 1 to length (Integer) Digit = substr(Integer,j,1) /* Triggered is set when sign inserted or blanking finished. if Triggered = 1 then do
r= r||Daigit iterate end
*/
/* Note default.
#Bif Arg.1
*/
*/
*/
*/
/ / If before sign insertion point then blank out zero. */
if Digit = '0' then if ats cael = '0' & j+l<length(Integer) then do re r ' ' iterate end /* j is the sign insertion point. */ if Digit = '0' & j = length(Integer) then Digit = ' '
if Sign = 1 then Digit = '-' r= xr||Digit Triggered = 1 end j Number = r||Afte
```

if ShowExp = 1 then do /* Format the exponent. / *Expart = ' ' SignExp = 0 if Exponent<0 then do SignExp = 1 Exponent = -Exponent end* / Make the exponent to the requested width. */ if #Bif_ ArgExists.4 = 0 then Expp = length (Exponent) if length(Exponent) > Expp then call Raise 40.38, 4, #Bif_ Arg.1 Exponent=right (Exponent,Expp,'0') if Exponent = 0 then do if #Bif_ ArgExists.4 then Expart = copies(' ',expp+2) end else if SignExp = 0 then Expart else Expart Number = Number | |Expart end return Number

'E+' Exponent 'E-' Exponent

12.4.3 MAX

MAX returns the largest of its arguments.

if #Bif_Arg.0 <1 then call Raise 40.3, 1 call CheckArgs 'rNUM'||copies(' rNUM', #Bif Arg.0 - 1)

Max = #Bif Arg.1

do i = 2 to #Bif Arg.0 by 1 Next = #Bif Arg.i if Max < Next then Max = Next end i

return Max

12.4.4 MIN

MIN returns the smallest of its arguments.

if #Bif_Arg.0 <1 then call Raise 40.3, 1 call CheckArgs 'rNUM'||copies(' rNUM', #Bif Arg.0 - 1)

Min = #Bif Arg.1

do i = 2 to #Bif Arg.0 by 1 Next = #Bif Arg.i if Min > Next then Min = Next end i

return Min

12.4.5 SIGN

SIGN returns '1', '0' or '-1' according to whether its argument is greater than, equal to, or less than zero.

call CheckArgs 'rNUM'

Number = #Bif Arg.1

select

when Number < 0 then Output = -1 when Number = 0 then Output = 0 when Number > 0 then Output = 1

end return Output

12.4.6 TRUNC

TRUNC returns the integer part of its argument, or the integer part plus a number of digits after the decimal point, specified by the second argument.

call CheckArgs ‘rNUM oWHOLE>=0’

Number = #Bif Arg.1 if #Bif_ArgExists.2 then Num

#Bif Arg.2 else Num 0

Integer =(10**Num * Number) %1 if Num=0 then return Integer

t=length (Integer) -Num if t<=0 then return ‘0.’right(Integer,Num,’0’) else return insert(‘.’,Integer,t)

12.5 State built-in functions

These functions return values from the state of the execution.

12.5.1 ADDRESS

ADDRESS returns the name of the environment to which commands are currently being submitted. Optionally, under control by the argument, it also returns information on the targets of command output and the source of command input.

call CheckArgs ‘oEINO’

if #Bif_ArgExists.1 then Optionl = #Bif_Arg.1 else Optionl=‘N’

if Optionl == ‘N’ then return #Env_Name.ACTIVE. #Level

Tail = Optionl’ .ACTIVE. ’#Level return #Env_Position.Tail #Env_Type.Tail #Env_-Resource.Tail

12.5.2 ARG

ARG returns information about the argument strings to a program or routine, or the value of one of those strings.

ArgData = ‘OWHOLE>0 oENO’ if #Bif_ ArgExists.2 then ArgData = ‘rWHOLE>0 rENO’ call CheckArgs ArgData

if #Bif_ArgExists.1 then return #Arg.#Level.0

ArgNum=#Bif Arg.1

if #Bif_ArgExists.2 then return #Arg.#Level.ArgNum

if #Bif_Arg.2 ==‘0O’ then return #ArgExists.#Level.ArgNum else return #ArgExists.#Level .ArgNum

12.5.3 CONDITION

CONDITION returns information associated with the current condition.

call CheckArgs ‘oCDEIS’

/* Values are null if this is not following a condition. */ if #Condition.#Level == ’ ’ then do #ConditionDescription.#Level = ’ ’ #ConditionExtra.#Level = ’ ’ #ConditionInstruction.#Level end

Option=#Bif Arg.1

if Option==‘C’ then return #Condition.#Level

if Option==‘D’ then return #ConditionDescription. #Level if Option==‘E’ then return #ConditionExtra.#Level

if Option==‘I’ then return #ConditionInstruction. #Level /* State is the current state. */

if #Condition.#Level = ’ ’ then return “ ” return #Enabling.#Condition.#Level ### DIGITS

DIGITS returns the current setting of NUMERIC DIGITS. call CheckArgs ’ ’

return #Digits.#Level

12.5.4 ERRORTEXT

ERRORTEXT returns the unexpanded text of the message which is identified by the first argument. A second argument of ‘S’ selects the standard English text, otherwise the text may be translated to another national language. This translation is not shown in the code below.

call CheckArgs ‘r0_90 oSN’

msgcode = #Bif Arg.1

if #Bif_ ArgExists.2 then Option else Option

return #ErrorText .msgcode

12.5.5 FORM

FORM returns the current setting of NUMERIC FORM.

#Bif_Arg.2 tint

call CheckArgs ’ ’

return #Form.#Level

12.5.6 FUZZ

FUZZ returns the current setting of NUMERIC FUZZ.

call CheckArgs ’ ’

return #Fuzz.#Level

12.5.7 SOURCELINE

If there is no argument, SOURCELINE returns the number of lines in the program, or ‘0’ if the source program is not being shown on this execution. If there is an argument it specifies the number of the line of the source program to be returned.

call CheckArgs ‘oWHOLE>0’

if #Bif_ArgExists.1 then return #SourceLine.0 Num = #Bif_Arg.1 if Num > #SourceLine.0O then call Raise 40.34, Num, #SourceLine.0 return #SourceLine.Num

12.5.8 TRACE

TRACE returns the trace setting currently in effect, and optionally alters the setting.

call CheckArgs ‘oACEFILNOR’ /* Also checks for ‘?!’ / / With no argument, this a simple query. */ Output=#Tracing.#Level

if #Interactive.#Level then Output = ‘?’||Output if #Bif_ArgExists.1 then return Output

Value=#Bif Arg.1

#Interactive.#Level=0

/* A question mark sets the interactive flag. */ if left(Value,1)==‘?’ then do

#Interactive.#Level = 1

Value=substr (Value, 2)

end /* Absence of a letter leaves the setting unchanged. */ if Value==’ ’ then do

Value=translate (left (Value,1)) if Value==‘0O’ then #Interactive.#Level=‘0’ #Tracing.#Level = Value end return Output

10.4 Conversion built-in functions

Conversions between Binary form, Decimal form, and heXadecimal form do not depend on the encoding (see nnn) of the character data.

Conversion to Coded form gives a result which depends on the encoding. Depending on the encoding, the result may be a string that does not represent any sequence of characters.

12.5.9 B2xX

B2X performs binary to hexadecimal conversion.

call CheckArgs ‘rBIN’

String = space (#Bif Arg.1,0) return ReRadix(String,2,16)

12.5.10 BITAND

The functions BITAND, BITOR and BITXOR operate on encoded character data. Each binary digit from the encoding of the first argument is processed in conjunction with the corresponding bit from the second argument.

call CheckArgs ‘rANY oANY oPAD’

Stringl = #Bif_ Arg.1 if #Bif ArgExists.2 then String2

#Bif Arg.2 else String2 rr

/* Presence of a pad implies character strings. */ if #Bif_ArgExists.3 then if length(Stringl) > length(String2) then String2=left (String2,length(String1l),#Bif_-Arg.3) else Stringl=left(Stringl,length(String2),#Bif_Arg.3)

/* Change to manifest bit representation. / *#Response=Config C2B(String1) String1l=#Outcome #Response=Config C2B(String2) String2=#0utcome* / Exchange if necessary to make shorter second. */ if length(Stringl)<length(String2) then do

t=Stringl

Stringl=String2

String2=t

end

/* Operate on common length of those bit strings. */ r=’ ’ do j=1 to length (String2)

bl=substr (Stringl,j,1)

b2=substr (String2,j,1)

select when #Bif=‘BITAND’ then b1=b1&b2 when #Bif=‘BITOR’ then bl=b1|b2

115 when #Bif=‘BITXOR’ then b1=b1&&b2 end r=r||bl end j rer || right (Stringl, length (String1) -length(String2))

/* Convert back to encoded characters. */ return x2c (b2x(r))

12.5.11 BITOR

See nnn

12.5.12 BITXOR

See nnn

12.5.13 C2D

C2D performs coded to decimal conversion.

call CheckArgs ‘rANY oWHOLE>=0’ if length(#Bif Arg.1)=0 then return 0

if #Bif_ArgExists.2 then do /* Size specified / *Size = #Bif_Arg.2 if Size = 0 then return 0* / Pad will normally be zeros / *t=right (#Bif Arg.1,Size,left (xrange(),1))* / Convert to manifest bit / *call Config C2B t* / And then to signed decimal. / *Sign = Left (#Outcome,1) #Outcome = substr(#Outcome, 2) t=ReRadix (#Outcome, 2,10)* / Sign indicates 2s-complement. / *if Sign then t=t-2**length(#Outcome) if abs(t) > 10** #Digits.#Level - 1 then call Raise 40.35, t return t end* / Size not specified. */ call Config C2B #Bif_Arg.1 t = ReRadix(#Outcome, 2,10) if t > 10 ** #Digits.#Level - 1 then call Raise 40.35, t return t

10.4.6 C2Xx C2X performs coded to hexadecimal conversion.

call CheckArgs ‘rANY’ if length(#Bif Arg.1) = 0 then return ’ ’

call Config C2B #Bif_Arg.1 return ReRadix (#Outcome,2,16)

12.5.14 D2C

D2C performs decimal to coded conversion.

‘rWHOLENUM>=0’! ‘rWHOLENUM rWHOLE>=0’

if #Bif_ArgExists.2 then ArgData else ArgData call CheckArgs ArgData

/* Convert to manifest binary / *Subject = abs(#Bif Arg.1) r = ReRadix(Subject,10,2)* / Make length a multiple of 8, as required for Config B2C */ Length = length(r) do while Length//8 = 0 Length = Length+1 end r= right(r,Length,‘0’)

/* 2s-complement for negatives. */ if #Bif_Arg.1<0 then do Subject = 2**length(r)-Subject r = ReRadix(Subject,10,2) end /* Convert to characters */ #Response = Config B2C(r) Output = #Outcome if #Bif_ArgExists.2 then return Output

/* Adjust the length with appropriate characters. */ if #Bif_Arg.1>=0 then return right (Output, #Bif_Arg.2,left(xrange(),1)) else return right (Output, #Bif Arg.2,right (xrange(),1))

12.5.15 D2X

D2X performs decimal to hexadecimal conversion. if #Bif_ArgExists.2 then ArgData = ‘rWHOLENUM>=0’ else ArgData = ‘rWHOLENUM rWHOLE>=0’

call CheckArgs ArgData

/* Convert to manifest hexadecimal / *Subject = abs(#Bif Arg.1) r = ReRadix(Subject,10,16)* / Twos-complement for negatives */ if #Bif_Arg.1<0 then do Subject = 16**length(r) -Subject r = ReRadix(Subject,10,16) end if #Bif_ArgExists.2 then return r /* Adjust the length with appropriate characters. */ if #Bif_ Arg.1>=0 then return right(r,#Bif Arg.2,‘0’) else return right(r,#Bif Arg.2,‘F’)

12.5.16 X2B

X2B performs hexadecimal to binary conversion.

call CheckArgs ‘rHEX’

Subject = #Bif Arg.1

if Subject == ' ' then return ' '

/* Blanks were checked by CheckArgs, here they are ignored. */ Subject = space (Subject, 0)

return ReRadix(translate (Subject) ,16,2)

12.5.17 X2C

X2C performs hexadecimal to coded character conversion.

call CheckArgs 'rHEX'

Subject = #Bif Arg.1

if Subject == ' ' then return ' '

Subject = space (Subject, 0)

/* Convert to manifest binary */

r = ReRadix(translate (Subject) ,16,2)

/* Convert to character */

Length = 8*((length (Subject) +1) %2) #Response = Config B2C(right(r,Length,'0')) return #Outcome

12.5.18 X2D

X2D performs hexadecimal to decimal conversion.

call CheckArgs 'rHEX OWHOLE>=0'

Subject = #Bif Arg.1 if Subject == ' ' then return '0'

Subject = translate (space (Subject,0))

if #Bif ArgExists.2 then

Subject = right (Subject, #Bif Arg.2,'0') if Subject ==' ' then return '0' /* Note the sign */ if #Bif ArgExists.2 then SignBit

else SignBit

/* Convert to decimal / *r = ReRadix(Subject,16,10)* / Twos-complement */ if SignBit then r = 2**(4*#Bif Arg.2) - r if abs(r)>10 ** #Digits.#Level - 1 then call Raise 40.35, t return r

left (x2b (Subject) ,1) ror

12.6 Input/Output built-in functions

The configuration shall provide the ability to access streams. Streams are identified by character string identifiers and provide for the reading and writing of data. They shall support the concepts of characters, lines, and positioning.

The input/output built-in functions interact with one another, and they make use of Config_ functions, see nnn. When the operations are successful the following characteristics shall be exhibited: - The CHARIN/CHAROUT functions are insensitive to the lengths of the arguments. The data written to a stream by CHAROUT can be read by a different number of CHARINs. - The CHARIN/CHAROUT functions are reflective, that is, the concatenation of the data read from a persistent stream by CHARIN (after positioning to 1, while CHARS(Stream)>0), will be the same as the concatenation of the data put by CHAROUT. - All characters can be used as CHARIN/CHAROUT data. - The CHARS(Stream, 'N') function will return zero only when a subsequent read (without positioning) is guaranteed to raise the NOTREADY condition. - The LINEIN/LINEOUT functions are sensitive to the length of the arguments, that is, the length of a line written by LINEOUT is the same as the length of the string returned by successful LINEIN of the line. - Some characters, call them line-banned characters, cannot reliably be used as data for LINEIN/LINEOUT. If these are not used, LINEIN/LINEOUT is reflective. If they are used, the result is not defined. The set of characters which are line-barred is a property of the configuration. - The LINES(Stream, 'N') function will return zero only when a subsequent LINEIN (without positioning) is guaranteed to raise the NOTREADY condition. - When a persistent stream is repositioned and written to with CHAROUT, the previously written data is not lost, except for the data overwritten by this latest CHAROUT. - When a persistent stream is repositioned and written to with LINEOUT, the previously written data is not lost, except for the data overwritten by this latest LINEOUT, which may leave lines partially overwritten. ### CHARIN CHARIN returns a string read from the stream named by the first argument.

call CheckArgs 'oSTREAM oOWHOLE>0 oOWHOLE>=0'

if #Bif_ArgExists.1 then Stream else Stream #StreamState.Stream = ' ' /* Argument 2 is positioning. */ if #Bif_ArgExists.2 then do #Response = Config Stream Position(Stream, 'CHARIN', #Bif Arg.2)

#Bif Arg.1

if left(#Response, 1) == 'R' then call Raise 40.41, 2, #Bif_ Arg.2 if left(#Response, 1) == 'T' then call Raise 40.42,Stream end

/* Argument 3 is how many. */ *if #Bif_ArgExists.3 then Count else Count if Count = 0 then do call Config Stream Charin Stream, 'NULL' /* "Touch" the stream /* return '! end /* The unit may be eight bits (as characters) or one character. */ call Config Stream Query Stream

#Bif Arg.3 1

Mode = #Outcome

do until Count = 0

#Response = Config Stream Charin(Stream, 'CHARIN')

if left(#Response, 1) == 'N' then do if left (#Response, 1) == 'E' then #StreamState.Stream = 'ERROR' /* This call will return. */ call #Raise 'NOTREADY', Stream, substr(#Response, 2) leave end

r = r| |#Outcome

Count = Count-1

end if Mode == ‘B’ then do call Config B2C r r = #Outcome end return r

12.6.1 CHAROUT

CHAROUT returns the count of characters remaining after attempting to write the second argument to the stream named by the first argument.

call CheckArgs ‘oSTREAM oANY oWHOLE>0’

if #Bif_ArgExists.1 then Stream else Stream

#Bif Arg.1

#StreamState.Stream = ’ ’ if #Bif ArgExists.2 & #Bif_ArgExists.3 then do /* Position to end of stream. */ #Response = Config Stream Close (Stream) if left (#Response,1) == ‘T’ then call Raise 40.42,Stream return 0 end

if #Bif_ArgExists.3 then do /* Explicit positioning. */ #Response = Config Stream Position(Stream, ‘CHAROUT’, #Bif Arg.3)

if left (#Response,1) == ‘T’ then call Raise 40.42,Stream if left(#Response, 1) == ‘R’ then call Raise 40.41, 3, #Bif_ Arg.3 end

if #Bif_ArgExists.2 | #Bif_Arg.2 == ’ ’ then do call Config Stream _Charout Stream, ‘NULL’ /* “Touch” the stream */ return 0 end

String = #Bif Arg.2 call Config Stream Query Stream Mode = #Outcome if Mode == ‘B’ then do call Config C2B String String = #Outcome Stride = 8 Residue = length(String)/8 end else do Stride = 1 Residue = length (String) end

Cursor = 1 do while Residue>0 Piece = substr (String, Cursor, Stride) Cursor = Cursor+Stride call Config Stream Charout Stream, Piece if left (#Response, 1) == ‘N’ then do if left (#Response, 1) == ‘E’ then #StreamState.Stream = ‘ERROR’ call #Raise ‘NOTREADY’, Stream, substr(#Response, 2)

return Residue end Residue = Residue - 1 end return 0

12.6.2 CHARS

CHARS indicates whether there are characters remaining in the named stream. Optionally, it returns a

count of the characters remaining and immediately available.

call CheckArgs ‘oSTREAM oCN’

if #Bif_ArgExists.1 then Stream = #Bif_Arg.1 else Stream = ’ ’

if #Bif_ArgExists.2 then Option = #Bif Arg.2 else Option = ‘N’

call Config Stream Count Stream, ‘CHARS’, Option return #Outcome

12.6.3 LINEIN

LINEIN reads a line from the stream named by the first argument, unless the third argument is Zero.

call CheckArgs ‘oSTREAM oOWHOLE>0 oOWHOLE>=0’

if #Bif_ArgExists.1 then Stream else Stream #StreamState.Stream = ’ ’ if #Bif_-ArgExists.2 then do #Response = Config Stream Position(Stream, ‘LINEIN’, #Bif Arg2) if left (#Response, 1) ‘T’ then call Raise 40.42,Stream if left (#Response, 1) ‘R’ then call Raise 40.41, 2, #Bif_ Arg.2 end if #Bif_ArgExists.3 then Count #Bif Arg.3 else Count 1 if Count>1 then call Raise 40.39, Count if Count = 0 then do call Config Stream Charin Stream, ‘NULL’ /* “Touch” the stream */ return ‘! end /* A configuration may recognise lines even in ’binary’ mode. */ call Config Stream Query Stream Mode = #Outcome re ter t = #Linein Position.Stream /* Config Stream Charin will alter #Linein Position. */ do until t = #Linein Position.Stream #Response = Config Stream Charin(Stream, ‘LINEIN’) if left(#Response, 1) == ‘N’ then do if left (#Response, 1) == ‘E’ then #StreamState.Stream = ‘ERROR’ call #Raise ‘NOTREADY’, Stream, substr(#Response, 2) leave end r = r||#Outcome end if Mode == ‘B’ then do call Config B2C r r= #Outcome end return r

#Bif Arg.1

12.6.4 LINEOUT

LINEOUT returns ‘1’ or ‘0’, indicating whether the second argument has been successfully written to the

stream named by the first argument. A result of ‘1’ means an unsuccessful write.
call CheckArgs ‘oSTREAM oANY oWHOLE>0’

if #Bif_ArgExists.1 then Stream else Stream

#Bif Arg.1

#StreamState.Stream = ’ ’

if #Bif ArgExists.2 & #Bif_ArgExists.3 then do

/* Position to end of stream.

*/

#Response = Config Stream Close (Stream) if left (#Response,1) == ‘T’ then call Raise 40.42,Stream

return 0 end

if #Bif_ArgExists.3 then do

#Response = Config Stream Position(Stream, ‘LINEOUT’, #Bif Arg.3)

if left(#Response, 1) == ‘T’ then call Raise 40.42,Stream if left (#Response, 1) == ‘R’ then call Raise 40.41, 3, #Bif_Arg.3 end

if #Bif ArgExists.2 then do

call Config Stream _Charout Stream, ’ ’ /* “Touch” the stream */

return 0 end

String #Bif Arg.2 Stride 1 call Config Stream Query Stream Mode = #Outcome if Mode == ‘B’ then do call Config C2B String String = #Outcome Stride = 8 Residue = length(String)/8 end else do Stride = 1 Residue = length(String) end Cursor = 1 do while Residue > 0

Piece = substr (String, Cursor, Stride)

Cursor = Cursor+Stride

call Config Stream Charout Stream, Piece then do

then #StreamState.Stream = ‘ERROR’

call #Raise ‘NOTREADY’, Stream, substr(#Response, 2)

if left(#Response, 1) == ‘N’ if left (#Response, 1) == ‘E’ return 1 end Residue = Residue-1 end call Config Stream Charout Stream, return 0 ### LINES

‘EOL’

LINES returns the number of lines remaining in the named stream.

call CheckArgs ‘oSTREAM oCN’

if #Bif_ArgExists.1 then Stream else Stream if #Bif_ ArgExists.2 then Option else Option

Call Config Stream Count Stream, return #Outcome

12.6.5 QUALIFY

#Bif Arg.1 ter #Bif_Arg.2 tint

LINES’, Option

QUALIFY returns a name for the stream named by the argument. The two names are currently associated with the same resource and the result of QUALIFY may be more persistently associated with

that resource. call CheckArgs ‘oSTREAM’

if #Bif_ArgExists.1 then Stream else Stream

#Bif Arg.1 #Response = Config Stream Qualified (Stream) return #Outcome

12.6.6 STREAM

STREAM returns a description of the state of, or the result of an operation upon, the stream named by the first argument.

/* Third argument is only correct with ‘C’ */

if #Bif ArgExists.2 & translate(left(#Bif Arg.2, 1)) == ‘C’ then ArgData = ‘rSTREAM rCDS rANY’

else ArgData = ‘rSTREAM oCDS’

call CheckArgs ArgData

Stream = #Bif_Arg.1

if #Bif_ ArgExists.2 then Operation = #Bif_Arg.2 else Operation = ‘S’ Select when Operation == ‘C’ then do

call Config Stream Command Stream, #Bif Arg.3 return #Outcome

end when Operation == ‘D’ then do #Response = Config Stream State (Stream) return substr(#Response, 2) end when Operation == ‘S’ then do if StreamState.Stream == ‘ERROR’ then return ‘ERROR’ #Response = Config Stream State (Stream)

‘N’ then return ‘READY’ ‘U’ then return ‘UNKNOWN’

if left (#Response, if left (#Response, return ‘NOTREADY’ end

end

1) ==
2) ==

12.7 Other built-in functions

12.7.1 DATE

DATE with fewer than two arguments returns the local date. Otherwise it converts the second argument (which has a format given by the third argument) to the format specified by the first argument. If there are fourth or fifth arguments, they describe the treatment of separators between fields of the date.

call CheckArgs ‘oBDEMNOSUW oANY oOBDENOSU oSEP oSEP’ /* If the third argument is given then the second is mandatory. */ if #Bif_ArgExists.3 & #Bif_-ArgExists.2 then

call Raise 40.19, ’ ’, #Bif Arg.3

if #Bif_ ArgExists.1 then Option else Option

#Bif Arg.1 tint

/* The date/time is ‘frozen’ throughout a clause. */ *if #ClauseTime.#Level == ’ ’ then do #Response = Config Time () #ClauseTime.#Level = #Time #ClauseLocal.#Level = #Time + #Adjust<Index “#Adjust” # “ ” > end* / English spellings are used, even if messages not in English are used. */ Months = ‘January February March April May June July’, ‘August September October November December’ WeekDays = ‘Monday Tuesday Wednesday Thursday Friday Saturday Sunday’

/* If there is no second argument, the current date is returned. */ if #Bif ArgExists.2 then return DateFormat (#ClauseLocal.#Level, Option)

/* If there is a second argument it provides the date to be converted. */

Value = #Bif_Arg.2 if #Bif_ ArgExists.3 then InOption else InOption if Option == ‘S’ then OutSeparator else OutSeparator if #Bif_ArgExists.4 then do

if OutSeparator == ‘x’ then call OutSeparator = #Bif.Arg.4 end

if InOption == ‘S’ then InSeparator

```
else InSeparator if #Bif_ArgExists.5 then do
#Bif_Arg.3 ’nt
translate (Option, “xx/x //x”,“BDEMNOUW”)
Raise 40.46, Option, 4
translate(InOption,“xx/ //”,“BDENOU”)
if InSeparator == ‘x’ then call Raise 40.46, InOption, 5
InSeparator = #Bif.Arg.5 end /* First try for Year Month Day */ Logic = ‘NS’ select
when InOption == ‘N’ then do if InSeparator == ’ ’ then do
if length(Value)<9 then return Year = right (Value, 4)
MonthIs = substr (right (Value,7),1,3) Day = left (Value, length (Value) -7)
end else
parse var Value Day (InSeparator) MonthIs (InSeparator) Year
do Month = 1 to 12 if left(word(Months, Month), 3)
== MonthIs then leave
parse var Value Year (InSeparator) Month (InSeparator) Day
end Month
end
when InOption == ‘S’ then if InSeparator == ’ ’ then
parse var Value Year +4 Month +2 Day
else
otherwise Logic = ‘EOU’ /* or BD */
end
/* Next try for year without century */
parse var Value Day (InSeparator) Month (InSeparator) YY
parse var Value YY (InSeparator) Month (InSeparator) Day
parse var Value Month (InSeparator) Day (InSeparator) YY
if logic = ‘EOU’ then Select when InOption == ‘E’ then if InSeparator == ’ ’
then parse var Value Day +2 Month +2 YY else when InOption == ‘O’ then if
InSeparator == ’ ’ then parse var Value YY +2 Month +2 Day else when InOption
== ‘U’ then if InSeparator == ’ ’ then parse var Value Month +2 Day +2 YY else
otherwise Logic = ‘BD’ end if Logic = ‘EOU’ then do
/* The century is assumed, on the basis of the current year. */
if datatype(YY,‘W’)=0 then return
YearNow = left(‘DATE’(‘S’),4)
Year = YY
do while Year < YearNow-50 Year = Year + 100
end
end /* Century assumption */
```

if Logic <> ‘BD’ then do /* Convert Month & Day to Days of year. */ *if datatype(Month,‘W’)=0 | datatype(Day,‘W’)=0 | datatype(Year,‘W’)=0 then return Days = word(‘0 31 59 90 120 151 181 212 243 273 304 334’,Month), + (Month>2)*Leap(Year) + Day-1

end else

if datatype(Value,‘W’)=0 then

return

if InOption == ‘D’ then do

Year = left(‘DATE’ (‘S’),4)

Days = Value - 1 /* ‘D’ includes current day */ end

/* Convert to BaseDays */ if InOption <> ‘B’ then

BaseDays = (Year-1)*365 + (Year-1)%4 - (Year-1)%100 + (Year-1)%400 + Days else

Basedays = Value

/* Convert to microseconds from 0001 */ *Micro = BaseDays* 86400 * 1000000

/* Reconvert to check the original. (eg for Month = 99) */

if DateFormat (Micro,InOption,InSeparator) == Value then call Raise 40.19, Value, InOption

return DateFormat (Micro, Option, OutSeparator)

DateFormat: /* Convert from microseconds to given format. */ parse value Time2Date(arg(1)) with, Year Month Day Hour Minute Second Microsecond Base Days

select

when arg(2) == ‘B’ then

return Base when arg(2) == ‘D’ then

return Days when arg(2) == ‘E’ then

return right (Day,2,‘0’) (arg(3)) right (Month,2,‘0’) (arg(3)) right (Year,2,‘0’) when arg(2) == ‘M’ then

return word (Months ,Month) when arg(2) == ‘N’ then

return (Day) (arg(3)) left (word(Months,Month),3) (arg(3))right (Year,4,‘0’) when arg(2) == ‘O’ then

return right (Year,2,‘0’) (arg(3))right (Month,2,‘0’) (arg(3))right (Day,2,‘0’) when arg(2) == ‘S’ then

return right (Year,4,‘0’) (arg(3))right (Month,2,‘0’) (arg(3))right (Day,2,‘0’) when arg(2) == ‘U’ then

return right (Month,2,‘0’) (arg(3)) right (Day,2,‘0’) (arg(3)) right (Year,2,‘0’) otherwise /* arg(2) == ‘W’ */

return word (Weekdays, 1+Base//7)

end

12.7.2 QUEUED

QUEUED returns the number of lines remaining in the external data queue.

call CheckArgs ' '

#Response = Config Queued() return #Outcome

12.7.3 RANDOM

RANDOM returns a quasi-random number.

call CheckArgs ‘oWHOLE>=0 oWHOLE>=0 oWHOLE>=0’

if #Bif_Arg.0 = 1 then do Minimum = 0

Maximum = #Bif Arg.1 if Maximum>100000 then call Raise 40.31, Maximum

end else do if #Bif_ArgExists.1 then Minimum = #Bif Arg.1 else Minimum = 0 if #Bif_ArgExists.2 then Maximum = #Bif_Arg.2 else Maximum = 999

end

if Maximum-Minimum>100000 then call Raise 40.32, Minimum, Maximum

if Maximum-Minimum<0 then call Raise 40.33, Minimum, Maximum

if #Bif_ArgExists.3 then call Config Random Seed #Bif_Arg.3 call Config Random Next Minimum, Maximum return #Outcome

12.7.4 SYMBOL

The function SYMBOL takes one argument, which is evaluated. Let String be the value of that argument. If Config_Length(String) returns an indicator ‘E’ then the SYNTAX condition 23.1 shall be raised. Otherwise, if the syntactic recognition described in section nnn would not recognize String as a symbol then the result of the function SYMBOL is ’BAD”.

If String would be recognized as a symbol the result of the function SYMBOL depends on the outcome of accessing the value of that symbol, see nnn. If the final use of Var_Value leaves the indicator with value ‘D’ then the result of the function SYMBOL is ‘LIT’, otherwise ‘VAR’.

12.7.5 TIME

TIME with less than two arguments returns the local time within the day, or an elapsed time. Otherwise it converts the second argument (which has a format given by the third argument) to the format specified by the first argument.

call CheckArgs ‘oCEHLMNORS oANY oCHLMNS’ /* If the third argument is given then the second is mandatory. */ if #Bif_ArgExists.3 & #Bif_ArgExists.2 then

call Raise 40.19, ' ', #Bif Arg.3

if #Bif_ ArgExists.1 then Option else Option

#Bif Arg.1 tint

/* The date/time is ‘frozen’ throughout a clause. */

if #ClauseTime.#Level == ’ ’ then do #Response = Config Time () #ClauseTime.#Level = #Time #ClauseLocal.#Level = #Time + #Adjust<Index “#Adjust” # “ ” > end

/* If there is no second argument, the current time is returned. */ if #Bif ArgExists.2 then return TimeFormat (#ClauseLocal.#Level, Option)

/* If there is a second argument it provides the time to be converted. */ if pos(Option, ‘ERO’) > 0 then call Raise 40.29, Option InValue = #Bif Arg.2

if #Bif_ArgExists.3 then InOption = #Bif_Arg.3 else InOption =

HH = 0

MM = 0

ss = 0

HourAdjust = 0

select

when InOption == ‘C’ then do parse var InValue HH ‘:’ . +1 MM +2 XX

if HH = 12 then

125 HH = 0

if XX == ‘pm’ then HourAdjust = 12 end when InOption == ‘H’ then HH = InValue when InOption == ‘L’ | InOption == ‘N’ then parse var InValue HH ‘:’ MM ‘:’ SS when InOption == ‘M’ then MM = InValue otherwise SS = InValue

end

if datatype(HH,‘W’)=0 | datatype(MM,‘W’)=0 | datatype(SS,‘N’)=0 then call Raise 40.19, InValue, InOption

HH = HH + HourAdjust

/* Convert to microseconds */

Micro = trunc((((HH * 60) + MM) * 60 + SS) * 1000000)

/* There is no special message for time-out-of-range; the bad-format

message is used. */

if Micro<0 | Micro > 24*3600*1000000 then call Raise 40.19, InValue, InOption

/* Reconvert to check the original. (eg for hour = 99) */

if TimeFormat (Micro,InOption) == InValue then call Raise 40.19, InValue, InOption

return TimeFormat (Micro, Option)

end /* Conversion */

TimeFormat: procedure /* Convert from microseconds to given format. / / The day will be irrelevant; actually it will be the first day possible. */ *x = Time2Date2(arg(1)) parse value x with Year Month Day Hour Minute Second Microsecond Base Days select when arg(2) == ‘C’ then select when Hour>12 then return Hour-*

12’:‘right (Minute,2,’0’)‘pm’ when Hour=12 then return ‘12:’right(Minute,2,’0’)‘pm’ when Hour>0 then return Hour’:‘right (Minute,2,’0’)‘am’ when Hour=0 then return ‘12:’right(Minute,2,’0’)‘am’ end when arg(2) == ‘H’ then return Hour when arg(2) == ‘L’ then return right (Hour,2,‘0’)‘:’right (Minute,2,’0’)‘:’right(Second,2,’0’), || *‘.’right (Microsecond,6,’0’) when arg(2) == ‘M’ then return 60*Hour+Minute when arg(2) == ‘N’ then return right (Hour,2,‘0’)‘:’right (Minute,2,’0’)‘:’right(Second,2,’0’) otherwise /* arg(2) == ‘S’ */ *return 3600*Hour+60*Minute+Second end

Time2Date: /* These are checks on the range of the date. */ if arg(1) < 0 then call Raise 40.19, InValue, InOption if arg(1) >= 315537897600000000 then call Raise 40.19, InValue, InOption return Time2Date2 (arg(1))

10.6.1 VALUE VALUE returns the value of the symbol named by the first argument, and optionally assigns it a new

value.

‘rANY oOANY oANY’

if #Bif ArgExists.3 then ArgData ‘rSYM oANY oOANY’

else ArgData call CheckArgs ArgData

Subject = #Bif Arg.1

if #Bif_ArgExists.3 then do /* An external pool, or the reserved pool. / / The reserved pool uses a null string as its pool identifier. */ Pool = #Bif_Arg.3

if Pool == ’ ’ then do Subject = ‘.’ || translate (Subject) /* The dot on the name is implied. */ *Value = .environment [Subject]* / Was the translate redundant? */

if #Bif ArgExists.2 then .environment [Subject] = #Bif_ Arg.2 return Value end

/* Fetch the original value */

#Response = Config Get (Pool, Subject)

#Indicator = left (#Response,1)

if #Indicator == ‘F’ then call Raise 40.36, Subject if #Indicator == ‘P’ then

call Raise 40.37, Pool Value = #Outcome if #Bif_ArgExists.2 then do /* Set the new value. */ *#Response = Config Set(Pool,Subject,#Bif Arg.2) if #Indicator == ‘P’ then call Raise 40.37, Pool if #Indicator == ‘F’ then call Raise 40.36, Subject end* / Return the original value. / *return Value end* / Not external / *Subject = translate(Subject)* / See nnn / *Pp = pos(Subject, ‘.’) if p = 0 | p = length(Subject) then do* / Not compound / *#Response = Var Value(#Pool, Subject, ‘0’)* / The caller, in the code of the standard, may need to test whether the Subject was dropped. */ *#Indicator = left(#Response, 1) Value = #Outcome if #Bif_ ArgExists.2 then #Response = Var Set(#Pool, Subject, ‘0’, #Bif_ Arg.2) return Value end* / Compound / *Expanded = left(Subject,p-1)* / The stem / *do forever Start = p+l1 Pp = pos(Subject,‘.’,Start) if p = 0 then p = length(Subject) Item = substr(Subject,Start,p-Start)* / Tail component symbol */ if Item==’ ’ then if pos(left(Item,1),‘0123456789’) = 0 then do #Response = Var Value(#Pool, Item, ‘0’) Item = #Outcome end* / Add tail component. */ Expanded = Expanded’.’Item end #Response = Var Value(#Pool, Expanded, ‘1’) #Indicator = left(#Response, 1) Value = #Outcome if #Bif_ ArgExists.2 then #Response = Var Set(#Pool, Expanded, ‘1’, #Bif_ Arg.2)

return Value

12.7.6 QUEUED

QUEUED returns the number of lines remaining in the external data queue.

call CheckArgs ' ' #Response = Config Queued() return #Outcome

12.7.7 RANDOM

RANDOM returns a quasi-random number.

call CheckArgs ‘oWHOLE>=0 oWHOLE>=0 oWHOLE>=0’

if #Bif_Arg.0 = 1 then do Minimum = 0 Maximum = #Bif_Arg.1 if Maximum>100000 then call Raise 40.31, Maximum

end else do if #Bif_ArgExists.1 then Minimum = #Bif Arg.1 else Minimum = 0 if #Bif_ArgExists.2 then Maximum = #Bif_Arg.2 else Maximum = 999

end

if Maximum-Minimum>100000 then call Raise 40.32, Minimum, Maximum

if Maximum-Minimum<0 then call Raise 40.33, Minimum, Maximum

if #Bif_ArgExists.3 then call Config Random Seed #Bif_Arg.3 call Config Random Next Minimum, Maximum return #Outcome

12.7.8 SYMBOL

The function SYMBOL takes one argument, which is evaluated. Let String be the value of that argument. If Config_Length(String) returns an indicator ‘E’ then the SYNTAX condition 23.1 shall be raised. Otherwise, if the syntactic recognition described in section nnn would not recognize String as a symbol then the result of the function SYMBOL is ’BAD”.

If String would be recognized as a symbol the result of the function SYMBOL depends on the outcome of accessing the value of that symbol, see nnn. If the final use of Var_Value leaves the indicator with value ‘D’ then the result of the function SYMBOL is ‘LIT’, otherwise ‘VAR’.

12.7.9 TIME

TIME with less than two arguments returns the local time within the day, or an elapsed time. Otherwise it converts the second argument (which has a format given by the third argument) to the format specified by the first argument.

call CheckArgs ‘oCEHLMNORS oANY oCHLMNS’ /* If the third argument is given then the second is mandatory. */ if #Bif_ArgExists.3 & #Bif_ArgExists.2 then

call Raise 40.19, ' ', #Bif Arg.3

if #Bif_ ArgExists.1 then Option else Option

#Bif Arg.1 tint

/* The date/time is 'frozen' throughout a clause. */

if #ClauseTime.#Level == ' ' then do #Response = Config Time () #ClauseTime.#Level = #Time #ClauseLocal.#Level = #Time + #Adjust<Index "#Adjust" # " " > end

/* If there is no second argument, the current time is returned. */ if #Bif ArgExists.2 then return TimeFormat (#ClauseLocal.#Level, Option)

/* If there is a second argument it provides the time to be

converted. */ if pos(Option, 'ERO') > 0 then

128 call Raise 40.29, Option InValue = #Bif Arg.2

if #Bif_ArgExists.3 then InOption = #Bif_Arg.3 else InOption = 'N' HH = 0 MM = 0 ss = 0 HourAdjust = 0 select when InOption == 'C' then do parse var InValue HH ':' . +1 MM +2 XX if XX == 'pm' then HourAdjust = 12 end when InOption == 'H' then HH = InValue when InOption == 'L' | InOption == 'N' then parse var InValue HH ':' MM ':' SS when InOption == 'M' then MM = InValue otherwise SS = InValue end

if (HH,'W') | (MM,'W') | (SS,'N') then call Raise 40.19, InValue, InOption

HH = HH + HourAdjust

/* Convert to microseconds */

Micro = trunc((((HH * 60) + MM) * 60 + SS) * 1000000)

/* Reconvert to check the original. (eg for hour = 99) */

if TimeFormat (Micro,InOption) == InValue then call Raise 40.19, InValue, InOption

return TimeFormat (Micro, Option)

end /* Conversion */

TimeFormat: /* Convert from microseconds to given format. */ parse value Time2Date(arg(1)) with, Year Month Day Hour Minute Second Microsecond Base Days select when arg(2) == 'C' then if Hour>12 then return Hour-12':'right (Minute,2,'0')'pm'

else return Hour':'right (Minute,2,'0')'am' when arg(2) == 'E' | arg(2) == 'R' then do

/* Special case first time */

if #StartTime.#Level == ' ' then do #StartTime.#Level #ClauseTime.#Level return '0O'

end Output = #ClauseTime.#Level-#StartTime. #Level if arg(2) == 'R' then

#StartTime.#Level = #ClauseTime.#Level return Output * 1E-6 end /* E or R */ when arg(2) == 'H' then return Hour when arg(2) == 'L' then return right (Hour,2,'0')':'right (Minute,2,'0')':'right(Second,2,'0'), || '.'right (Microsecond,6,'0')

when arg(2) == 'M' then return 60*Hour+Minute when arg(2) == 'N' then return*

right (Hour,2,‘0’)‘:’right (Minute,2,’0’)‘:’right(Second,2,’0’) when arg(2) == ‘O’ then return trunc(#ClauseLocal.#Level - #ClauseTime.#Level) otherwise / arg(2) == ‘S’ */ *return 3600*Hour+60*Minute+Second end 10.6.5 VALUE

VALUE returns the value of the symbol named by the first argument, and optionally assigns it a new value.

‘rANY oOANY oANY’ ‘rSYM OANY oANY’

if #Bif ArgExists.3 then ArgData else ArgData call CheckArgs ArgData

129 130

Subject = #Bif Arg.1

if #Bif_ArgExists.3 then do /* An external pool, or the reserved pool. */ /* The reserved pool uses a null string as its pool identifier. */ Pool = #Bif_Arg.3

if Pool == ’ ’ then do Subject = ‘.’ || translate (Subject) /* The dot on the name is implied. */ *Value = .environment [Subject]* /* Was the translate redundant? */

if #Bif ArgExists.2 then .environment [Subject] = #Bif_ Arg.2 return Value end

/* Fetch the original value */

#Response = Config Get (Pool, Subject)

#Indicator = left (#Response,1)

if #Indicator == ‘F’ then call Raise 40.36, Subject if #Indicator == ‘P’ then

call Raise 40.37, Pool Value = #Outcome if #Bif_ArgExists.2 then do /* Set the new value. */ *#Response = Config Set(Pool,Subject,#Bif Arg.2) if #Indicator == ‘P’ then call Raise 40.37, Pool if #Indicator == ‘F’ then call Raise 40.36, Subject end* /* Return the original value. */ *return Value end* /* Not external */ *Subject = translate(Subject)* /* See nnn */ *Pp = pos(Subject, ‘.’) if p = 0 | p = length(Subject) then do* /* Not compound */ *#Response = Var Value(#Pool, Subject, ‘0’)* /* The caller, in the code of the standard, may need to test whether the Subject was dropped. */ *#Indicator = left(#Response, 1) Value = #Outcome if #Bif_ ArgExists.2 then #Response = Var Set(#Pool, Subject, ‘0’, #Bif_ Arg.2) return Value end* /* Compound */ *Expanded = left(Subject,p-1)* /* The stem */ *do forever Start = p+l1 Pp = pos(Subject,‘.’,Start) if p = 0 then p = length(Subject) Item = substr(Subject,Start,p-Start)* /* Tail component symbol */ *if Item==’ ’ then if pos(left(Item,1),‘0123456789’) = 0 then do #Response = Var Value(#Pool, Item, ‘0’) Item = #Outcome end* /* Add tail component. */ Expanded = Expanded’.’Item end #Response = Var Value(#Pool, Expanded, ‘1’) #Indicator = left(#Response, 1) Value = #Outcome if #Bif_ ArgExists.2 then #Response = Var Set(#Pool, Expanded, ‘1’, #Bif_ Arg.2)

return Value

13

Built-in classes

13.1 Notation

The built-in classes are defined mainly through code. The code refers to state variables. This is solely a notation used in this standard.

13.2 Object, class and method

These objects provide the basis for class structure.

13.2.1 The object class

::class object

::method new class Returns a new instance of the receiver class.

call Config ObjectNew return #Outcome

::method ‘==’ ‘==’ with no argument gives a hash value in OOI.

call Config ObjectCompare #Receiver, #Arg.1

if #Outcome == ‘equal’ then return ‘1’

else return ‘0’

::method ‘<>!’

use arg a

return ==a

::method ‘><’ forward message ‘<>’

::method ‘=’ forward message ‘==’

::method ‘=! forward message’<>’

::method ‘==’ forward message ‘<>’

::method copy Returns a copy of the receiver object. The copied object has the same methods as the receiver object and an equivalent set of object variables, with the same values.

call Config ObjectCopy #Receiver

return #Outcome

Since we have var_empty we could save a primitive by rendering ‘new’ as ‘copy’ plus ‘empty’. ::method defaultname

Returns a short human-readable string representation for the object.

call var_value #Receiver, ‘#Human’, ‘0’

return #Outcome

This field would have been filled in at ‘NEW’ time.

::method ‘OBJECTNAMES’ /* rvSTRING */

Sets the receiver object’s name to the specified string.

call var_set #Receiver, #ObjectName, ‘0’, #Arg.1 return

Initialized to #Human? Or ObjectName does forwarding until assigned to?

::method objectname

Returns the receiver object’s name (which the OBJECTNAME= method sets). call var_value #Receiver, #ObjectName, ‘0’ return #Outcome

::method string

Returns a human-readable string representation for the object. return #Receiver~ObjectName

::method class Returns the class object that received the message that created the object.

call var value #Receiver, #IsA, ‘0’ return #Outcome

::method setmethod /* rSTRING oSTRING/METHOD/ARRAY */ Adds a method to the receiver object’s collection of object methods.

Is ‘object methods’ what is intended; you add to a class without adding to its instance methods? Yes. if #ArgExists.2 then m = Arg.2

else m = .NIL call set_var #Receiver, ‘METHODS.’#Arg.1, ‘1’, m return ::method hasmethod /* rSTRING */

Returns 1 (true) if the receiver object has a method with the specified name (translated to uppercase); otherwise, returns 0 (false).

This presumably means inherited as well as SETMETHOD ones. What about ones set to .NIL?

Need to use the same search as for sending.

::method unsetmethod private

Removes a method from the receiver object’s collection of object methods. Use var_drop

Private means Receiver = Self check.

::method request /* rSTRING */ Returns an object of the specified class, or the NIL object if the request cannot be satisfied. t = ‘MAKE’#Arg.1

if #Receiver~hasmethod(t) then return .NIL forward message(t) array ()

::method run private /* rMETHOD Ugh keyoptions */ Runs the specified method. The method has access to the object variables of the receiver object, just as if the receiver object had defined the method by using SETMETHOD.

::smethod startat Undocumented?

::method start /* rMESSAGE oArglist */

Returns a message object and sends it a START message to start concurrent processing. ::method init

Performs any required object initialization.

13.2.2 The class class

::class class

Lots of these methods are both class and instance. | don't know whether to list them twice. ::method new class /* OARGLIST */

Returns a new instance of the receiver class, whose object methods are the instance methods of the class. This method initializes a new instance by running its INIT methods.

::method subclass class

Returns a new subclass of the receiver class.

::method subclasses class

Returns the immediate subclasses of the receiver class in the form of a single-index array of the required size.

::method define class /* rSTRING oMETHOD */

Incorporates the method object in the receiver class's collection of instance methods. The method name is translated to upper case.

::method delete

Removes the receiver class's definition for the method name specified.

Builtin classes cannot be altered.

::method method class /* rSTRING */

Returns the method object for the receiver class's definition for the method name given.

Do we have to keep saying "method object" as opposed to "method" because "method name" exists? ::method querymixinclass

Returns 1 (true) if the class is a mixin class or 0 (false) otherwise.

::method mixinclass class /* 3 of em */

Returns a new mixin subclass of the receiver class.

::method inherit class /* rCLASS oCLASS */

Causes the receiver class to inherit the instance and class methods of the class object specified. The

optional class is a class object that specifies the position of the new superclass in the list of superclasses. ::method uninherit class /* rCLASSOBJ */

Nullifies the effect of any previous INHERIT message sent to the receiver for

the class specified. ::smethod enhanced class /* rCOLLECTION oArgs */

Returns an enhanced new instance of the receiver class, with object methods that are the instance

methods of the class enhanced by the methods in the specified collection of methods. ::method baseclass class

Returns the base class associated with the class. If the class is a mixin class, the base class is the first superclass that is not also a mixin class. If the class is not a mixin class, then the base class is the class receiving the BASECLASS message.

::method superclasses class Returns the immediate superclasses of the receiver class in the form of a single-index array of the

required size. ::method id class

Returns a string that is the class identity (instance SUBCLASS and MIXINCLASS methods.)

::method metaclass class

Returns the receiver class's default metaclass. ::method methods class /* oCLASSOBJECT */

Returns a supplier object for all the instance methods of the receiver class and its superclasses, if no argument is specified. ### The method class

::class method

::method new class /* rSTRING rSOURCE */ Returns a new instance of method class, which is an executable representation of the code contained in

the source. ::method setprivate

Specifies that a method is a private method. ::method setprotected

::method setsecuritymanager

::method setguarded Reverses any previous SETUNGUARDED messages, restoring the receiver to the default guarded

status. ::method setunguarded

Lets an object run a method even when another method is active on the same object. If a method object

does not receive a SETUNGUARDED message, it requires exclusive use of its object variable pool. ::method source

Returns the method source code as a single index array of source lines. ::method interface

::method setinterface

11.3 The string class The string class provides conventional strings and numbers.

Some differences from REXX class of NetRexx. ::class string

::method new class

::method '' We can do all the operators by appeal to classic section 7. ::method'-' ::method '-'

use arg a return eneral problem of making the error message come right.

:method :method

:method :method

All

centre center changestr compare copies counstr datatype delstr delword insert lastpos left length overlay pos reverse right space strip substr subword translate verify word wordindex wordlength wordpos words abs format max

min

sign trunc B2x bitand bitor bitxor C2D

C2x

D2xX ::smethod D2C ::smethod X2B ::smethod X2C ::smethod X2D ::method string

::method makestring ### The array class

The main features of a single dimension array are provided by the configuration. This section defines further methods and multi-dimensional arrays.

To be done. Dimensionality set at first use. Count commas, not classic arg().

::class array

::method new class /* 0 or more WHOLE>=0 */

Returns a new empty array.

::method of class /* 0 or more ANY */

Returns a newly created single-index array containing the specified value objects. ::method put /* rANY one or more WHOLE>0 */

Makes the object value a member item of the array and associates it with the specified index or indexes. ::method ' []=' /* 1 or more WHOLE>0O */

This method is the same as the PUT method.

::method at /* 1 or more WHOLE>0O */

Returns the item associated with the specified index or indexes.

::method '[]' /* 1 or more WHOLE>0O */

Returns the same value as the AT method.

::method remove /* 1 or more WHOLE>0O */

Returns and removes the member item with the specified index or indexes from the array. ::method hasindex /* 1 or more WHOLE>0O */

Returns 1 (true) if the array contains an item associated with the specified index

or indexes. Returns 0 (false) otherwise.

::method items /* (None) / *Returns the number of items in the collection. ::method dimension* / OWHOLE>0O */

Returns the current size (upper bound) of dimension specified (a positive whole number). If you omit the argument this method returns the dimensionality (number of dimensions) of the array.

::method size /* (None) */

Returns the number of items that can be placed in the array before it needs to be extended.

::method first /* (None) */

Returns the index of the first item in the array, or the NIL object if the array is empty.

::method last /* (None) */

Returns the index of the last item in the array, or the NIL object if the array is empty.

::method next /* rcWHOLE>O */

Returns the index of the item that follows the array item having the specified index or returns the NIL object if the item having that index is last in the array.

::method previous /* rcWHOLE>O */

Returns the index of the item that precedes the array item having index index or the NIL object if the item having that index is first in the array.

::method makearray /* (None) */

Returns a single-index array with the same number of items as the receiver object. Any index with no associated item is omitted from the new array.

Returns a new array (of the same class as the receiver) containing selected items from the receiver array. The first item in the new array is the item corresponding to index start (the first argument) in the receiver

array. ::method supplier /* (None) / *Returns a supplier object for the collection. ::method section* / rcWHOLE>0O oOWHOLE>=0 */

13.3 The supplier class

A supplier object enumerates the items a collection contained at the time of the supplier’s creation. ::class supplier

::method new class /* rANYARRAY rINDEXARRAY */ Returns a new supplier object.

::method index

Returns the index of the current item in the collection. ::method next

Moves to the next item in the collection.

::method item

Returns the current item in the collection.

::method available

Returns 1 (true) if an item is available from the supplier (that is, if the ITEM method would return a value). Returns 0 (false) otherwise.

11.5 The message class ::class message

::method init class /* Ugh */

Initializes the message object for sending……

::method completed

Returns 1 if the message object has completed its message; returns 0 otherwise.
::method notify /* rMESSAGE */

Requests notification about the completion of processing for the message SEND or START sends. ::method start /* oANY */

Sends the message for processing concurrently with continued processing of the sender. ::method send /* oANY */

Returns the result (if any) of sending the message.

::method result

Returns the result of the message SEND or START sends.

14

Provided classes

(Informative)

14.1 Notation

The provided classes are defined mainly through code. ## The Collection Classes

14.1.1 Collection Class Routines

These routines are used in the definition of the collection classes ::routine CommonxXor /* Returns a new collection that contains all items from self and the argument except that all indexes that appear in both collections are removed. / / When the target is a bag, there may be an index in the bag that is duplicated and the same value as an index in the argument. Should one copy of the index survive in the bag? */ *vel if (arg(1)~class==.Set & arg(2)~class==.Bag) then v=2 if (arg(1)~class==.Table & arg(2)~class==.Bag) then v=2 if (arg(1)~class==.Table & arg(2)~class==.Relation) then v=2 if (arg(1)~class==.Directory & arg(2)~class==.Bag) then v=2 if (arg(1)~class==.Directory & arg(2)~class==.Relation) then v=2* / This version it does: */ *if v=1 then do This = arg(1)* / self of caller */ r=This$_{class}$new ab=MayEnBag (arg (2)) ss=This~supplier do while ss~available r[ss~index] =ss~item ss~next end cs=ab~supplier do while cs~available if r$_{hasindex(cs}$index) then r~remove (cs~index) else r[cs~index] =cs~item cs~next end return r end

/* But following matches practice on Set~XOR(bag) etc. */

This = arg(1) /* self of caller */

r=This$_{class}$new

ab=MayEnBag (arg (2))

ss=This~supplier

do while ss~available if $_{hasindex(ss}$index) then r[ss~index] =ss~item ss~next end

cs=ab~supplier

do while cs~available if $_{hasindex(cs}$index) then r[cs~index] =cs~item es-next end

return r

::routine CommoniIntersect /* Returns a new collection of the same class as SELF that contains the items from SELF that have indexes also in the argument.

/ / Actually an index in SELF can only be ‘matched’ with one in the argument once. Hence copy and removal. / *This = arg(1)* / self of caller */ w= .Bag~new sc=This~supplier do while sc~available wlsc~index] =sc~index sc-next end r=This$_{class}$new

cs=MayEnBag (arg(2))~supplier do while cs~available i=cs~index if w~hasindex(i) then do r[i]=This [i] w~remove (i) end cs~next end return r

::routine CommonUnion /* Returns a new collection of the same class as SELF that contains all the items from SELF and items from the argument that have an index not in the first. / / Best to add them all. By adding non-receiver first we ensure that receiver takes priority when same indexes. / *This = arg(1)* / self of caller */ r=This$_{class}$new cs=MayEnBag (arg(2))~supplier do while cs~available r[cs~index] =cs~item cs~next end cs=This~supplier do while cs~available r[cs~index] =cs~item cs~next end return r

::routine CommonDifference /* Returns a new collection containing only those index-item pairs from the SELF whose indexes the other collection does not contain. / *This = arg(1)* / self of caller */ r=This$_{class}$new cs=This~supplier do while cs~available r[cs~index] =cs~item es-next end cs=MayEnBag (arg(2))~supplier do while cs~available r~remove (cs~index) es-next end return r

::routine MayEnBag

/* For List and Queue the indexes are dropped. */ rearg(1) if r-clags == .List | r-class == .Queue then r=EnBag(r) return r

::routine EnBag r=.Bag~new s=arg(1)~supplier do while s~available if arg(1)~class == .List | arg(1)~class == .Queue then r[s~item] =s~item else /* This case is when the receiver is a Bag. */ r[s~index] =s~index s~-next end return r

14.1.2 The collection class

::Cclass ‘Collection’

INIT

::method init

expose a /* A collection is modelled as using 3 slots in an array for each element. The first slot holds the item, the second the index, and the third is used by particular types of collection. This order of slots is arbitary, chosen to match order of arguments for PUT and SUPPLIER~NEW. / / The first set of 3 slots is reserved for other purposes, to avoid having separate variables which the subclassing would need to access. */

a=.array~new

a[1] /*ItemsCount*/=0

a[2]/*Unique*/=0

return self

EXPOSED

::method exposed private expose a

/* This method allows subclasses to get at the implementation of Collection. */
return a

FINDINDEX

::method findindex private expose a /* Returns array index if the collection contains any item associated with the index specified or returns 0 otherwise. / *do j=4 by 3 to 1+3*a[1]/*ItemsCount*/ if alj+l]==arg(1) then return j end j return 0

AT

::method at /* vANY / *expose a* / Returns the item associated with the specified index. */ j=self~findindex(arg(1)) if j=0 then return .nil return a[j]

[]

::method '[]' /* Synonym for the AT method. */ forward message 'AT'

PUT

::method put /* rANY rANy / *expose a use arg item, index* / Replaces any existing item associated with the specified index with the new item. Otherwise adds the item-index pair. / *j=self~findindex (index) if j>0 then do alj]=item return end a[1]* /ItemsCount*/=a [1]* /ItemsCount*/+1 j=1+3*al[1] /*ItemsCount*/ alj]=item a[lj+1] =index a[j+2]=0 return /* Error 91 in OOI if context requiring result. */

[]=

:smethod '[]=' /* Synonym for the PUT method. */ forward message 'PUT'

HASINDEX

::method hasindex /* vANY / / Returns 1 (true) if the collection contains any item associated with the index specified or returns 0 (false) otherwise. */

return self~findindex (arg(1))>0

ITEMS

::method items expose a

/* Returns the number of items in the collection. / *return a[1]*/ItemsCount*/

REMOVE

::method remove /* vANY */

expose a /* Returns and removes from a collection the member item with the specified index. */

j=self~findindex(arg(1))

if j=0 then return .nil

r=al[j]

self~removeit (j)

return r

REMOVEIT

::method removeit private

expose a

use arg j

/* Remove relevant slots from the array, with compaction. */

do j=j+3 by 3 to 14+3*a[1]*/ItemsCount*/ alj-3l=saljl;alj-2]=al[j+1];alj-ll=alj+2] end j

a[1] /*ItemsCount*/=a [1] /*ItemsCount*/-1

return

MAKEARRYA

::method makearray

expose a /* Returns a single-index array containing the receiver list items. */ *r= .array~new* / To build result in. */

do j=4 by 3 to 1+3*a[1]*/ItemsCount*/ r[r~dimension(1)+1]=a[j] end j

return r

MAKEARRAYX

::method makearrayx private expose a /* Returns a single-index array containing the receiver index items. */ *r= .array~new* / To build result in. */ *do j=4 by 3 to 1+3*a[1]/*ItemsCount*/ r[r~dimension(1)+1]=a[j+1] end j return r

SUPPLIER

::method supplier expose a

/* Returns a supplier object for the list. */ return .supplier$_{new(self}$makearray: .collection, self~makearrayx)

14.1.3 Class list

::¢class ‘List’ subclass Collection

PUT

::method put /* vANY rANY */ use arg item, index a=self~exposed

/* PUT for a List must not be an insertion. */ j=self~findindex (index) if j=0 then call Raise ‘Syntax’,93.918 alj]=item return

OF

::method of class /* 1 or more oANY Are they omittable? Not in I00 / / Returns a newly created list containing the specified value objects in the order specified. */ r= self ~ new do j = 1 to arg() r~ insert (arg(j)) end j return r

INSERT

::method insert /* rANY oANY */

use arg item, index

a=self~exposed /* Returns a list-supplied index for a new item, of specified value, which is added to the list. The new item follows the existing item with the specified index in the list ordering. / / Establish the index of what preceeds the new element. / / If there was no index given, the new item becomes the last on list. / / .nil argument means first */

if arg(2,‘E’) then p=arg(2)

else p=self~last

/* Convert from list index to underlying array index. */

if p==.nil then j=1

else j=self~findindex(p)

if j=0 then call Raise ‘Syntax’,93.918

j=j+3 /* Where new entry will be. / / Move space to required place. */

a[1] /*ItemsCount*/=a [1] /*ItemsCount*/+1

do k=1+3*a[1]*/ItemsCount*/ by -3 to j+3

a[k] =a[k-3] ;a[k+1] =a[k-2] ;a[k] =a[k-3] end

/* Insert new element. */

alj]=item /* A new, unique, index is needed. / / The basic requirement is for something unique, so this would be correct:

is.object~new /* a unique object, used as a key (the index on the list) /

/ /* However, a number can be used. (At risk of the user thinking it is sensible to do arithmetic on it.) */

a[j+1]=a[2]/*Unique*/;a[2] /*Unique*/=a[2] /*Unique*/+1

a[j+2]=0

return a[j+1]

FIRST

::method first a=self~exposed

/* Returns the index of the first item in the list. */ *if a[1]*/ItemsCount*/=0 then return .nil

141 return a[5]

LAST

::smethod last a=self~exposed

/* Returns the index of the last item in the list. */ *if a[1]*/ItemsCount*/=0 then return .nil return a[3*a[1]*/ItemsCount*/+2]

FIRSTITEM

::method firstitem a=self~exposed

/* Returns the first item in the list. */ *if a[1]*/ItemsCount*/=0 then return .nil return a[4]

LASTITEM

::method lastitem a=self~exposed

/* Returns the last item in the list. */ *if a[1]*/ItemsCount*/=0 then return .nil return a[3*a[1]*/ItemsCount*/+1]

NEXT

::method next /* vANY */ *a=self~exposed* / Returns the index of the item that follows the list item having the specified index. */ *j=self~findindex(arg(1)) if j=0 then call Raise 'Syntax',93.918 j=j+3 if j>3*a[1]/*ItemsCount*/ then return .nil /* Next of last was requested. */ return a[j+1]

PREVIOUS

::method previous /* vANY */

a=self~exposed /* Returns the index of the item that precedes the list item having the specified index. */

j=self~findindex(arg(1))

if j=0 then call Raise ‘Syntax’,93.918

j=j-3

if j<4 then return .nil /* Previous of first was requested. */

return a[j+1]

SECTION

::method section /* rANY oWHOLE>=0 */

a=self~exposed /* Returns a new list containing selected items from the receiver list. The first item in the new list is the item corresponding to the index specified, in the receiver list. */

j=self~findindex(arg(1))

if j=0 then call Raise ‘Syntax’,93.918

r= .list~new /* To build result in. */ *if arg(2,‘E’) then s = arg(2) else s = self~items; do s r~insert (a[j]) j=j+3 if j>1+3*a[1]/*ItemsCount*/ then leave end return r

14.1.4 Class queue

::class ‘Queue’ subclass Collection

/* A queue is a sequenced collection with whole-number indexes. The

indexes specify the position of an item relative to the head (first item) of the queue. Adding or removing an item changes the association of an index to its queue item. */

PUSH

::method push /* vANY */

/* Adds the object value to the queue at its head. */ *a=self~exposed a[1]* /ItemCount/*=a [1]* /ItemCount*/+1

/* Slide along to make a space. */ *do j=1+3*a[1]/*ItemCount*/ by -3 to 7

aljl=alj-3] alj+l]=a[j-2]+1; /* Index changes */ *end j a[4]=arg(1) a[5J=1 return 12.2.4.2 PULL]=a[j+4]-1;* / Index changes */ end j return r

QUEUE

::method queue /* vANY / / Adds the object value to the queue at its tail. / *a=self~exposed a[1]* /ItemCount*/=a [1]* /ItemCount*/+1 a[1+3*a[1] /*ItemCount*/]=arg(1) a[2+3*a[1] /*ItemCount*/]=a[1]*/ItemCount*/ return

PEEK

::method peek

a=self~exposed /* Returns the item at the head of the queue. The collection remains unchanged. */

return a[4]

REMOVE

::method remove /* rcWHOLE>O / / Returns and removes from a collection the member item with the specified index. */ *a=self~exposed if a[1]*/ItemCount*/<arg(1) then return .nil r=self~remove: super (arg(1))* / Reset the indexes. / *k=0 do j=4 by 3 to 1+3*a[1]/*ItemsCount*/ k=k+1 alj+l]=k end j return r

14.1.5 Class table

::Class ‘Table’ subclass Collection

MAKEARRAY

::method makearray /* Returns a single-index array containing the index objects. / / This is different from Collection MAKEARRAY where items rather than indexes are in the returned array. */ forward message ‘MAKEARRAYX’

UNION

::method union /* rCOLLECTION */ return CommonUnion (self,arg(1))

INTERSECTION

::method intersection /* rCOLLECTION */ return CommoniIntersect (self,arg(1))

XOR

::method xor /* rCOLLECTION */ return CommonXor (self,arg(1))

DIFFERENCE

::method difference /* rCOLLECTION */ return CommonDifference(self,arg(1))

SUBSET

::method subset /* rCOLLECTION */ return self$_{difference(arg(1))}$items = 0

Class set

::¢class ‘Set’ subclass table

/* A set is a collection that restricts the member items to have a value that is the same as the index. Any object can be placed in a set. There can be only one occurrence of any object in a set. */

PUT

/* Second arg same as first. Committee has dropped second? / *::method put* / YANY oANY / / Makes the object value a member item of the collection and associates it with specified index. */ *if arg(2,‘E’) then if arg(2)==arg(1) then signal error* / 949 */ self~put:super(arg(1),arg(1))

OF

::method of class /* 1 or more rANYy / / Returns a newly created set containing the specified value objects. */ r=self~new do j=1 to arg() r~put (arg(j)) end j return r

UNION

::method union /* rCOLLECTION */ return CommonUnion (self, EnBag(arg(1)))

INTERSECTION

::method intersection /* rCOLLECTION */ return CommoniIntersect (self,EnBag(arg(1)))

XOR

::method xor /* rCOLLECTION */ return CommonXor (self, EnBag(arg(1)))

DIFFERENCE

::method difference /* rCOLLECTION */ return CommonDifference (self, EnBag(arg(1)))

14.1.6 Class relation

::Class 'Relation' subclass Collection

PUT

::method put /* vANY rANY */

use arg item, index

a=self~exposed /* Makes the object value a member item of the relation and associates it with the specified index. If the relation already contains any items with the specified index, this method adds a new member item value with the same index, without removing any existing members */

a[1] /*ItemsCount*/=a [1] /*ItemsCount*/+1

j=1+3*al[1]* /ItemsCount*/

alj]=item

a[lj+1] =index

a[j+2]=0

return /* Error 91 in OOI if context requiring result. */

ITEMS

::method items /* oANY / *a=self~exposed* / Returns the number of relation items with the specified index. If you specify no index, this method returns the total number of items associated with all indexes in the relation. */ *if* arg$(1, 'E')thenreturna[1]/ItemsCou$
$a[j+1]thenn = n+1endjreturnn$

MAKEARRAY

::method makearray forward message 'MAKEARRAYX'

SUPPLIER

::method supplier / oANY* / a=self~exposed / *Returns a supplier object for the collection. If an index is specified, the supplier enumerates all of the items in the relation with the specified index.* / m=.array~new / *For the items* / r=.array~new / *For the indexes* / do j=4 by 3 to 1+3*a[1]*/ItemsCount*/ if arg(1,'E') then if arg(1)=s=a[j+1] then iterate n=r~dimension(1)+1*

m[n] =a[j] r[n] =a[j+1]

end j return .supplier~new(m,r) 12.2.7.5 UNION ::method union / rCOLLECTION */*

/ Union for a relation is just all of both. */ r=self$_{class}$new cs=self~supplier do while cs~available*

r[cs~index] =cs~item es-next end cs=MayEnBag (arg(1))~supplier do while cs~available r[cs~index] =cs~item es-next end return r

INTERSECTION

::method intersection / rCOLLECTION */*

/ Intersection for a relation requires the items as well as the keys to match. */*

r=self$_{class}$new sc=self$_{class}$new cs=self~supplier do while cs~available sc [cs~index] =cs~item cs~next end cs=MayEnBag (arg(1))~supplier do while cs~available if sc$_{hasitem(cs}$item,cs~index) then r [ecs~index] =sc$_{removeitem(cs}$item, cs~next end return r

cs~index)

XOR

::method xor / rCOLLECTION / / Returns a new relation that contains all items from self and*

*the argument except that all index-item pairs that appear in both collections are removed. */*

r=self$_{class}$new cs=self~supplier do while cs~available r[cs~index] =cs~item cs~next end cs=MayEnBag (arg(1))~supplier do while cs~available if self$_{hasitem(cs}$item,cs~index) then r$_{removeitem(cs}$item, cs~index) else r[cs~index] =cs~item cs~next end return r

DIFFERENCE

::method difference / rCOLLECTION */*

/ Returns a new relation containing only those index-item pairs from the g Y P SELF whose indexes the other collection does not contain. */ r=self$_{class}$new cs=self~supplier*

do while cs~available r[cs~index] =cs~item cs~next end

cs=MayEnBag (arg(1))~supplier

do while cs~available r$_{removeitem(cs}$item, cs~index) cs~next end

return r

SUBSET

::method subset / rCOLLECTION */ return self$_{difference(arg(1))}$items = 0*

REMOVEITEM

::method removeitem / YANY rANY* / a=self~exposed / *Returns and removes from a relation the member item value (associated with the specified index). If value is not a member item associated with index index, this method returns the NIL object and removes no item.* / do j=4 by 3 to 1+3*a[1]*/ItemsCount*/ if alj]==arg(1) & a[j+1l]==arg(2) then do self~removeit (j) return arg(1) end end j return .nil*

INDEX

::method index / vANY* / a=self~exposed / *Returns the index for the specified item. If there is more than one index associated with the specified item, the one this method returns is not defined.* / do j=4 by 3 to 1+3*a[1]*/ItemsCount*/ if arg(1)==a[j] then return a[j+1] end j return .nil*

ALLAT

::method allat / vANY* / a=self~exposed / *Returns a single-index array containing all the items associated with the specified index.* / r=.array~new do j=4 by 3 to 1+3*a[1]*/ItemsCount*/ if arg(1)==a[j+1] then r[r~dimension(1)+1] = aljl end j return r*

HASITEM

::method hasitem / YANY rANY* / a=self~exposed / *Returns 1 (true) if the relation contains the member item value (associated with specified index). Returns 0 (false) otherwise.* / do j=4 by 3 to 1+3*a[1]*/ItemsCount*/ if alj]==arg(1) & a[j+l]==arg(2) then return 1 end j return 0*

ALLINDEX

::method allindex / vANY */*

a=self~exposed / Returns a single-index array containing all indexes for the specified item.* / r=.array~new do j=4 by 3 to 1+3*a[1]*/ItemsCount*/ if alj]==arg(1) then do r[r-~dimension (1)+1] =a[j+1] end end j return r*

##3 The bag class

::class 'Bag' subclass relation

/ A bag is a collection that restricts the member items to having a value that is the same as the index. Any object can be placed in a bag, and the same object can be placed in a bag multiple times. */*

OF

::method of class / 1 or more rANYy / / Returns a newly created bag containing the specified value objects. */ r=self~new do j=1 to arg() r~put (arg(j)) end j return r*

PUT

::method put / vANY oANY / / Committee does away with second argument? / / Makes the object value a member item of the collection and associates it with the specified index. If you specify index, it must be the same as value. */ if arg(2,'E') then if arg(2)=s=arg(1) then signal error self~put: super (arg(1),arg(1))*

UNION

::method union / rCOLLECTION */ return CommonUnion (self, EnBag(arg(1)))*

INTERSECTION

::method intersection / rCOLLECTION */ return CommoniIntersect (self,EnBag(arg(1)))*

XOR

::method xor / rCOLLECTION */ return CommonXor (self, EnBag(arg(1)))*

DIFFERENCE

::method difference / rCOLLECTION */ return CommonDifference (self, EnBag(arg(1)))*

14.1.7 The directory class

::class 'Directory' subclass Collection

AT

::method at / vANY */ a=self~exposed*

/ Returns the item associated with the specified index. */ j=self~findindex(arg(1))*

if j=0 then return .nil

/ Run the method if there is one. */ if a[j+2] then return self~run(a[j]) return a[j]*

PUT

::method put / vANY rANY /* a=self~exposed */ Makes the object value a member item of the collection and associates it with the specified index. */ if* arg(2)~$hasmethod(`MAKESTRING')thencallRaise`Syntax', 93.938self$~$put : super(arg(1), arg(2)$~$makestring)return$

MAKEARRAY

::method makearray forward message 'MAKEARRAYX'

SUPPLIER

::method supplier a=self~exposed / Returns a supplier object for the directory. / / Check out what happens to the SETENTRY fields. /* r=.array~new /* For items /* do j=4 by 3 to 1+3a[1]/*ItemsCount*/ r[r~dimension(1)+1]=a[j]*

end j return .supplier$_{new(r,self}$makearray) #### UNION ::method union / rCOLLECTION */*

return CommonUnion (self,arg(1))

INTERSECTION

::method intersection / rCOLLECTION */ return CommoniIntersect (self,arg(1))*

XOR

::method xor / rCOLLECTION */ return CommonXor (self,arg(1))*

DIFFERENCE

::method difference / rCOLLECTION */ return CommonDifference(self,arg(1))*

SUBSET

::method subset / rCOLLECTION */ return self$_{difference(arg(1))}$items = 0*

SETENTRY

::smethod setentry / rSTRING oANY */*

a=self~exposed / Sets the directory entry with the specified name (translated to uppercase) to the second argument, replacing any existing entry or method for the specified name. */*

n=translate(arg(1))

j=self~findindex(n)

if j=0 & arg$(2, `E')then return$

if arg$(2, `E')then do/ * Removal * /$

self~removeit (j)

return end

if j=0 then do / It's new /* a[1]/*ItemsCount*/=a[1]/*ItemsCount*/ +1 j=1+3*al[1]* /ItemsCount/ *alj+l]=n end*

alj]=arg(2)

a[4j+2]=0

return

ENTRY

::smethod entry / rSTRING /* a=self~exposed /* *Returns the directory entry with the specified name (translated to uppercase). /* n=translate(arg(1)) j=self~findindex(n) /if j=0 then signal error according to online / / Online has something about running UNKNOWN. /* if j=0 then return .nil /* *If there is an entry decide whether to invoke it. */ if a~hasindex(j) then do if [j+2] then return al[j] return self~run(al[jl]) end*

HASENTRY

::method hasentry / rSTRING / / Returns 1 (true) if the directory has an entry or a method for the specified name (translated to uppercase) or 0 (false) otherwise. */*

return self~findindex (translate(arg(1)))>0

SETMETHOD

::method setmethod / rSTRING oMETHOD /* a=self~exposed /* *Associates entry with the specified name (translated to uppercase) with method method. Thus, the language processor returns the result of running method when you access this entry. / / (Part of METHOD checking converts string or array to actual method.) /* n=translate(arg(1)) j=self~findindex(n) if j=0 &* arg$(2, `E')thenreturnif$ arg$(2, `E')thendo$ $0thendo/It'snew/$ *a[1]/*$ItemsCount$*/=a[1]/*$ItemsCount$*/ +1 j=1+3*$al[1]/$*ItemsCount*$/alj+l] = nendalj] = arg(2)a[j + 2] = 1return$

UNKNOWN

::method unknown /* rSTRING rARRAY */

/* Runs either the ENTRY or SETENTRY method, depending on whether the message name supplied ends with an equal sign. If the message name does not end with an equal sign, this method runs the ENTRY method, passing the message name as its argument. */

if right (arg(1),1)=s='=' then

return self~entry(arg(1)) /* 2?? Not clear whether second argument is mandatory. */ t=.nil

if arg(2,'E') then t=arg(2) [1] self~setentry (left (arg(1),length(arg(1))-1),t) 12.3 The stem class For some reason, the stem class doesn't have PUT and AT methods, which stops us having a general rule about [] synonyms AT, []= synonyms PUT. Anyway, committee doing without this class as such.

Here is temporary stuff showing how to use algebra in the collection coding.

/* This 1998 version uses Rony's rules for XOR and INTERSECTION based on UNION and DIFFERENCE */

/* Test Set-Operator-Methods on different collection objects */

/* This top part has some rough parts - not meant for standard. */

/* The dumps put out results sorted, so that comparisons can be made between implementations that keep collections in different orders. */

/* Invocation example: settest.cmd 1> tmp.res 2> tmp.err */

/* Jnitial verification that new definitions are in effect */ J18list = .List~new if 18list~hasmethod("J18") then signal error

/* Input collections used for the tests */

coll.1 = .array~of(1, 2„ 4)

coll.2 = list~of(2, 3, 6)

coll.3 = .queue$_{newPUSH(2)PUSH(3)}$~PUSH(7)

coll.4 = .directory$_{new}$~setentry(1, "eins")~~setentry(3, "drei")

coll.5 = .bag$_{newput(2)put(3)put(5)}$~put(2)

coll.6 = .relation$_{new}$~"[J="("zwei", 2)"[]="(' "drei", 3)"[J="('vier", 8)~~"J="C"drei",3) coll.7 = .set~of(2, 3, 9)]=tmpSupp~ITEM tmpSupp~NEXT END do until hope hope=1 do j=1 to k~dimension(1)-1 if k[j]$_{string>k[j+1]}$string |, (k[j]$_{string=k[j+1]}$string & i[j]$_{string<i[j+1]}$string) then do]=t hope=0 end end end if O=collection~items then say" The result is empty!" else do j=1 to k~dimension(1) SAY " " "index" pp(k[j]) "item" ppd[j)]=alj+1];a[j- 1]=alj+2] end j

a[1]/*ItemsCount*/=a[1]/*ItemsCount*/-1

return

::method makearray expose a /* Returns a single-index array containing the receiver list items. */ *r=.array~new* = / To build result in. */ *do j=4 by 3 to 143*a[1]/*ItemsCount*/ r[r~dimension(1)+1]=a[j] end j returm r

::method makearrayx private expose a

/* Returns a single-index array containing the receiver index items. */ *r=.array~new* = / To build result in. */ *do j=4 by 3 to 143*a[1]/*ItemsCount*/

r[r~dimension(1)+1]=a[j+1] end j return r

:imethod supplier expose a /* Returns a supplier object for the list. */ return .supplier$_{new(self}$makearray:.collection,self~makearrayx)

::class ‘List’ subclass Collection

zimethod J18 = /* Here to demonstrate .LIST is replaced */ return

/* List and Queue are special because there is an order to their elements. */

::method put /* rANY rANY */ use arg item, index a=self~exposed

/* PUT for a List must not be an insertion. */ j=self~findindex(index) if j=0 then call Raise ‘Syntax’,93.918 a[j]=item retum

zimethod of class /* 1 or more oANY Are they omittable? Not in IOO / / Returns a newly created list containing the specified value objects in the order specified. */ r=self ~ new do j = 1 to argQ) r ~ insert(arg(j)) end j return r

zimethod insert /* rANY oANY */

use arg item, index

a=self~exposed /* Returns a list-supplied index for a new item, of specified value, which is added to the list. The new item follows the existing item with the specified index in the list ordering. / / Establish the index of what preceeds the new element. / / Tf there was no index given, the new item becomes the last on list. / / mil argument means first */

if arg(2,‘E’) then p=arg(2)

else p=self~last

/* Convert from list index to underlying array index. */

if p==.nil then j=1 else j=self~findindex(p)

if j=0 then call Raise ‘Syntax’,93.918

j=J+3 /* Where new entry will be. / / Move space to required place. */

a[1]/*ItemsCount*/=a[1]/*ItemsCount*/+1

do k=1+3*a[1]*/ItemsCount*/ by -3 to j+3

a[k]=a[k-3];a[k+1]=a[k-2];a[k]=a[k-3] end

/* Insert new element. */

a[j]=item /* A new, unique, index is needed. / / The basic requirement is for something unique, so this would be correct:

i=.object~new /* a unique object, used as a key (the index on the list) / / /* However, a number can be used. (At risk of the user thinking it is sensible to do arithmetic on it.) */

a[j+1]=a[2]/*Unique*/;a[2]/*Unique*/=a[2]/*Unique*/+1

a[j+2]=0

return a[j+1]

:smethod first a=self~exposed

/* Returns the index of the first item in the list. / *if a[1]*/*ItemsCount*/=0 then return

.nil return a[5]

::method last a=self~exposed

/* Returns the index of the last item in the list. / *if a[1]*/ItemsCount*/=0 then return .nil return a[3*a[1]*/ItemsCount*/+2]|

::method firstitem a=self~exposed

/* Returns the first item in the list. / *if a[1]*/ItemsCount*/=0 then return .nil return a[4]

::method lastitem a=self~exposed

/* Returns the last item in the list. / *if a[1]*/ItemsCount*/=0 then return .nil return a[3*a[1]/*ItemsCount*/+1 |] a[j+1]=al[j+4]-1; /* Index changes */ end j returm r

zimethod queue =/* rANY */

/* Adds the object value to the queue at its tail. / *a=self~exposed a[1]*/ItemCount*/=a[1 |/ItemCount*/+1 a[{1+3*a[1]*/ItemCount*/|=arg(1) a[2+3*a[1]*/ItemCount*/|=al[1]*/ItemCount*/ return] r[n]=a[j+1] end j return .supplier~new(m,r)]=1

returm

:imethod unknown =/* rSTRING rARRAY */

/* Runs either the ENTRY or SETENTRY method, depending on whether the message name supplied ends with an equal sign. If the message name does not end with an equal sign, this method runs the ENTRY method, passing the message name as its argument. */ if right(arg(1),1)==‘=“’ then

return self~entry(arg(1)) /* 22 Not clear whether second argument is mandatory. */ t=.nil

if arg(2,‘E’) then t=arg(2)[1] self~setentry(left(arg(1),length(arg(1))-1),t)

routine CommonXor /* Returns a new collection that contains all items from self and the argument except that all indexes that appear in both collections are removed. / / When the target is a bag, there may be an index in the bag that is duplicated and the same value as an index in the argument. Should one copy of the index survive in the bag? */ lhs=arg(1)~difference(arg(2)) rhs=Cast(arg(1),MayEnBag(arg(2)))~difference(arg(1)) return lhs~union(rhs)

nroutine CommonUnion /* Returns a new collection of the same class as SELF that contains all the items from SELF and items from the argument that have an index not in the first. / / Best to add them all. By adding non-receiver first we ensure that receiver takes priority when same indexes. / *This = arg(1)* / self of caller */ r=This$_{class}$new cs=MayEnBag(arg(2))~supplier do while cs~available r[cs~index]=cs~item cs~next end cs=This~supplier do while cs~available r[cs~index]=cs~item cs~next end return r

nroutine CommonDifference /* Returns a new collection containing only those index-item pairs from the SELF whose indexes the other collection does not contain. / *This = arg(1)* / self of caller */ r=This$_{class}$new cs=This~supplier do while cs~available r[cs~index]=cs~item cs~next end cs=MayEnBag(arg(2))~supplier

do while cs~available r$_{remove(cs}$index) cs~next end

return r

routine MayEnBag

/* For List and Queue the indexes are dropped. */ r=arg(1) if r~class == .List | r~class == .Queue then r=EnBag(r) retum r

routine EnBag r=.Bag~new s=arg(1)~supplier do while s~available if arg(1)~class == .List | arg(1)~class == .Queue then 1[s~item]=s~item else /* This case is when the receiver is a Bag. */ 1[s~index]=s~index s~next end return r

/* This Cast routine commented away, since replaced by Oct 98 Rony version. routine Cast public use arg Target, Other TmpColl = Target$_{class}$new /* Create an instance of type Target / *TmpSupp = Other~supplier* / Get supplier from Other */ signal on syntax do while TmpSupp~available TmpColl[TmpSupp~index] = TmpSupp~item TmpSupp~next end return TmpColl

/* Tf syntax error 93.949, then target is an index-only collection like a set.*/ syntax:

if condition(“O”)~code = “93.949” then signal IndexOnly

raise propagate /* Unhandled syntax error, raise in caller */

IndexOnly: /* This for index-only collections. */ do while TmpSupp~available TmpColl[TmpSupp~index] = TmpSupp~index TmpSupp~next

end return TmpColl End commented away */

/* 98-09-24, —ref; CAST2.CMD return a collection of type “target” which collected all item/index pairs of the argument “other” */

2: ROUTINE cast PUBLIC USE ARG target, other

SIGNAL ON SYNTAX IF other ~ HASMETHOD(“SUPPLIER”) THEN RAISE SYNTAX 98.907 ARRAY (“COLLECTION (i.e. argument2=‘other’-object must have a ‘SUPPLIER’-method)”)

tmpColl = target ~ CLASS ~ NEW /* create a an instance of type target / *tmpSupp = other~ SUPPLIER* / get supplier from other */

/* is index of “other” usable ? */ bIndexUsable = other ~ HASMETHOD(“UNION”)

IF .Debug = .true THEN IF bIndexUsable THEN SAY” /// index of ‘other’ not usable for setlike-operations”

/* possible syntax-error, if index and item must have the same value, e.g. for sets/bags */ *SIGNAL ON SYNTAX NAME INDEX ONLY target ~ CLASS ~ NEW ~ PUT(1,2)* / test, if target-type is index-only */

SIGNAL ON SYNTAX DO WHILE tmpSupp ~ AVAILABLE IF bIndexUsable THEN tmpColl[tmpSupp ~ INDEX] = tmpSupp ~ ITEM ELSE tmpColl[tmpSupp ~ ITEM] = tmpSupp ~ ITEM tmpSupp ~ NEXT END RETURN tmpColl

INDEX_ONLY : /* this is for index-only collections (e.g. sets, bags) */ SIGNAL ON SYNTAX IF .Debug = .true THEN SAY” \‘target’ is an index-only collection (index==item)” DO WHILE tmpSupp ~ AVAILABLE

IF bIndexUsable THEN tmpColl[tmpSupp ~ INDEX] = tmpSupp ~ INDEX ELSE tmpColl[tmpSupp ~ ITEM] = tmpSupp ~ ITEM tmpSupp ~ NEXT END RETURN tmpColl

SYNTAX: RAISE PROPAGATE /* raise error in caller */

14.2 The stream class

The stream class provides input/output on external streams. ::class stream

::method init /* rString / *Initializes a stream object for a stream named name, but does not open the stream. ::smethod query* / keywords */

There is also QUERY as command with method COMMAND.

Used with options, the QUERY method returns specific information about a stream. ::method charin

::smethod charout

::method chars

::method linein

::method lineout

::method lines

::method qualify

::method command /* rString */

Returns a string after performing the specified stream command. ::method open

There is also OPEN as command with method COMMAND.

Opens the stream to which you send the message and returns “READY:”.

Committee dropping OPEN POSITION QUERY SEEK as methods in favour of command use. ::method state

Returns a string that indicates the current state of the specified stream. ::method say

::method uninit

::method position /* Ugh / *POSITION is a synonym for SEEK. ::method seek* / Ugh */

Sets the read or write position a specified number (offset) within a persistent stream.

::method flush

Returns “READY:”. Forces any data currently buffered for writing to be written to the stream receiving the message.

There is also FLUSH as command with method COMMAND.

Committee dropping FLUSH.

::method close

Closes the stream that receives the message.

There is also CLOSE as command with method COMMAND.

Semantics are ‘seen by other thread’. ::method string

::method makearray /* rCHARLINE */ Returns a fixed array that contains the data from the stream in line or character format, starting from the

current read position. ::method supplier

Returns a supplier object for the stream. ::method description

::smethod arrayin /* rCHARLINE */

Mixed case value works on OOI.

Committee dropping Arrayin & Arrayout. Arrayin == MakeArray

Returns a fixed array that contains the data from the stream in line or character format, starting from the

current read position. ::smethod arrayout /* rARRAY rCHARLINE */

Returns a stream object that contains the data from array.

14.3 The alarm class

::class alarm

::method init /* Time, Msg */ Sets up an alarm for a future time atime. ::method cancel

Cancels the pending alarm request represented by the receiver. This method takes no action if the specified time has already been reached.

14.4 The monitor class

The Monitor class forwards messages to a destination object.

-local [‘OUTPUT’] = .monitor~new(.output)

::class monitor

14.4.1 INIT

Initializes the newly created monitor object.

::method init /* oDESTINATION */ expose Destination Destination = .queue~new if arg(1,‘E’) then Destination~push (arg(1)) return

14.4.2 CURRENT

Returns the current destination object.

::smethod current expose Destination return Destination [1]

14.4.3 DESTINATION

Returns a new destination object.

::method destination /* oDESTINATION */ expose Destination if arg(1,‘E’) then Destination~push (arg(1)) else Destination~pull return Destination [1]

14.4.4 UNKNOWN

Reissues or forwards to the current monitor destination all unknown messages sent to a monitor object

::method unknown expose Destination

Extra parens needed here in original OREXX syntax

forward to destination[1] message arg(1) arguments arg(2) return

15

Rationale

This annex explains some of the decisions made by the committee that drafted this standard, and assists in the understanding of this document. Some of the statements made here are opinions rather than facts. These should be interpreted as if prefixed by “In the opinion of the X3J18 committee…”.

The language described in this standard is, almost entirely, a compatible extension of the language described by the third reference of Annex C, which we call “Classic Rexx”.

The extension allows programs to be written in a less monolithic fashion; “Directives” are introduced to allow one file to contain several executable units and to allow a program to be written as several files. The functional extension centers on the addition of objects. Unlike the individual strings which are the data of Classic Rexx, an object may be composite. The use of identifiers to reference objects is an indirect reference, that is two identifiers may refer to the same object. Classic Rexx avoided any aliasing, even to the extent having by-reference parameters, to promote simple error free programming. In the years since Rexx originated the problems tackled by programmers have become more complex and data structures larger, so that the benefit of simplicity is outweighed by the power of assignment semantics that are not simply copying all the data.

Even with the addition of references Rexx remains a typeless language, in the sense that the programmer need not consider underlying hardware formats such as LONG or FLOAT representations. Object Rexx does have classes, which are the hardware independent analogy to types. The class of an object corresponds to the operations that can be performed upon it.

16

Incompatibilities

17

The incompatibilities from Classic Rexx are:

Assignment of compound variables, as in ABC. = PQR., is an assignment of references so that ABC. subsequently refers to the same object as PQR., as opposed to making the default value of ABC. that of PQR.. This change was necessary to fit compound variables into the object framework, in particular allowing USE ARG to handle compound variables as by-reference parameters. The first reference of Annex B discouraged use of this construct in Classic Rexx programs. “Breakage” of programs due to this incompatibility is rare.

Also something in condition handling that I don’t know the reason for.

17.1 Call

The call instruction has been extended to allow for a computed name of the callee. Syntax considerations prevent a similar thing being done for functions.

17.2 Concurrency

Meet 17 minutes

17.3 Guard

Meet 17 minutes

18

To be processed:

The following decisons are abstracted from minutes. We need to ensure they are covered in the main standard and their rationale appropriately reworded for this annex.

Aliasing. Assignment is viewed as making the target reference the same object as the source. Hence the object (and changes to it) may be accessed through more than one name. For ‘immutable’ objects a changed version of an object can only be produced by creating a new object. For compatibility with classic Rexx, strings are immutable objects. Non-strings may or may not be immutable. Note that there is an alternative model in which distinction is made between assignments which copy values and assignments which copy references. This alternative was not chosen; the committee prefered the model in which all data names are naming references (which may be implicitly followed to values).

Arguments ‘by-reference’. The introduction of aliasing makes this natural although the detail has simple-versus-general contentions. (Is it necessary for simple strings to be passed by reference. Encapsulation. An object may ‘own’ some variables and access to those may be limited (so that re-implementation of the object could use different variables without upsetting the usage of the object). Classess. There will be ‘factory’ objects capable of creating multiple new objects which have common characteristics about how they can be used.

Inheritance and hierachy. The semantics of a clas may be specified by adding to the semantics of another class. This relation is used to form a tree. We prefer a singly rooted tree, rooted in the class ‘Object’ which is built-in to the language. Other classes will also be built-in. Experience with OOM and other languages is that unrestricted inheritance by one class from multiple classes does not work in the way the coder intended (the implementations of the classes do not combine successfully). If multiple inheritance is added to Rexx at all, it will be in the cautious ‘MIXIN’ flavor of OOI.

Messaging: Executing some labelled code which is associated with objects of a given class is a form of invocation that is sufficiently different from classic Rexx to justify a new syntax construct. The new syntax is Receiver ~ MethodName(Arguments) and implies both a different search for the method to be invoked and a special role for the receiver as opposed to the other arguments of the invocation.

Packaging: In principle a ‘program builder’ could be used in developing Rexx programs with many classes and methods, and that builder could hide from the coder the details of how the configuration held the methods. However, rather than define a program builder we are choosing to define a simple method of

holding multiple classes & mthods (with specification of their hierarchy) within a single text file. The non-executable dividers in such a file are known as directives. The files are known as packages and a package may specify (by directive) that it requires another package in order to function correctly. There are questions about when initialization of required packages occurs; we intend to find a solution that does not require the complete graph of requirements to be initialized before other code is executed.

A note on the syntax of directives. When no special token (eg ::) is used to introduce directives the directives are be recognizable by the spelling of the keyword. (CLASS REQUIRES etc.) The purpose of the special token is emphasis of directives rather than implementation ease in “pre-processing” the directives.

Packages in non-Rexx. It is necessary to exploit packages that are not written in Rexx. To invoke their methods it is necessary that the package makes known to the Rexx method search the names of the classes and their methods. To do more than invoke the methods (eg to subclass the the external classes) requires complicated mechanisms and may not be a requirement.

External procedures. To allow Classic internal procedures to be separated into different files with undue change of semantics, the PROCEDURE statement will be permiteed as the first statement of a routine which is in a separate file.

Concurrency will be added, that is multiple execution cursors progressing through one program. The mechanism for creating multiple cursors will be the “early reply” where one cursor becomes two; one of two progresses by “falling through” the early reply and the other starts its progress after the site of the current invocation. Multiple cursors carry the risk of execution interleaving in a way which negates the coder’s intentions in writing so that clauses would execute sequentially. The language definition will be tightened to ensure atomicity of string assignment etc. Additionally, a set of rules about allowing two cursors on the same method at the same time will provide a reduction of the risk. Since in many cases the data which have to be maintained consistent will reside in a single object the rules are object-based. In general a cursor on a method executing against a particular object will delay any other cursor from executing methods against that object.

This rule provides sensible synchronization without much effort from the programmer but other controls may be provided:

a) Stronger control, eg only one cursor within the methods of a set of objects.

b) More detailed control, eg division of a method into sections which allow/disallow other cursors into the section.

Extended Variable Pools. The API for variable pools will need to be extended to reflect the model in which the named content in a pool is always a reference (and the reference is followed when the value of a string is required.) We note that OO! adopts a convention that names starting with ‘!’ (shriek) name objects that are not intended for access by the coder. These objects will not be standardized. Additionally some objects without shriek names are not

candidates for standardising, eg SYSTEM, .KERNEL.

A model is needed for whether changes made to methods are seen by objects created before the changes. Changes that are seen are preferable where a long-lived object is being brought up-to-date. Changes that only apply to future objects are preferable if avoiding failure of what “used to work” is the priority. In view of OOM experience the standard should allow both, on a method by method choice. (eg perhaps a bug fix applied retrospectively but not an enhancement.)

Multiple inheritance. Study of the ‘method search’ algorithm, see later, shows that this is an “add-on” that could readily be retained or omitted. That argues in principle for retention, since the non-user of multiple inheritance would not suffer from it. On the other hand it adds complexity and can be misused even in the conservative form that OOI has it.

Signature-based method search. This is not in OOI but is in languages such as Java.

Subclassing of imported classes. It is our intention to say that imported classes can be used in all the same ways as builtin classes. Because this may be impractical to implement with some external classes, a conforming language processor will have a list (which may be empty) of external classes it supports. (And hence nothing of the current SOM interface will be part of the standard.)

Persistent objects. It is our belief that support for very-long-running programs is required. It is a moot point whether the ENVIRONMENT directory is enough.

If persistent objects are to converted to a form which is platform independent, (“pickling”), there are difficulties in deciding what pointers should be followed and further objects included, as opposed to objects being assumed available on all platforms. This topic is defered.

Locking across a set of objects. In OOI this can only be done by locking the events serially, which has more risk of deadlock than locking them simultaneously. The decision was made not to add simultaneous locking.

Critical sections. The GUARD mechanism can be used in a ‘critical section’ style. Nothing will be added to the definition.

Old objects seeing new changed methods. When bugs in long running programs are fixed, there can be a benefit if old objects see the corrected methods. It seems practical to offer a variation of DEFINE for this - see method lookup discussion.

The committee does not find the current OOI approach to merging ‘classic’ stems with OO stems satisfactory. It invalidates some existing programs. (A warning about this was put in A8.3.3 of X3.274.) It produces surprises for OO programmers, eg a==b after a=.stem~new; b=.stem~new. The proposed alternative is to make the presence/absence of a dot at the end of the name determine whether coercion to string is done. The ‘classic’ meaning of A.=B. would be restored but AA=BB, AA==BB etc. would have their OO meanings. The meaning of USE ARG with a dotted name would be defined to allow ‘by reference’ passing of a stem. Square brackets could be used with both

dotted and undotted names. A further proposal is to note that this leaves few differences between the DIRECTORY class and the non-dotted STEM class so that it might be a further improvement if the DIRECTORY class was extended to the extent that the STEM class was unnecessary.

There is a potential problem which the committee has not fully analysed in the OO! treatment of SAY and streams. OOI has made features (of the STREAM bif) that were configuration determined in X3-274 into OO language methods, and has made SAY a method (undocumented?). Full analysis may show that more of I/O could (& should?) be made standard or may show that some OO! I/O language should not be standardized.

The committee discussed what parts of the OOI implementation were suitable to be defined in a standard. Potentially, all the builtin classes and objects (which are reachable from .ENVIRONMENT) might be standardized. However, names which start with an exclamation mark denote unsuitable things. The committee also thought the following unsuitable:

- Anything specific to SOM. - RX_QUEUE - Stream_Supplier - Parts of LOCAL other than direct reference to the default streams. There is a naming problem with this. The names in OOI are STDIN, STDOUT and STDERR. We would prefer INPUT, OUTPUT, and ERROR to be_ consistent with the keywords. OOI has used those names for something else. We will work on the proposal that we use the prefered names and the MONITOR class is dropped. (Users who want the monitor function can get it with a few lines of directive.)

The committee feels that OOI over-specifies the index of an item in a LIST. In OO it is a count giving the sequence over time of the insertions in the list. The risk in using numbers is that they may be (wrongly) used as positions, and arithmetic done on them. It is proposed that the index of a list item be of class OBJECT rather than of class STRING.

In OOI, the .ENVIRONMENT is global, not read-only, and contains builtin objects such as .TRUE and .FALSE. The committee regards this as too risky - suppose that .TRUE was accidentally or maliciously revalued as 0! It seems sufficient to add read-only as a characteristic of directories. (This characteristic at the element level might be expensive to implement.) Reserved symbols (X8-274 clause 6.2.3.1) also provide a mechanism for preventing the override of builtin names. It won't be possible for a standard to exactly define in a system-independent way the scopes/lifetimes of -ENVIRONMENT and .LOCAL but (as with OOl) the .LOCAL will relate to "One API_START" and -ENVIRONMENT will have a wider scope. (Power on to power off of some system'). The proposed "search order" is:

1. Things provided by the system which no user is expected to want to override. Perhaps .TRUE .FALSE NIL.
2. The .LOCAL read/write directory, initialized with the default streams, changable by the user for individual program executions. Perhaps METHODS here.
3. The read-only part of the environment, that is the builtin classes and

objects. Also .SYSTEM perhaps.

4. The read/write ENVIRONMENT directory. Changable by programmers co-operating at the system level. Final placement of all builtins needs discussion, but the read-only true&false requirement will be met. Note that the algorithm of method lookup does not change if “old objects see newest methods” is desired. What changes is whether the method tables are updated in place or copied-and-updated when they are changed.
5. There have been sugestions to allow the REQUIRES directive appear in more places. The committee agrees with this and proposes:

A) All REQUIRES directives must appear together in the file. B) These directives may appear anywhere the OOI implementation currently allows them to appear.

2. Message numbers and prose are now allocated to messages detected by the syntax, additional to the messages known to the first standard. Most messages simply involve new minor codes sequential beyond those defined in the first standard.
3. Proposed language, eg FORWARD, METHOD, and CLASS clauses, allow for many options which can appear in any order. These can be written in the BNF (in the manner that TO BY FOR were handled in the first standard) but it is neater to extend the BNF metalanguage.
4. The OOI syntax used in the FORWARD instruction has examples of the ‘argument’ construct, which is either a symbol-or-string taken as a constant or is an expression in parentheses. The committee will define ‘term’ to be allowed in such places. This is a change to the OOI for valid programs only in the case where a MESSAGE option used a symbol intending it to be ‘taken as a constant’. (As opposed to taken as a variable with the value defaulting to its name when uninitialized.)
5. In a similar vein to 4 above, some other positions where the “variable reference” notation is used (or proposed) will be changed. It would be nice to allow “term” in all these places but ambiguity consideration means some will be “sub-expression”, ie parenthesed expression, notation.
6. The colon used for superclass specification will allow symbol-or-string to follow. DATA:
7. The model of data used in defining the first standard needs changing for OO, to:

- Variable pools are objects, objects are variable pools.
- Variable pool contents are references to objects, not values of strings.
- Pools are not numbered, they are referenced.
- The state variables (those with names beginning ‘#’ used to define processing in the standard) are present in all pools, as opposed to being in a separate pool.

This data model gives a natural interpretation to the variable pool API applied to local pools. (Local pools may access non-local pool items by reason of EXPOSE.) In principle this leads to different threads of execution (resulting

from REPLY) being able to execute the API. (In practice OOI has a restriction to executing the API only on the ‘main’ thread and the committee needs to know if this is due to a generally applicable difficulty.)

The committee considered the relevance of IBM’s “Object Rexx Programming Guide” G25H-7597-1 to the Configuration section of the standard. The material there in Appendix A under headings External Function Interface, System Exit Interface, and Variable Pool Interface was deemed material for inclusion, and the rest not. This is similar to the first standard, although there will be an extra trap, for method calls. The committee considered the relevance of the STREAM section of IBM’s “Object Rexx Reference”, G25H-7598-0. That stream class brings into the language more I/O than the original Rexx, eg an explicit CLOSE. The new standard will partially follow this trend also.

PEEK on queue unnecessary - same as AT[1]?

Also need to resolve the issues on Monitor class and on run time inspection.

19

Annex B

(informative)

Method of definition

This annex describes the methods chosen to describe Rexx for this standard.

Definitions

Definitions are given for some terms which are both used in this standard and also may be used elsewhere. This does not include names of syntax constructions; for example, group, which are distinguished in this standard by the use of italic font.

Conformance

Note that irrespective of how this standard is written, the obligation on a conforming processor is only to achieve the defined results, not to follow the algorithms in this standard.

Notation

The notation used to describe functions provided by the configuration is like a Rexx function call but it is not defined as a Rexx function call since a Rexx function call is described in terms of one of these configuration functions.

Note that the mechanism of a returned string with a distinguishing first character is part of the notation used in this standard to explain the functions; implementations may use a different mechanism. Notation for completion response and conditions

The testing of ‘X’ and ‘S’ indicators is made implicit, for brevity. Even when written as a subroutine call, each use of a configuration routine implies the testing. Thus:

call Config Time

implies

#Response = Config Time()

if left (#Response,1) == ‘X’ then call #Raise ‘SYNTAX’, 5.1, substr (#Response, 2) if left (#Response,1) == ‘S’ then call #Raise ‘SYNTAX’, 48.1, substr(#Response, 2)

Source programs and character sets

The characters required by Rexx are identified by name, with a glyph associated so that they can be printed in this standard. Alternative names are shown as a convenience for the reader.

Notation

Note that nnn is not specifying the syntax of a program; it is specifying the notation used in this standard for describing syntax.

Lexical level

Productions nnn and nnn contain a recursion of comment. Apart from this recursion, the lexical level is a finite state automaton.

Syntax level

This syntax shows a null_clause list, which is minimally a semicolon, being required in places where programmers do not normally write semicolons, for example after ‘THEN’. This is because the ‘THEN’ implies a semicolon. This approach to the syntax was taken to allow the rule ‘semicolons separate clauses’ to define ‘clauses’.

The precedence rules for the operators are built into this grammar

Data Model

The following explanation of data in terms of Classic Rexx may be helpful. References to clauses of the existing standard have 274 as a prefix.

We start with the data model from the first Standard - a number of variable pools. Two mechanisms, the external access of section 274.5.13 (API_Drop etc) and the internal of 274.7.1 (Var_Drop etc). Pools are numbered, with pool 0 reserved for reserved names (.MN etc) and pool N-1 being related to pool N as the caller’s pool. The symbols which index the pools are distinquished as tailed or non-tailed. The items in the pool have attributes ‘exposed”, ‘dropped’, and ‘implicit’. The values in the pools are string values.

An extra scope is used for ‘state variables’ used in the definition of the standard. These follow the same lookup rules in a conceptual and separate pool.

The first change necessary is to define the values in the pools as references. For string values this is just a change in definition style, since a reference always followed to a string value is semantically identical with the notion of having the value in the pool. However, references open the possibility of referencing

non-strings, which can behave in a changed way while still being refered to by the same reference. (Mutable objects)

It is reasonable that the definition should have the pools reference one another rather than use numbered pools. It is difficult to have a notion of numbering the pools when any object can have a set of variables associated with it.

Assignment is defined as assignment of references. The language could have been designed differently, for example to make assignment behave like the COPY method, but assignment of references is the natural, powerful, choice.

If pools are not numbered, the notation of the first standard, where some state variables use the #Level number as part of their names, will not suffice. An appropriate solution is to say that each variable pool can have state variables and user program variables in it. Placing the state variables that are per-procedure-level in the variable pool for their level avoids the need to specify #Level in their tails. There are pre-existing objects such as all possible values that can be written as literals and the objects accessed by .SYSTEM etc. Further

objects are created by the NEW method.

Editorial note: It looks nice to unify: an object *is* a variable pool and a variable pool *is* an object. There is some awkwardness describing the classic API_-function as applying to an object. There don't seem to be difficulties in defining any object behaviour we want in terms of state variables that refer from one object to another.

Evaluation (Definitions written as code) There is no single definitional mechanism for describing semantics that is predominantly used in standards describing programming languages, except for the use of prose. The committee has chosen to define some parts of this standard using algorithms written in Rexx. This has the advantages of being rigorous and familiar to many of the intended readers of this standard. It has the potential disadvantage of circularity - a definition based on an assumption that the reader already understands what is being defined. Circularity has been avoided by: - specifying the language incrementally, so that the algorithms for more complex details are specified in code that uses only more simple Rexx. For example, the notion that an expression evaluates to a result can be understood by the reader even without a complete specification of all operators and built-in functions that might be used in the expression; - specifying the valid syntax of Rexx programs without using Rexx coding. The method used, Backus Normal Form, can adequately be introduced by prose. Ultimately, some understanding of programming languages is assumed in the reader (just as the ability to read prose is assumed) but any remaining circularity in this standard is harmless. The comparison of two single characters is an example of such a circularity; Config_Compare can compare two characters but the outcome can only be tested by comparing characters. It has to be assumed that the reader understands such a comparison. Some of the definition using code is repeating earlier definition in prose. This duplication is to make the document easier to understand when read from front to back. Note that the layout of the code, in the choices of instructions-per-line, indentations etc., is not significant. (The layout style used follows the examples in the base reference and it is deliberate that the DO and END of a group are not at the same alignment.) The code is not intended as an example of good programming practice or style. The variables in this code cannot be directly referenced by any program, even if the spelling of some VAR_SYMBOL coincides. These variables, referred to as state variables, are referenced throughout this document; they are not affected by any execution activity involving scopes. Some of more significant variables and routines are written with # as their first character. The following list of them is intended as an aid to understanding the code. The index of this standard shows the main usage, but not all usage, of these names. The following are constants set by the configuration, by Config_-Constants: #Configuration is used for PARSE SOURCE. #Version is used for PARSE VERSION. #Bif_Digits. represents numeric digits settings, tails are built-in function names. #Limit_Digits is the maximum significant digits. #Limit_-EnvironmentName is a maximum length.

#Limit_ExponentDigits is the maximum digits in an exponent. #Limit_Literal is a maximum length.

#Limit_MessageInsert is a maximum length.

#Limit_Name is a maximum length.

#Limit_String is a maximum length.

#Limit_TraceData is a maximum length.

These are named outputs of configuration routines:

#Response is used to hold the result from a configuration routine. #Indicator is used to hold the leftmost character of Response. #Outcome is the main outcome of a configuration routine.

#RC is set by Contig_Command.

#NoSource is set by Config_NoSource.

#Time is set by Config_Time

#Adjust<Index “#Adjust” #“ ” > is set by Config_Time

These variables are set up with output from configuration routines:

#HowInvoked records from API_Start, for use by PARSE SOURCE.

#Source records from API_ Start for use by PARSE SOURCE.

#AIIBlanks<Index “#AllBlanks” # “ ” > is a string including Blank and equivalents. #ErrorText.MsgNumber is the text as altered by limits.

#SourceLine. is a record of the source, retained unless NoSource is set. #SourceLine.0 is a count of lines.

#Pool is a reference to the current variable pool.

These are variables not initialized from the configuration:

#Level is a count of invocation depth, starting at one.

#NewLevel equals #Level plus one.

#Pool1 is a reference to the variable pool current when the first instruction was executed.

#Upper is a reference to the variable pool which will be current when the current PROCEDURE ends. #Loop is a count of loop nesting.

#LineNumber is the line number of the current clause.

#Symbol is a symbol after tails replacement.

#API_Enabled determines when the application programming interface for variable pools is available. #Test is the Greater/Lesser/Equal result.

#InhibitPauses is a numeric trace control.

#InhibitTrace is a numeric trace control.

#AtPause is on when executing interactive input.

#AllowProcedure provides a check for the label needed before a procedure.

#DatatypeResult is a by-product of DATATYPE().

#Condition is a condition, eg ‘SYNTAX’.

#Trace_QueryPrior detects an external request for tracing.

#TraceInstruction detects TRACE as interactive input.

These are variables that are per-Level, that is, have #Level as a tail component:

#IsFunction. indicates a function call.

#IsProcedure. indicates indicates the routine is a procedure.

#Condition. indicates whether the routine is handling a condition.

#ArgExists.#Level.ArgNumber indicates whether an argument exists. (Initialized from API_Start for Level=1)

#Arg.#Level.ArgNumber provides the value of an argument. (Initialized from API_Start for Level=1) When ArgNumber=0 this gives a count of the arguments.

#Tracing. is the trace setting letter.

#Interactive. indicates when tracing is interactive. ('?' trace setting) #ClauseLocal. ensures that DATE/TIME are consistent across a clause. #ClauseTime. is the TIME/DATE frozen for the clause.

#StartTime. is for 'Elapsed' time calculations.

#Digits. is the current numeric digits.

#Form. is the current numeric form.

#Fuzz. is the current numeric fuzz.

These are qualified by #Condition as well as #Level:

#Enabling. is 'ON', 'OFF' or 'DELAYED".

#Instruction. is 'CALL' or 'SIGNAL'

#TrapName. is the label.

#ConditionDescription. is for CONDITION('D')

#ConditionExtra. is for CONDITION('E')

#ConditionInstruction. is for CONDITION('T')

#PendingNow. indicates a DELAYED condition. #PendingDescription. is the description of a DELAYED condition. #PendingExtra. is the extra description fora DELAYED condition. #EventLevel. is the #Level at which an event was DELAYED.

These are qualified by ACTIVE, ALTERNATE, or TRANSIENT as well as #Level:

#Env_Name. is the environment name.

#Env_Type. is the type of a resource, and is additionally qualified by input/output/error distinction. #Env_Resource. is the name of a resource, and is additionally qualified by input/output/error distinction. #Env_Position. is INPUT or APPEND or REPLACE, and is additionally qualified by input/output/error distinction.

These are variables that are per-loop:

#Identity. is the control variable.

#Repeat. is the repetition count.

#By. is the increment.

#To. is the limit.

#For. is that count.

#Iterate. holds a position in code describing DO instruction semantics. #Once. holds a position in code describing DO instruction semantics.

#Leave. holds a position in code describing DO instruction semantics.

These are variables that are per-stream:

#Charin_Position.

#Charout_Position.

#Linein_Position.

#Lineout_Position.

#StreamState. records ERROR state for return by STREAM built-in function.

These are commonly used prefixes: Config_ is used for a function provided by the configuration. API_is used for an application programming interface.

Trap_ is used for a routine called from the processor, not provided by it. Var_ is used for the routines operating on the variable pools.

These are notation routines, only available to code in this standard:

#Contains checks whether some construct is in the source. #Instance returns the content of some construct in the source. #Evaluate returns the value of some construct in the source. #Execute causes execution of some construct in the source. #Parses checks whether a string matches some construct. #Clause notes some position in the code.

#Goto continues execution at some noted position.

#Retry causes execution to continue at a previous clause.

These are frequently used routines: #Raise is a routine for condition raising. #Trace is a routine for trace output.

#TraceSource is a routine to trace the source program. #CheckArgs processes the arguments to a built-in function.

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