CREXX
Language Reference

The CREXX team

September 14, 2026

THE REXX LANGUAGE ASSOCIATION
CREXX Programming Series
ISBN 978-94-039116-3-2

Publication Data

©Copyright The Rexx Language Association, 2020 - 2026

All original material in this publication is published under the Creative Commons - Share Alike 3.0 License as stated at `http://creativecommons.org/licenses/by-nc-sa/3.0/us/legalcode`.

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.

This edition is registered under ISBN 978-94-039116-3-2

ISBN 978-94-039116-3-2

Content is up to date with version *crexx-1.0.0-beta.3+local.g4c4f4c48643e*

Contents

The CREXX Publication Series

This book is part of a library, the *CREXX Programming Series*, documenting the CREXX programming language and its use and applications. This section lists the other publications in this series, and their roles. These books can be ordered in convenient hardcopy and electronic formats from the Rexx Language Association.

Programming Guide	The Programming Guide explains the working of the tools and has examples for building programs on the platforms it supports.
Language Reference	Referred to as the CRL, this is meant as the formal definition for the language, documenting its syntax and semantics, and prescribing minimal functionality for language implementers.
Library Reference	A complete reference of all included functionality in the CREXX distribution.
VM Specification	The CREXX VM Specification documents the CREXX Assembly Language and its execution by the CREXX Virtual Machine. It also contains low level virtual machine and ABI specifications.

Typographical conventions

In general, the following conventions have been observed in the CREXX publications:

- Body text is in this font
- Examples of language statements are in a `keyword` or **bold** type
- Items that are introduced, or emphasized, are in an *italic* type
- Included program fragments are listed in this fashion:

```
/* salute the reader */
say 'hello reader'
```

Console output generated from programs contained in the text is shown in this form:

```
hello reader
```

1 About This Book

This language reference documents the implemented CREXX language surface for the Release 1 beta line.

The main release language is Level B: a typed Rexx-family systems language compiled by `rxc`, assembled by `rxas`, and executed by the `rxvm` runtime. Level B is not a complete Classic REXX compatibility mode, and this reference does not describe planned future levels as current behaviour.

1.1 Audience

This reference is for programmers who need precise syntax and behaviour:

- application and library authors writing Level B source
- contributors changing compiler, library, or VM behaviour
- users inspecting generated RXAS or bytecode
- maintainers checking whether documentation matches the implementation

New users should start with the README, the documentation map, and the programming guide. This book is the detailed reference.

1.2 Document Structure

This document is in three parts:

Objectives The Objectives part explains what CREXX is intended to achieve and defines the scope of the current release. It describes the principles that guide the language design, the intended uses of the language, and the balance between compatibility with traditional REXX and the introduction of new facilities. It also identifies which language levels and features are included in this release, which are still under development, and which are deliberately outside its scope.

Overview The Overview introduces the language as a whole before its individual features are described in detail. It explains the CREXX language model, the organization of source programs into modules, and the mechanisms used to call routines, functions, and methods. It also describes compiler and language options, the structure of a program, and the relationships between declarations, executable code, imported modules, and external libraries. This part provides the conceptual framework needed to understand the more formal reference material.

Reference The Reference gives the detailed definition of the language. It specifies the syntax and behaviour of types, literal values, variables, expressions, and operators. It describes classes and objects, namespaces and name resolution, numeric settings and arithmetic, and all executable and declarative statements. It also documents the standard libraries and their interfaces. Where relevant, the reference states the applicable language level, default behaviour, restrictions, error conditions, and interactions with other language features.

PART I

Objectives

2

Introduction

CREXX is a Rexx[1] -family language and toolchain that compiles source programs to an open bytecode format, then executes that bytecode on the CREXX virtual machine or packages it into a native executable.

The Release 1 beta line is centred on Level B. Level B is the implemented systems language used by the project itself: it is statically typed, module based, and close enough to REXX to keep the language readable while giving the compiler and VM explicit type, module, and contract information.

The public documentation should describe what the current toolchain can do. Ideas for future levels remain part of the project direction, but they are not release facts. In this book, unless a section says otherwise, syntax and behaviour refer to the current Level B implementation.

2.1 The origins of CREXX

REXX did not derive from a single earlier language. Its immediate predecessor was IBM's EXEC 2, from which it inherited the idea of a general command and application-extension language. Its structured syntax and many of its programming constructs belong principally to the PL/I and Algol tradition, with Pascal forming part of the same background. Cowlishaw also acknowledged influences from APL and from several unpublished experimental languages that he had de-

[1] M. F. COWLISHAW 1985

veloped during the 1970s. These influences were not adopted mechanically: existing ideas were repeatedly simplified or reformulated according to Rexx's central objective of making programs easy to write and read.

In "The Design of the REXX Language[2] ," Cowlishaw discusses BASIC and Logo as languages intended for general or non-specialist users. He places Rexx in the same broad "personal language" category, while arguing that Rexx was intended to support both casual and professional programming.

crexx is part of this tradition and apart from the mentioned languages, several Rexx variants as brexx, Object Rexx and NetRexx influenced its design. With native (hardware) types, compilation to native executables, ambitious performance goals, Unicode support and libraries for contemporary IT, it explicitly aims at the professional programmer, while keeping the simplicity and easy integration with platforms and applications its predecessors are known for.

2.2 What The Release 1 Beta Line Contains

The implemented Level B surface includes:

- source modules compiled from `.crexx` to `.rxas` assembly and then to `.rxbin` bytecode
- namespaces, imports, exposed procedures, global values, and class/interface metadata
- static scalar types, arrays, and object values
- procedures, functions, methods, factories, and varargs
- structured control flow including `if`, `do`, `select`, `leave`, `iterate`, and `return`
- `parse`, `address`, `signal`, and `trace` through the current compiler-exit and VM runtime support
- Level B classes and interfaces, including default/final interface methods, factories, checked casts, type tests, and concrete type introspection
- a standard library built largely in crexx itself, with native support where direct VM or platform access is required

The main tools are:

[2] M. Cowlishaw 1987

- `rxc`, the compiler from `.rexx` source to `.rxas` assembly
- `rxas`, the assembler from `.rxas` to `.rxbin`
- `rxlink`, the linker for combining modules into a shared-pool linked image
- `rxvm` and related interpreters for running `rxbin` bytecode
- `crexx`, the convenience driver that compiles, links or packages, and runs common programs

2.3 Design Position

CREXX keeps the REXX emphasis on readability and directness, but Level B is not Classic Rexx. It is the typed foundation used to build libraries, tools, and later language layers. That is why Level B source normally starts with:

```
options levelb
```

and why reusable libraries usually declare a namespace:

```
namespace rxfnsb expose abs
```

Headerless top-level scripts run through the `crexx` driver are treated as Level B scripts with `rxfnsb` imported automatically. Library and compiler work should still use explicit `options`, `namespace`, and `import` declarations so the source remains clear to both readers and tools.

2.4 Reading This Reference

The reference is intentionally factual rather than aspirational:

- Syntax sections describe accepted source forms.
- Tool sections describe the current command-line tools and generated files.
- VM sections describe the current bytecode and runtime model at the level useful to RXAS authors and embedding work.
- Experimental areas are labelled as experimental when the implementation is intentionally small or still being shaped.

3

Language Levels

CREXX uses language levels to name related Rexx-family language surfaces without pretending they are all implemented by the same compiler. The letters are a map, not a compatibility promise.

For the Release 1 beta line, Level B is the supported CREXX language documented by this reference. Other levels are either historical context, project direction, or editor/tooling targets unless their documentation explicitly says that a compiler feature is implemented and tested.

3.1 Summary

- Level B will be used for current CREXX programs
- Level C represents Classic REXX compatibility work with real DSLSH syntax highlighting / parser-mode progress, but not yet a release compiler language
- Levels E and N are planned DSLSH syntax-highlighting targets for Object REXX and NetRexx source, not as languages that CREXX intends to compile or run
- Level G has an implemented initial task/parallel surface and libraries; Levels D and L remain directions for future language work

3.2 Level B: Current cRexx

Level B is the foundation language for the current compiler, standard library, tooling, and examples. It is deliberately more explicit than Classic Rexx:

- source files normally begin with `options levelb`
- values have known types such as `.int`, `.boolean`, `.float`, `.decimal`, `.string`, `.binary`, `.object`, and `.void`
- arrays are declared from typed values, for example `.string[]`
- reusable code is grouped through `namespace`, `import`, and `expose`
- procedures, methods, factories, and interfaces have checked signatures
- modules compile to `.rxbin` bytecode and can be linked into deployable images

Level B is used to implement much of the standard library. That is an intentional part of the architecture: the same language available to users is also used to build the platform.

3.3 Level Catalogue

The following names are used by the project, but their status is deliberately different:

Level	Meaning	Current project status
Level A	Early compact Rexx proof-of-concept foundation.	Historical only. Not a release target.
Level B	Current typed crexx foundation language.	Implemented and documented as the Release 1 beta user language.
Level C	Classic Rexx compatibility.	Not a release compiler language yet. A dedicated DSLSH syntax-highlighting / parser-mode front end now exists as the first concrete compatibility slice.
Level D	A cRexx-compatible extension direction above Classic Rexx.	Direction only. Not a release language yet.

Level	Meaning	Current project status
Level E	Object Rexx / ooRexx relationship point.	Planned only as a DSLSH syntax-highlighting target. crexx does not plan to compile or run ooRexx as Level E.
Level G	General-purpose modern crexx built on Level B.	Task declarations, ordinary-call task expressions, `DO PARALLEL`, concurrency classes, concurrent HTTP, mathematics, and explicit Unicode text services are implemented behind `OPTIONS LEVELG`; Level G is not the stable baseline language for this release.
Level L	Language-engineering crexx direction for parser, grammar, AST, and symbol-table work.	Directional, with an initial `rxfnsl` generated-output proving demo; not a release language yet.
Level N	NetRexx relationship point: Rexx-family syntax with Java/JVM integration.	Planned only as a DSLSH syntax-highlighting target. crexx does not plan to compile or run NetRexx as Level N.

3.4 DSLSH And Compatibility

DSLSH syntax-highlighting support is allowed to be wider than the compiler. It can help users edit and diagnose Rexx-family source without claiming that crexx can execute that source.

Level C is the first example of that split. The project has Level C parser-mode and syntax-highlighting work for Classic Rexx source, and normal `rxc` compilation now lowers a proven subset of Classic Rexx into the canonical Level B runtime path. Classic constructs outside that implemented slice still fail closed with a Level C unsupported-shape diagnostic. This lets the project build useful editor support, standard-diagnostic experience, and incremental runtime coverage without claiming full Classic Rexx compatibility before the remaining lowering / runtime work is complete.

Level C owns the Classic Rexx byte-text compatibility decision. Direct Level C BIF calls carry a `RexxClassicConfig` through `RexxBifCallContext`: `BYTE` is the default compatibility profile and `UTF8` is the opt-in text profile. This library/runtime configuration does not require a compiler or lowering change. Per-value `RexxValue`

flags describe text/binary validity but do not select the profile. See Unicode for the exact contracts.

Level G already owns the initial structured-concurrency surface above Level B. Its syntax is enabled only by `OPTIONS LEVELG` and lowers through the public Level B concurrency classes. See Concurrent programming for checked examples and Tasks and parallel execution for the formal source rules.

Level G also owns explicit Unicode services above the Level B codepoint contract. `rxunicode` provides Unicode 17.0.0 normalization and predicates, full default case mapping, case folding, default extended grapheme clusters, and strict-by-default typed codecs between `.string` and `.binary`. The implementation uses private Level B executors and generated immutable data; normalization certificates are VM-carried in a protected language-owned register-flag sub-band and are invalidated by content mutation. Importing the module does not change ordinary equality, codepoint indexing, or text I/O. See Unicode and Unicode text services.

Level L currently uses the Level B-derived compiler pipeline plus `options levell` for library-shaped experiments. Its first concrete library, `rxfnsl`, is a generated-output proof rather than a generator: it shows what a future lexer/parser generator might emit using binary constants, packed token records, and direct RXAS binary-memory operations. That target shape should be proved by examples before deciding whether to port a generator such as re2c or adapt a generator backend to emit cRexx/RXAS directly.

Levels E and N should be understood in that same tooling sense. They reserve clear names for Object Rexx and NetRexx editor support, but they are not crexx runtime or source-compatibility commitments.

3.5 Defaults

The compiler can be given a default level with `rxc --level levelb`. The source file wins if it contains an explicit `options` instruction.

The `crexx` driver gives headerless top-level scripts a practical default by compiling them as Level B and importing `rxfnsb`. This convenience does not change the recommendation for reusable source: write the `options` and `import` lines explicitly.

3.6 Compatibility

Level B borrows from Rexx, but it is not a drop-in Classic Rexx interpreter. Programs that rely on Classic Rexx's untyped variable model, late binding, or default built-in function visibility usually need small, explicit Level B changes.

The compatibility goals for future levels are important project direction, but this Release 1 beta documentation should not describe future compatibility as current behaviour.

3.7 Extensibility

A small language is easy to learn and easy to use. When the language is too small, and there is no large, well organised runtime library, a lot of function needs to be provided by the user. Classic Rexx provides extensibility through addressing environments and function packages. These environments need to be addressed each in their own way, the function packages need to be registered and checked; the searching and linkage conventions differ per platform and operating system.

crexx adds an easily extensible module system which integrates extended runtime functionality which behaves in the same manner over library extensions written in Rexx or native functions written in C and in other languages. These work in tandem with the compiler exit facility.

4

Unicode

CREXX separates text semantics by language level. This avoids making Classic byte-oriented programs and modern Unicode programs silently share incompatible rules.

Unicode property tables that claim a version are pinned to Unicode 17.0.0. Updating that version requires updating the tables, tests, and this document together. Level B's deliberately limited case table is not a claim to implement the complete Unicode 17 case algorithm.

4.1 Level B

Level B `.string` values are valid UTF-8. Their public character operations use Unicode scalar/codepoint positions and lengths, never UTF-8 byte offsets. `.binary` is the distinct type for arbitrary bytes.

Level B intentionally supplies a limited Unicode foundation rather than the full Unicode algorithm suite:

- string length, indexing, slicing, searching, and reversal are codepoint based;
- default word blanks use the Unicode `White_Space` property;
- case conversion uses the runtime's locale-independent simple mapping;
- operations do not normalize text implicitly;
- canonical equivalence and locale-sensitive comparison are not inferred.

Text I/O into a Level B `.string` validates UTF-8. Invalid byte sequences signal `UNICODE_ERROR`; they are not repaired or reinterpreted. `LINEIN` returns text lines and `CHARIN` counts codepoints. Text output writes the string's UTF-8 encoding. Arbitrary file or protocol bytes belong in `.binary` and byte-oriented I/O APIs rather than being smuggled through `.string`.

The VM whitespace table and the Level C UTF8 profile are pinned to the same Unicode version. U+180E is not whitespace in that version.

4.2 Level G

Level G owns general-purpose Unicode behavior above the Level B codepoint foundation. The `rxunicode` namespace provides explicit Unicode 17.0.0 normalization, full default case mapping, case folding, default extended grapheme-cluster operations, and typed byte/text codecs. It is imported explicitly:

```
options levelg
import rxunicode
```

The normalization family is `toNFD`, `toNFC`, `toNFKD`, and `toNFKC` plus the matching `isNFD`, `isNFC`, `isNFKD`, and `isNFKC` predicates. Normalization is never implicit. Compatibility forms may remove distinctions and must be selected deliberately. There is no public normalizer object.

`toUppercase` and `toLowercase` apply locale-neutral Unicode full default case mappings. They may expand and do not normalize. They are distinct from the existing Level B simple `upper`/`lower` contract. Titlecase and locale-tailored case mapping are not part of the baseline.

The case-fold procedures are:

- `toCasefold(text)` for default full folding;
- `toSimpleCasefold(text)` for default simple folding;
- `toTurkicCasefold(text)` for full folding with Turkic I mappings; and
- `toTurkicSimpleCasefold(text)` for simple folding with Turkic I mappings.

The shortest name is Unicode Default Case Folding and is compatible with the useful TUTOR vocabulary. Full folding may expand a string, while simple folding does not. These direct procedures are the complete public case-fold surface; there

is no reusable case-folder object because folding retains no useful state between calls.

Case folding is for caseless matching. It is not locale-sensitive case conversion, does not preserve or apply normalization, and does not retain a source-to-result index map. The Turkic forms are explicit operations rather than process or task locale state. `.binary` is not accepted.

The direct grapheme procedures are `graphemeCount`, `graphemeSubstr`, `graphemePos`, and `graphemeReverse`. They implement the default extended grapheme-cluster profile `UAX29-C1-1` without tailoring. The immutable `.graphemes` class prepares one boundary index for repeated access, slicing, searching, reversal, and forward iteration. Grapheme operations neither normalize nor case-fold their input.

`encode(.string[, encoding[, replacement]])` returns `.binary` and `decode(.binary[, encoding[, replacement]])` returns `.string`. UTF-8 is the default. The baseline also supports explicit-endian UTF-16/UTF-32, US-ASCII, ISO-8859-1, Windows-1252, IBM437, IBM850, and IBM1047. Conversion is strict by default; an explicitly supplied typed third argument opts into replacement. Codecs do not add or consume a BOM as metadata. `isDecodable` checks a complete byte value and `isEncodingSupported` checks a name. Whole-file `readbinary` and `writebinary` operations let callers compose binary I/O with these codecs.

These algorithms are explicit services. Level B does not silently acquire grapheme, normalization, case-folding, or encoded-stream semantics when `rxunicode` is imported. Ordinary string comparison and indexing remain exact and codepoint based.

See the Unicode text services chapter for the complete API, codec/file examples, error boundaries, and TUTOR migration guide.

4.3 Level C

Classic compatibility selects a character profile through the `RexxBifCallContext` configuration object. The profile is call/runtime state; it is not selected by flags on an individual `RexxValue`.

Two profiles are defined:

Profile	Meaning
BYTE	Default Classic compatibility profile. Character units are exact bytes. Arbitrary binary values remain valid inputs, PAD means one byte, positions are byte positions, and the configured range is 00 through FF.
UTF8	Opt-in text profile. Inputs used as text must be valid UTF-8; character positions are Unicode codepoints, PAD means one codepoint, and default blanks use Unicode 17.0.0 White_Space.

Both profiles can add configured blank characters. BYTE additions are bytes; UTF8 additions are codepoints. With no additions, the UTF8 word scanner uses the VM fast path.

RexxValue text/binary flags only describe which representations are current and whether held bytes are valid UTF-8. They never change the active profile. There is no implicit fallback from UTF8 to BYTE and no implicit normalization.

The direct Level C conversion BIFs preserve exact encoded bytes. C2X and C2D read those bytes; X2C and D2C produce them. In the UTF8 profile, a produced value is marked as text only when its exact bytes are canonical valid UTF-8; otherwise it remains binary-authoritative. BYTE XRANGE provides the inclusive, wrapping 256-byte configured range. UTF8 XRANGE is intentionally unavailable because it is not a Unicode range API; Level B sequence is the non-wrapping Unicode-codepoint operation.

5 Performance

For precise technical documentation, we refer to the *cRexx VM Specification* publication, which contains the last word on the architecture and implementation of the bytecode compiler, the assembler and the first two virtual machines. One of these is a conventional byte code interpreter, the other is a *threaded code interpreter*. These share over 99% of their source code.

The virtual machine executing the `rxas` assembler instructions is written according to a strict set of rules that limits the possibility of pipeline stalls in modern processor architectures.

5.1 Compiler inlining

The `rxc` compiler performs conservative inlining as an optimisation pass. When the compiler can prove that replacing a call with the body of the called procedure, method, or factory preserves the same behaviour, it may do so before emitting `rxas` assembly. If that proof is not available, the call is left as an ordinary call. This means inlining should be treated as an implementation optimisation, not as a source-language guarantee.

The implementation is deliberately conservative because CREXX inlining is not simple text substitution. The compiler rewrites the already-validated abstract syntax tree and then continues through the normal compiler pipeline. That tree surgery must preserve:

- argument binding, including defaults, by-value copies, and exposed/reference arguments
- return values, including object, binary, and array-shaped values
- nested scopes, local variables, and source/debug locations
- method receiver state and copyback for mutating methods
- factory setup and object initialization
- multi-level imports, where source, rxas, rxdas, binary metadata, and linked libraries must all describe the same callable contracts

Some cases are intentionally left as calls. Examples include procedure-level `expose`, dynamic-index varargs, assembler code that aliases registers, receiver expressions that need complex copyback, interface/member dispatch that cannot be proved to target one concrete implementation, and factories or methods whose class layout cannot yet be proved safely. These exclusions are not failures; they are how the optimiser avoids changing program semantics.

For the initial public release the compiler uses a 300 AST-node body cutoff for inline candidates. This is a profitability and metadata-size limit rather than a language rule. Larger bodies, or bodies that cross one of the semantic gates above, remain normal calls.

Use `rxc -n` to disable compiler optimisation, including inlining, when comparing generated assembly or investigating optimiser behaviour.

5.2 SELECT and equality dispatch

The compiler can replace suitable static SELECT or equality ladders with a packed jump table. This is an implementation optimization: SELECT remains first-match-wins, its selector is evaluated at the source-defined point, and a table miss continues through the original fallback path.

Release 1 uses measured, type-specific minimum run lengths:

Dispatch kind	Minimum consecutive constant cases
Integer	8
Exact string	3
Blank-padded nonnumeric string	2
Numeric string	2

Dispatch kind	Minimum consecutive constant cases
Exact binary	3

Integer tables have a fixed lookup cost, so short ladders and first-case-heavy traffic often favor predicted branches. String tables avoid repeated text or numeric comparisons and become worthwhile earlier. These thresholds choose between a branch ladder and a table; the assembler independently chooses the table's `linear`, `openhash`, or `acph` representation.

Explicit C-style SELECT may be lowered in optimized and non-optimized builds when its selector and constant cases are safe. General IF and classic SELECT recognition is optimized-build-only and requires repeated reads of the same resolved scalar variable with no alias, reference, call, getter, mutation, or other observable evaluation risk. Unsupported, dynamic, duplicate, or short runs remain ordinary comparisons. Consecutive eligible runs may be tabled without moving an intervening expression.

For a known hotspot, inspect generated RXAS for `.jtable` and the relevant `jump*` instruction. Hand-written RXAS can also select an explicit table algorithm. Do not rewrite clear source solely to force this optimization; thresholds may be retuned as VM and platform measurements improve.

5.3 Native executables

Level B can produce native[3] executables which can be distributed and run on machines that do not have any CREXX infrastructure installed. These contain a compiled version of the programs linked into it, as well as their own version of the CREXX virtual machine; the startup of these programs is very fast because the compilation, assembly and linkedit steps are skipped.

[3] native to combination of the instruction set architecture of the machine and its operating system calling conventions.

PART II

Overview

6 Beginnings

6.1 Greeting the world

Rexx is designed as a small language, but there are already a lot of things you can do without involving anything outside of it. One of those is to greet the world, as is the usual and obligatory first example of any programming language book:

```
options levelb
/* rexx: welcoming the world */
say 'hello world!'
```

```
hello world!
```

There is no need to include the starting comment, but by all means include it if you want to.[4]

`say` is a statement of the Rexx language and as such built-in. It does not matter if it is expressed as `say`, `Say`, or `"SAY""`; in other words, the statements of the language are case-insensitive.

With the roots of the Rexx language (when it still went with one less 'x' at the end) as a command language, there is one other important thing we can do without any outside support, and that is sending commands to the environment.

```
options levelb
```

[4] Level B crexx has a few `options` like `comments_hash`, `comments_dash` or `comments_slash` to tune the comment format but the starting comment is meant for some operating systems to distinguish Rexx from the other, earlier command processors they have, like exec/exec2 for VM, CLIST for TSO on z/OS and the .bat language for DOS and OS/2.

```
/* rexx: welcoming the world */
say 'hello world!'
address system 'date'
```

```
hello world!
ma 14 sep. 2026 17:11:23 CEST
```

As `crexx` is the standard environment before we change it (by specifying something else on the `address` statement), this is sent to this environment, which executes it. The `crexx` environment enables issueing commands in a system independent manner, including the commands which are shell builtins, which can be tricky to address in programs that need to be system independent (see The Address Statement on page 151 in the Reference section for all available commands. The `address system` command adresses the shell under which your program executes. If this has a `date` command, the output of it will be returned. The I/O to the environment using the `address` statement can also be redirected using *stem variables*. In fact, sending commands to an environment is one of the examples of Object Orientation using Classic Rexx; the same message can be sent to different objects, as long as these are implemented in the environment, and will potentially yield different answers.

7 Program Control

If we go further than scripts that are straightforward lists of clauses and external commands, we need keywords to manipulate the flow of control in a program. These are shown in this chapter.

7.1 Changing the flow of a program

A program can be a single list of instructions, or a number of lists of instructions that determine which list to process and how many times to process it.

Instructions that direct the processing of the program are called *control instructions*. They do the following things:

Branching Selecting one of several lists of instructions to process. The branching instructions are `IF` and `SELECT`.

Looping Repeating a list of instructions, either for a speficied number of times or as long as some condition is satisfied. The `LOOP` or `DO` instruction (when used with keywords like `UNTIL` and `WHILE`) do the looping in Rexx.

Exiting A program that is a single list ends when it reaches the last instruction. To explicity end a program, use the instructions `EXIT` and `RETURN`.

7.2 Grouping instructions

What most control instructions have in common is that they often use groups of clauses that act as a single clause. The simplest way to group clauses is with the keyword DO. For example:

```
DO
  clause1
  clause2
  clause3
  ...
END
```

If the keyword DO is a clause by itself, the list of clauses that follows (up to the END keyword) is processed once (no loop is implied). This form of the DO instruction and the END keyword associated with it tell REXX to treat the enclosed instructions as a single instruction.

7.3 Testing Conditions

In a comparison, two terms are joined by operators as = (equal to), > (greater than), or < (less than) in order to pose a test of the terms. The expression itself evaluates as 1 if the comparison is *true* or 0 if the comparison is *false*. For example:

```
options levelb
/* some comparisons */
say 5 = 5           /* displays '1' - true  */
say 5 < 4           /* displays '0' - false */
say 5 = 4           /* displays '0' - false */
say 5 > 4           /* displays '1' - true  */

reply = "YES"       /* assigns the string "YES"
                       to the variable REPLY */

say reply           /* displays "YES"       */
say reply = "MAYBE" /* displays '0' - false */
say reply = "YES"   /* displays '1' - true  */
```

```
1
0
0
1
```

```
YES
0
1
```

7.4 Simple Branching

To tell Rexx how to make a decision about a single instruction, use:

```
IF expression
THEN instruction
```

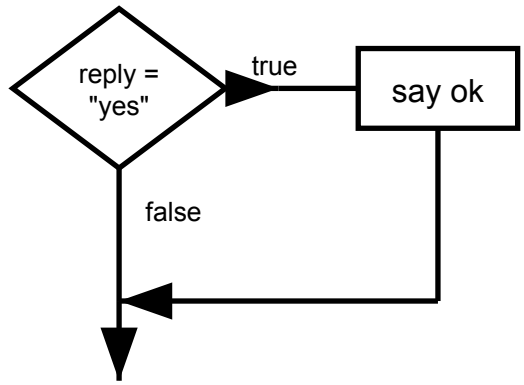
Rexx processes `instruction` only if `expression` is true.

```
options levelb
say "Type YES to continue"
reply = "YES"
if reply = "YES" then say "OK!"
/* program continues from here ...*/
```

```
Type YES to continue
OK!
```

The instruction `say "OK!"` is only executed if the variable `reply` has the value `YES`.

The IF instruction introduces a new branch of instructions to process when the `IF` expression is true. Programmers often visualize the action of a decision-making instruction by using a diagram like this, called a flowchart.



The decision-making expression is represented by a diamond. If the expression (here REPLY="YES") evaluates as true, then the program branches, or takes a detour, through one additional instruction before resuming on the next line.

7.5 Using DO..END for multiple clauses

To put a list of instructions after the THEN, use the DO instruction and the END keyword. That turns the whole group into a single instruction. For example:

```
IF expression THEN
DO
instruction1
instruction2
instruction3
< ... and so on>
END
```

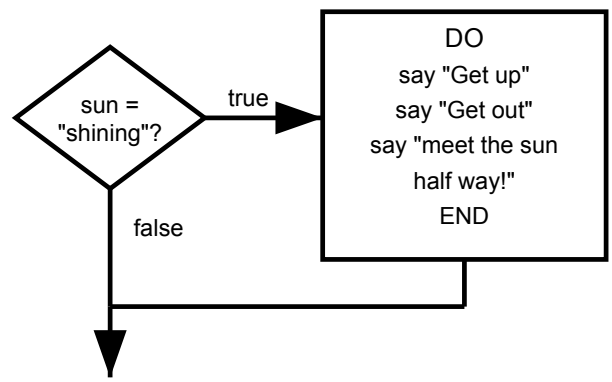
With the DO and END keywords bracketing the list, Rexx knows to treat the listed instructions as a unit to:

- Process all of them if `expression` is true
- Ignore them all if `expression` is false.

```
options levelb
/* Wake-up call */
sun = 'shining'
if sun = "shining"
then
  do
    say "Get up ! "
    say "Get out ! "
    say "Meet the sun half way! "
  end
```

```
Get up !
Get out !
Meet the sun half way!
```

The flowchart diagram would look like this:



In the previous example, if `sun = "shining"` evaluates as 1 (*true*), then all three `SAY` instructions are processed. But if `sun = "shining"` evaluates as 0 (*false*), then none of the `SAY` instructions are processed.

The THEN and DO keywords are each on a separate line. This is optional. You could also write the program as:

```
options levelb
/* Wake-up call */
sun = 'shining'
if sun = "shining" then
do
  say "Get up ! "
  say "Get out ! "
  say "Meet the sun half way! "
end
```

or

```
options levelb
/* Wake-up call */
sun = 'shining'
if sun = "shining" then do
  say "Get up ! "
  say "Get out ! "
  say "Meet the sun half way! "
end
```

7.6 Two paths: ELSE

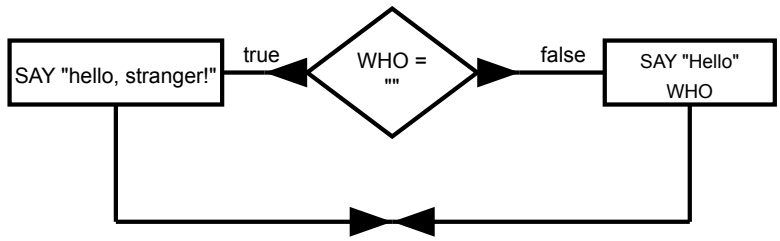
Used alone, IF … THEN adds a branch of instructions to process when the controlling expression is true. You can also add a second branch of instructions to process when the expression is false. The keyword ELSE introduces this alternate list. For example:

```
IF expression
THEN instruction1
ELSE instruction2
```

When IF is used this way, REXX processes only one of these instructions, not the other. It will process:

- Instruction l only if `expression` is true
- Instruction 2 only if `expression` is false.

The flowchart diagram would look like this:



7.7 The SELECT instruction

You are not limited to two choices. You can use the `SELECT` instruction to have a Rexx program select one of any number of branches. For example:

```
SELECT
WHEN expression1 THEN instruction1
WHEN expression2 THEN instruction2
WHEN expression3 THEN instruction3
OTHERWISE
   instruction
   instruction
   instruction
END
```

- If `expression1` is true, `instruction1` is processed. After this, processing continues with the instruction following the `END`.
- If `expression1` is false, then `expression2` is tested. If it is true, then instruction2 is processed and processing continues with the instruction following the `END`.
- If `expression1`, `expression2`, and so on, are all *false*, then processing continues with the instruction following the `OTHERWISE`.

`OTHERWISE` is essentially the `SELECT`-equivalent of `ELSE`. If no `WHEN` expressions evaluate to *true* and there is no `OTHERWISE` clause, the `SELECT` instruction simply acts as a `NOP` (null operation) and does nothing.

7.8 Looping with DO

The `DO` instruction also controls iteration. There are three common forms:

- Simple `DO … END` (grouping only; runs once)
- Counted `DO` (with a control variable and optional `TO`/`BY`/`FOR`)
- Conditional `DO WHILE` / `DO UNTIL`

Like in other Rexx variants the keyword `loop` can be used instead of `do` for the purpose of iteration; this will emphasize a loop to distinguish it from the other uses of `do`.

7.8.1 Simple DO (grouping only)

```
options levelb
if cond then do
  say "A"
  say "B"
end
```

multiple instructions into one block.

7.8.2 Counted DO

```
options levelb
sum = .int
/* i counts from 1 to 5 by 1 */
do i = 1 to 5
  sum = sum + i
end
say sum  /* 15 */

DO i = expr_start [ TO expr_stop ] [ BY expr_step ] [ FOR expr_count ]
  ...
END
```

inclusive upper bound (when present). - `BY` sets the step (defaults to 1 if omitted; negative steps count down). - `FOR` limits the number of iterations (optional), stopping earlier even if `TO` would allow more.

7.8.3 Conditional DO

```
options levelb
/* while-loop */
```

```
do while i < 10
  i = i + 1
end

/* until-loop */
do until ready
  call work
end
```

executes the body as long as `e` is true. - `DO UNTIL e` executes until `e` becomes true (i.e., repeats while `\e`).

7.8.4 LEAVE and ITERATE

- `ITERATE` restarts the current loop at the next iteration (optionally naming a control variable to target an outer loop).
- `LEAVE` exits the current loop immediately (optionally naming a control variable to target an outer loop).

7.8.5 Block Expressions

`DO ... END` can also appear inside an expression. In that form it behaves like a local statement block that yields a value back to the parent expression.

```
options levelb
total = 10 + do
  a = 5
  if a > 0 then leave with a * 2
  leave with 0
end
say total   /* 20 */
```

Rules:

- A block expression introduces a local block scope, just like a grouped `DO`.
- The block result is produced with `LEAVE WITH expression`.
- Every reachable value-producing path should end in `LEAVE WITH ...`.
- If more than one `LEAVE WITH` is present, all yielded values must have the same type.
- Plain `LEAVE` keeps its loop meaning. Inside an expression block, use `LEAVE WITH` when you intend to return a value.

Because block expressions are ordinary expressions, they can be nested and used anywhere a primary expression is valid, such as arithmetic, comparisons, function arguments, and short-circuit boolean expressions.

7.8.6 Variable shadowing and Scoping (Level B)

CREXX Level B introduces block-level scoping for certain control structures. It is important to distinguish between single-instruction branches and grouped instructions (DO blocks).

- **Single-instruction branches**: These execute in the current (procedure) scope.

```
if condition then x = 10 /* x is in procedure scope */
```

- **Grouped instructions (DO blocks)**: These create a `SCOPE_LOCAL` (block scope).
- **Implicit assignments**: An assignment such as `x = 1` uses an existing visible variable. If no visible `x` exists, it creates a variable in the current block scope; that variable ceases to be visible at `END`.
- **Variable Shadowing (Typed)**: A typed declaration (`x = .int`) inside a `DO` block always creates a block-local that shadows any outer variable (or constant) of the same name for the duration of the block.
- **Temporal resolution**: A use in a subscope binds to an outer variable only when that binding already exists. A same-named binding introduced later is a separate variable. Creating or reading that later variable is legal, but the compiler reports `#NOT_IN_SAME_SCOPE` because it commonly indicates an accidental attempt to use the earlier block-local. An explicit declaration such as `x = .int` makes the new binding intentional and suppresses this warning. This check is lexical, not path-sensitive: an early `RETURN`, mutually exclusive branches, or other control-flow fact does not make an implicit same-named binding unambiguous.

This difference in behavior between single-instruction branches and `DO` blocks is an intentional architectural design choice in Level B to balance REXX compatibility with modern structured scoping.

- A typed declaration inside a DO block always creates a block-local that shadows any outer variable (or constant) of the same name for the duration of the block:

```
options levelb
main: procedure
  x = .int; x = 1
  do
    x = .int  /* block-local, shadows outer x */
    x = 9
  end
  say x  /* 1 */
```

- An implicit assignment inside a DO block uses an existing visible variable; otherwise it creates a new block-local variable:

```
options levelb
main: procedure
  do
    y = 3  /* no visible y => creates block-local y */
  end
  say y    /* taken constant Y; compiler warns NOT_IN_SAME_SCOPE */
```

- Counted DO header: the control variable behaves the same way — if an outer variable with that name exists, it is reused; otherwise a loop-local control variable is created. However, you can explicitly force the creation of a block-local control variable that shadows the parent scope by using a typed constructor initializer:

```
options levelb
main: procedure
  i = .int; i = 100
  do i = 1 to 3   /* reuses outer i */
    /* ... */
  end
  /* i is 4 here (the reused outer control variable is one past the
      limit) */

  do j = 1 to 3   /* creates loop-local j */
    /* ... */
  end
  /* j is not visible here */

  do i = .int(1) to 3 /* explicit shadowing loop-local */
    /* ... */
  end
  /* i is still 4 here (the explicit loop-local did not modify it) */
```

Note: This syntax candy `do i = .int(1) to 3` desugars into a wrapped block `do; i = .int; do i = 1 to 3; ...; end; end`.

It is only supported for fundamental types (`.int`, `.string`, `.float`, `.boolean`, `.decimal`). Attempting to instantiate a non-fundamental class in a loop initialization (e.g. `do i = a_class(5) to 10`) will result in a `\#LOOP_CLASSES_NOT_SUPPORTED` compilation error.

Rationale: typed declarations always define intent and should introduce a new local; untyped uses favour existing bindings to reduce surprises, while remaining safe by creating a loop-local when none exists.

8

Programs and Libraries

Level B code is organized as modules that publish procedures, functions, classes, interfaces, methods, factories, and global values. Call resolution is checked by the compiler using the declarations available in the current source file and its imports.

8.1 Procedures and Functions

A callable procedure is declared with a label and `procedure`:

```
twice: procedure = .int
  arg value = .int
  return value * 2
```

Call it by name:

```
say twice(21)
```

The return type after `=` is part of the callable contract. Use `.void` when no value is returned.

Arguments are declared with `arg` lines. Optional arguments, varargs, and caller-owned state are described in the argument and system-symbol chapters.

8.2 Imports

Level B does not assume Classic REXX built-in functions are always visible. Import the library that defines the function:

```
options levelb
import rxfnsb

say date("w")
```

This explicit import model lets the compiler check signatures and lets programs choose which library namespace they use.

8.3 Local Shadowing and Literal Calls

A local procedure shadows an imported function with the same call name. The compiler warns when this happens:

```
options levelb
import rxfnsb

translate: procedure = .string
  arg text = .string
  return "[" || text || "]"
```

Calls to `translate()` in that file resolve to the local procedure.

If code intentionally needs the imported external function while a local symbol shadows it, a literal function call can bypass the local unexposed procedure:

```
say "TRANSLATE"("abc", "ABC", "abc")
```

The compiler emits a warning for this compatibility path. Prefer a clearer namespace or local procedure name in new code when possible.

8.4 Methods and Factories

Methods are called through object values:

```
shape = .box()
say shape.summary()
```

Factories are declared on classes and interfaces. The default factory uses `*`; named factories use their declared member name. Interface factory calls select an implementing class through the current factory-provider rules.

```
asset = .asset("log.txt")
asset2 = .asset.from_size(8)
```

See Classes and Interfaces for factory selection, `match`, casts, type tests, and default methods.

8.5 External Libraries and Plugins

An imported callable may be implemented in cRexx, in RXAS, or in native code through the plugin architecture. The source-level call form is the same:

Runtime loading depends on how the program is run or linked. During compilation, `rxc -s` controls source import roots and `rxc -i` controls binary import roots. At runtime, provide the needed RXBIN modules, linked image, or plugins through the VM or `crexx` driver library path.

8.6 Finding external modules

As described in[5] there are different ways in which Rexx interpreters find the called modules.

[5] Blasco 2023

9

The `OPTIONS` Instruction

`OPTIONS` configures file-level parsing rules and language defaults. When it is present, it must be the first instruction in the source file.

```
options levelb numeric_common comments_slash
```

9.1 Language Level

The Release 1 beta line documents Level B as the supported compiler language:

```
options levelb
```

Other level names identify historical, directional, or DSLSH tooling surfaces. See Language levels for the full catalogue. Do not treat a level name as a release compiler language unless the page describing it explicitly says so.

The compiler can also receive a default with `rxc --level levelb`. A source file's explicit `options` line overrides that command-line default.

The `crexx` driver compiles headerless top-level scripts as Level B and imports `rxfnsb` for convenience. Reusable modules should still write the option and imports explicitly.

9.2 Arithmetic Standard

The file-level arithmetic option changes parser behaviour for arithmetic expressions and sets the default `NUMERIC STANDARD` for procedures that do not override it.

9.2.1 numeric_common

`numeric_common` is the Level B default. It follows common C-like precedence and associativity choices:

- prefix minus has lower priority than power, so `-3**2` is parsed as `-(3**2)`
- power is right-associative, so `2**2**3` is parsed as `2**(2**3)`
- `%` is the common integer/remainder-style operator spelling in Level B

9.2.2 numeric_classic

`numeric_classic` follows Classic REXX arithmetic parsing choices where that mode is used:

- prefix minus has higher priority than power, so `-3**2` is parsed as `(-3)**2`
- power is left-associative, so `2**2**3` is parsed as `(2**2)**3`
- `//` is the Classic remainder spelling

`NUMERIC STANDARD` inside a procedure controls numeric semantics, but it does not reparse expressions. Parser-level choices belong to file-level `OPTIONS`.

9.3 Comment Style

The single-line comment controls are:

- `comments_hash` / `comments_nohash`
- `comments_slash` / `comments_noslash`
- `comments_dash` / `comments_nodash`

Hash comments are enabled by default so a POSIX shebang can be used:

```
#!/usr/bin/env crexx
options levelb
```

Use `comments_slash` to enable `//` comments and `comments_dash` to enable `--` comments.

Block comments use the traditional REXX `/* ... */` form.

9.4 Floating-Point Type

Level B can select how `.float` source values are treated:

- `floats_binary`: binary floating point, the default
- `floats_decimal`: decimal floating point treatment where supported

Use `.decimal` explicitly when decimal behaviour is part of a published signature or value contract. With the default treatment, an otherwise ordinary dotted literal is nevertheless parsed directly from its source spelling when an enclosing decimal assignment, argument, return, cast, constructor, or expression supplies the expected `.decimal` type. `floats_binary` and an explicit `.float(...)` boundary force binary treatment; a `d` suffix explicitly selects decimal.

10 Command line arguments

Support for using command line arguments is built in the Rexx family of languages as fundamental and has its own statement, `arg`, which picks up arguments to procedures as well as to the `main()` procedure, which is by convention the first procedure called by the operating system when a crexx program is started.

As an example,

```
options levelb
import rxfnsb
arg fn = .string[]
/* when no commandline arguments found, display help */
if fn[0]=0 then do
  call help
  exit
end

help: procedure
say 'this program needs arguments'
```

When there is program text before the first procedure, the `main()` procedure is automatically generated. In crexx level B, the arguments to a procedure have a type, in the example this is a string array which is referenced in a variable of type `.string[]`.

After this assignment by `arg fn = .string[]` (in line 3) the number of command line arguments to the program is contained in `fn.0`, which can also be addressed as `fn[0]`. We can now use these arguments in the program, for example by looping over the elements of the string array.

```
/* loop through arguments and find options and program files */
do i=1 to fn.0
  if left(fn.i,1)<>'-' then
    do /* it is not a flag but a filename */
      filename=fn.i
      filenames=strip(filenames) strip(filename)
    end
  else
    do
      if fn.i = '-noexec' then execute=0
[...]
```

These two fragments illustrate how to handle command line arguments to a program; when there are no arguments, `fn.0` is 0 and we call a `help` function to explain about the usage of this program. If there are arguments, we loop over them and decide whether they are options (these start with a dash) or files (if they don't).

10.1 Procedure arguments and expose

Named procedures use the same `ARG` statement to bind call arguments. Level B code normally gives each argument a type:

```
worker: procedure = .int
  arg value = .int, name = .string
  return value
```

Plain `arg name = type` parameters are by value. If the procedure must update the caller's variable, declare that parameter with `expose`:

```
bump: procedure = .void
  arg expose value = .int
  value = value + 1
  return
```

The `procedure expose` clause is different: it exposes module-global storage to the procedure, while `arg expose` exposes a call argument by reference.

Optional arguments can be tested with the `?name` form. A final `... = type` tail accepts a variable number of arguments; those values are available through the pseudo array `arg[]`/`arg.0` and the compatibility `arg()` operator. See Statements on page 151 for the complete procedure, argument, ellipsis, and `arg()` rules.

11 Modules

Each source file compiles to a module. New CREXX source normally uses `.crexx` or `.crx`; `.rexx` remains the compatibility/classic extension. The compiler writes *rxas* assembly (`.rxas`), and the assembler writes *rxbin* bytecode (`.rxbin`).

A module can contain:

- procedures and functions
- global values
- classes and interfaces
- factories, methods, and class/interface metadata
- import and namespace information used by downstream compilation and runtime linking

11.1 Namespace

The namespace is the public identity of reusable code. The source filename is a convenient discovery hint, but the namespace is what callers import and what qualified references name.

```
options levelb
namespace rxfnsb expose abs

abs: procedure = .string
  arg number = .string
  ...
```

Only exposed symbols are intended to be used from outside the namespace.

11.2 Imports

`import name` makes another namespace visible to the current source file:

```
options levelb
import rxfnsb

say date("w")
```

The compiler searches source roots for source imports and binary roots for `.rxbin`, optional `.rxas`, and `.rxplugin` imports. `rxc -s` adds source roots; `rxc -i` adds binary roots. Repeated `-s` and `-i` options are accumulated. The source file's directory is a source root only, not an implicit binary root; use `-i .` or `-i build-dir` when a nearby `.rxbin` should satisfy compile-time imports.

Source roots search `.crexx`, `.crx`, and `.rexx`. If the initial source file uses another extension, such as `.the`, that extension is searched as source for this compile too. Default levels are Level G for `.crexx`, `.crx`, and arbitrary initial extensions, and Level C for `.rexx`, unless the source has an explicit `options level...` clause.

Qualified references use the imported namespace:

```
say rxfnsb..date("w")
```

`namespace::symbol` remains accepted as a compatibility spelling, but new docs and examples should prefer `namespace..symbol`.

11.3 Libraries and Linking

Importing source makes its contracts available during compilation; it does not automatically build or load the imported module. Supply application bytecode with `crexx -l ./library.rxbin program.crexx`, or build the explicit source members together using `crexx --program output program.crexx library.crexx`. Packaged binary imports can carry exact runtime autoload hints. See the worked import and loading example.

Multiple `.rxbin` modules can be loaded together by the VM. For deployable images, use `rxlink`. It combines selected modules into one linked image with a

shared constant pool while preserving module boundaries and runtime metadata.

`rxlink` can also strip source/TRACE debug metadata for smaller deployable artifacts. Runtime contract metadata for imports, interfaces, implementations, members, and factories is preserved by the current linker.

Simple bytecode concatenation can still be useful for development or archive experiments, but `rxlink` is the release path for compact deployable images.

11.4 Classes and Interfaces in Modules

Classes and interfaces are implemented in the current Level B toolchain. Their contracts are recorded in RXBIN metadata so downstream imports can reconstruct headers without reparsing source bodies.

The current metadata model carries:

- interface declarations
- class implementation relationships
- callable members
- factories and method contracts

See Classes and Interfaces for the source model.

12 Parsing templates

The `parse` instruction allows a selected string to be parsed (split up) and assigned to variables, under the control of a *template*.

The various mechanisms in the template allow a string to be split up by explicit matching of strings (called *patterns*), or by specifying numeric positions (*positional patterns* - for example, to extract data from particular columns of a line read from a character stream). Once split into parts, each segment of the string can then be assigned to variables as a whole or by words (delimited by blanks).

This section first gives some informal examples of how the parsing template can be used, and then defines the algorithms in detail.

12.1 Introduction to parsing

The simplest form of parsing template consists of a list of variable names. The string being parsed is split up into words (characters delimited by blanks), and each word from the string is assigned to a variable in sequence from left to right. The final variable is treated specially in that it will be assigned whatever is left of the original string and may therefore contain several words. For example, in the `parse` instruction:

```
parse 'This is a sentence.' v1 v2 v3
say v1
say v2
say v3
```

the term (in this case a literal string) following the instruction keyword is parsed, and then: the variable ***v1*** would be assigned the value "**This**", ***v2*** would be assigned the value "**is**", and ***v3*** would be assigned the value "**a sentence.**".

```
This
is
a sentence.
```

Leading blanks are removed from each word in the string before it is assigned to a variable, as is the blank that delimits the end of the word. Thus, variables set in this manner (***v1*** and ***v2*** in the example) will never have leading or trailing blanks, though ***v3*** could have both leading and trailing blanks. Note that the variables assigned values in a template are always given a new value and so if there are fewer words in the string than variables in the template then the unused variables will be set to the null string. The second parsing mechanism uses a literal string in a template as a pattern, to split up the string. For example:

```
parse 'To be, or not to be?' w1 ',' w2
say w1
say w2
```

would cause the string to be scanned for the comma, and then split at that point; the variable ***w1*** would be set to "**To be**", and ***w2*** is set to " **or not to be?**". Note that the pattern itself (and **only** the pattern) is removed from the string.

```
To be
or not to be?
```

Each section of the string is treated in just the same way as the whole string was in the previous example, and so either section could be split up into words. Thus, in:

```
parse 'To be, or not to be?' w1 ',' w2 w3 w4
say w1
say w2
say w3
say w4
```

w2 and ***w3*** would be assigned the values "**or**" and "**not**", and ***w4*** would be assigned the remainder: "**to be?**".

```
To be
or
not
to be?
```

If the string in the last example did not contain a comma, then the pattern would effectively "match" the end of the string, so the variable to the left of the pattern would get the entire input string, and the variables to the right would be set to a null string. The pattern may be specified as a variable, by putting the variable name in parentheses. The following instructions therefore have the same effect as the last example:

```
c=','
parse 'To be, or not to be?' w1 (c) w2 w3 w4
say w1
say w2
say w3
say w4
```

```
To be
or
not
to be?
```

The third parsing mechanism is the numeric positional pattern. This works in the same way as the string pattern except that it specifies a column number. So:

```
parse 'Flying pigs have wings' x1 5 x2
say x1
say x2
```

would split the string at the fifth column, so ***x1*** would be "**Flyi**" and ***x2*** would start at column 5 and so be "**ng pigs have wings**".

```
Flyi
ng pigs have wings
```

More than one pattern is allowed, so for example:

```
parse 'Flying pigs have wings' x1 5 x2 10 x3
say x1
```

```
say x2
say x3
```

```
Flyi
ng pi
gs have wings
```

would split the string at columns 5 and 10, so ***x2*** would be "**ng pi**" and ***x3*** would be "**gs have wings**". The numbers can be relative to the last number used, so:

```
parse 'Flying pigs have wings' x1 5 x2 +5 x3
say x1
say x2
say x3
```

```
Flyi
ng pi
gs have wings
```

would have exactly the same effect as the last example; here the **+5** may be thought of as specifying the length of the string to be assigned to ***x2***. As with literal string patterns, the positional patterns can be specified as a variable by putting the name of a variable, in parentheses, in place of the number. An absolute column number should then be indicated by using an equals sign ("=") instead of a plus or minus sign. The last example could therefore be written:

```
start=5
length=5
data='Flying pigs have wings'
parse data  x1 =(start) x2 +(length) x3
say x1
say x2
say x3
```

```
Flyi
ng pi
gs have wings
```

String patterns and positional patterns can be mixed (in effect the beginning of a string pattern just specifies a variable column number) and some very powerful

things can be done with templates. The next section describes in more detail how the various mechanisms interact.

12.2 Parsing definition

This section describes the rules that govern parsing. In its most general form, a template consists of alternating pattern specifications and variable names. Blanks may be added between patterns and variable names to separate the tokens and to improve readability. The patterns and variable names are used strictly in sequence from left to right, and are used once only. In practice, various simpler forms are used in which either variable names or patterns may be omitted; we can therefore have variable names without patterns in between, and patterns without intervening variable names. In general, the value assigned to a variable is that sequence of characters in the input string between the point that is matched by the pattern on its left and the point that is matched by the pattern on its right. If the first item in a template is a variable, then there is an implicit pattern on the left that matches the start of the string, and similarly if the last item in a template is a variable then there is an implicit pattern on the right that matches the end of the string. Hence the simplest template consists of a single variable name which in this case is assigned the entire input string. Setting a variable during parsing is identical in effect to setting a variable in an assignment. The constructs that may appear as patterns fall into two categories; patterns that act by searching for a matching string (literal patterns), and numeric patterns that specify an absolute or relative position in the string (positional patterns). Either of these can be specified explicitly in the template, or alternatively by a reference to a variable whose value is to be used as the pattern. For the following examples, assume that the following sample string is being parsed; note that all blanks are significant - there are two blanks after the first word “is” and also after the second comma:

```
'This is  the text which, I think,  is scanned.'
```

12.2.1 Parsing with literal patterns

Literal patterns cause scanning of the data string to find a sequence that matches the value of the literal. Literals are expressed as a quoted string. The null string matches the end of the data. The template:

```
tx ='This is  the text which, I think,  is scanned.'
parse tx w1 ',' w2 ',' w3
say 'w1 has the value "'w1'"'
say 'w2 has the value "'w2'"'
say 'w3 has the value "'w3'"'
```

when parsing the sample string, results in:

```
w1 has the value "This is  the text which"
w2 has the value " I think"
w3 has the value "  is scanned."
```

Here the string is parsed using a template that asks that each of the variables receive a value corresponding to a portion of the original string between commas; the commas are given as quoted strings. Note that the patterns themselves are removed from the data being parsed. A different parse would result with the template:

```
tx ='This is  the text which, I think,  is scanned.'
parse tx w1 ',' w2 ',' w3 ',' w4
say 'w1 has the value "'w1'"'
say 'w2 has the value "'w2'"'
say 'w3 has the value "'w3'"'
say 'w4 has the value "'w4'"'
```

which would result in:

```
w1 has the value "This is  the text which"
w2 has the value " I think"
w3 has the value "  is scanned."
w4 has the value ""
```

This illustrates an important rule. When a match for a pattern cannot be found in the input string, it instead "matches" the end of the string. Thus, no match was found for the third **,** in the template, and so ***w3*** was assigned the rest of the string. ***w4*** was assigned a null string because the pattern on its left had already reached the end of the string. Note that **all** variables that appear in a template in this way are assigned a new value.

12.2.2 Parsing strings into words

If a variable is directly followed by one or more other variables, then the string selected by the patterns is assigned to the variables in the following manner. Each blank-delimited word in the string is assigned to each variable in turn, except for the last variable in the group (which is assigned the remainder of the string). The values of the variables which are assigned words will have neither leading nor trailing blanks. Thus the template:

```
tx ='This is  the text which, I think,  is scanned.'
parse tx w1 w2 w3 w4 ','
say 'w1 has the value "'w1'"'
say 'w2 has the value "'w2'"'
say 'w3 has the value "'w3'"'
say 'w4 has the value "'w4'"'
```

would result in:

```
w1 has the value "This"
w2 has the value "is"
w3 has the value "the"
w4 has the value "text which"
```

Note that the final variable (***w4*** in this example) could have had both leading blanks and trailing blanks, since only the blank that delimits the previous word is removed from the data. Also observe that this example is not the same as specifying explicit blanks as patterns, as the template:

```
tx ='This is  the text which, I think,  is scanned.'
parse tx w1 ' ' w2 ' ' w3 ' ' w4 ','
say 'w1 has the value "'w1'"'
say 'w2 has the value "'w2'"'
say 'w3 has the value "'w3'"'
say 'w4 has the value "'w4'"'
```

would in fact result in:

```
w1 has the value "This"
w2 has the value "is"
w3 has the value ""
w4 has the value "the text which"
```

since the third pattern would match the third blank in the data. In general, when a variable is followed by another variable then parsing of the input into individual words is implied. The parsing process may be thought of as first splitting the original string up into other strings using the various kinds of patterns, and then assigning each of these new strings to (zero or more) variables.

12.2.3 Use of the period as a placeholder

A period (separated from any symbols by at least one blank) acts as a placeholder in a template. It has exactly the same effect as a variable name, except that no variable is set. It is especially useful as a "dummy variable" in a list of variables, or to collect (ignore) unwanted information at the end of a string. Thus the template:

```
. . . word4 .
```

("**text**") from the sample string and place it in the variable ***word4***. Blanks between successive periods in templates may be omitted, so the template:

```
... word4 .
```

would have the same result as the last template.

12.2.4 Parsing with positional patterns

Positional patterns may be used to cause the parsing to occur on the basis of position within the string, rather than on its contents. They take the form of whole numbers, optionally preceded by a plus, minus, or equals sign which indicate relative or absolute positioning. These may cause the matching operation to "back up" to an earlier position in the data string, which can only occur when positional patterns are used.

Absolute positional patterns:

A number in a template that is **not** preceded by a sign refers to a particular (absolute) character column in the input, with 1 referring to the first column. For example, the template:

```
tx ='This is  the text which, I think,  is scanned.'
parse tx s1 10 s2 20 s3
say 's1 has the value "'s1'"'
```

```
say 's2 has the value "'s2'"'
say 's3 has the value "'s3'"'
```

results in:

```
s1 has the value "This is  "
s2 has the value "the text w"
s3 has the value "hich, I think,  is scanned."
```

Here ***s1*** is assigned characters from the first through the ninth character, and ***s2*** receives input characters 10 through 19. As usual the final variable, ***s3***, is assigned the remainder of the input.

An equals sign ("=") may be placed before the number to indicate explicitly that it is to be used as an absolute column position; the last template could have been written:

```
s1 =10 s2 =20 s3
```

A positional pattern that has no sign or is preceded by the equals sign is known as an *absolute positional pattern*.

Relative positional patterns

A number in a template that is preceded by a plus or minus sign indicates movement relative to the character position at which the previous pattern match occurred. This is a *relative positional pattern*. If a plus or minus is specified, then the position used for the next match is calculated by adding (or subtracting) the number given to the last matched position. The last matched position is the position of the first character of the last match, whether specified numerically or by a string.

For example, the instructions:

```
parse '123456789'  3 w1 +3 w2 3 w3
say 'w1 has the value "'w1'"'
say 'w2 has the value "'w2'"'
say 'w3 has the value "'w3'"'
```

result in

```
w1 has the value "345"
w2 has the value "6789"
w3 has the value "3456789"
```

The **+3** in this case is equivalent to the absolute number **6** in the same position, and may also be considered to be specifying the length of the data string to be assigned to the variable ***w1***. This example also illustrates the effects of a positional pattern that implies movement to a character position to the left of (or to) the point at which the last match occurred. The variable on the left is assigned characters through the end of the input, and the variable on the right is, as usual, assigned characters starting at the position dictated by the pattern. A useful effect of this is that multiple assignments can be made:

```
parse x 1 w1 1 w2 1 w3
```

This results in assigning the (entire) value of ***x*** to ***w1***, ***w2***, and ***w3***. (The first "**1**" here could be omitted as it is effectively the same as the implicit starting pattern described at the beginning of this section.) If a positional pattern specifies a column that is greater than the length of the data, it is equivalent to specifying the end of the data (*i.e.*, no padding takes place). Similarly, if a pattern specifies a column to the left of the first column of the data, this is not an error but instead is taken to specify the first column of the data. Any pattern match sets the "last position" in a string to which a relative positional pattern can refer. The "last position" set by a literal pattern is the position at which the match occurred, that is, the position in the data of the *first* character in the pattern. The literal pattern in this case is **not** removed from the parsed data. Thus the template:

```
',' -1 x +1
```

will:

1. Find the first comma in the input (or the end of the string if there is no comma).
2. Back up one position.
3. Assign one character (the character immediately preceding the comma or end of string) to the variable ***x***.

One possible application of this is looking for abbreviations in a string. Thus the instruction:

```
/* Ensure options have a leading blank and are
   in uppercase before parsing. */
parse (' 'opts).upper ' PR' +1 prword ' '
```

will set the variable ***prword*** to the first word in ***opts*** that starts with "**PR**" (in any case), or will set it to the null string if no such word exists. **Notes:**

1. The positional patterns **+0** and **-0** are valid, have the same effect, and may be used to include the whole of a previous literal (or variable) pattern within the data string to be parsed into any following variables.
2. As illustrated in the last example, patterns may follow each other in the template without intervening variable names. In this case each pattern is obeyed in turn from left to right, as usual.
3. There may be blanks between the sign in a positional pattern and the number, because CREXX defines that blanks adjacent to special characters are removed.

12.2.5 Parsing with variable patterns

It is sometimes desirable to be able to specify a pattern by using the value of a variable instead of a fixed string or number. This may be achieved by placing the name of the variable to be used as the pattern in parentheses (blanks are not necessary either inside or outside the parentheses, but may be added if desired). This is called a *variable reference*; the value of the variable is converted to string before use, if necessary. If the parenthesis to the left of the variable name is not preceded by an equals, plus, or minus sign ("=", "+", or "-") the value of the variable is then used as though it were a literal (string) pattern. The variable may be one that has been set earlier in the parsing process, so for example:

```
input="L/look for/1 10"
parse input  verb 2 delim +1 string (delim) rest
say "verb to "'"verb"'"
say "delim to "'"delim"'"
say "string to "'"string"'"
say "rest to "'"rest"'"
```

will set:

```
verb to 'L'
delim to '/'
string to 'look for'
rest to '1 10'
```

If the left parenthesis **is** preceded by an equals, plus, or minus sign then the value of the variable is used as an absolute or relative positional pattern (instead of as a literal string pattern). In this case the value of the variable must be a non-negative whole number, and (as before) it may have been set earlier in the parsing process.

12.2.6 PARSE VERSION

`PARSE VERSION` uses the string returned by the `version()` built-in function as its source and applies the normal template rules.

For example:

```
parse version version
say version
```

```
macOS 64 crexx-1.0.0-beta.3+local.g4c4f4c48643e 20260914
```

assigns the complete version string to version, while:

```
parse version version level
say version
say level
```

version string to version and level.

```
macOS
64 crexx-1.0.0-beta.3+local.g4c4f4c48643e 20260914
```

12.2.7 PARSE SOURCE

`PARSE SOURCE` uses the string returned by the `sourceinfo()` built-in function as its source and applies the normal template rules.

The source string contains:

- execution environment
- invocation mode
- source file name

For example:

```
parse source environment mode filename
say environment
say mode
say filename
```

assigns the corresponding components of the source information to the specified variables using the standard PARSE rules.

> macOS
>
> COMMAND
>
> parse32.crexx

13

Queue, Push and Pull

The current implementation supports the following language features:

- `PUSH`
- `QUEUE`
- `PULL`
- `PARSE PULL`

The queue is local to the current CREXX execution and is shared by the main program and all nested procedures.

At this stage, the implementation does not support[6]:

- named queues;
- externally managed queues;
- queue sharing between independent CREXX executions.

`PARSE PULL` has been integrated into the existing `PARSE` implementation and behaves as expected.

An example:

```
options levelb

main: procedure
  first = .string
```

[6] A future enhancement may introduce externally managed named queues. One possible approach would be to implement a lightweight TCP-based queue manager, allowing independent CREXX executions to access the same named queues without changing the REXX language interface.

```
second = .string
parsed_first = .string
push "front"
queue "back"

pull first
pull second

say first
say second

queue "classic-value"
parse pull parsed_first

say parsed_first
```

```
front
back
classic-value
```

PART III

Reference

14

Character Set

14.1 Characters, text, and binary data

Character handling in CREXX involves two separate concerns:

- the characters and encoding accepted in a CREXX source program;
- the representation of text and byte data manipulated by the compiled program.

These concerns are related, but they are not the same. The encoding of a source file determines how the compiler reads the program. It does not, by itself, determine how the program interprets data read from a file, received over a network connection, or supplied by an external library.

14.1.1 Source-program encoding

In normal builds, CREXX source files are UTF-8 text. This permits comments, string literals, and other textual source content to contain Unicode characters.

The fixed elements of the language use the portable ASCII subset. These include language keywords, operators, punctuation, compiler directives, and symbols generated for RXAS. Keeping the language syntax within ASCII makes source programs easier to exchange between development environments and avoids dependence on locale-specific character variants.

The use of UTF-8 for source files does not imply that every Unicode character

is meaningful in every syntactic position. The lexical rules of the language still determine which characters may occur in identifiers, numeric literals, operators, and other language elements.

14.1.2 Text values

String values are intended for textual data and may contain Unicode text. Unicode separates characters from their encoded byte representation: the same text may be represented differently when written using UTF-8, UTF-16, EBCDIC, or another encoding.

Encoding and decoding therefore arise at the boundaries of a program. Input bytes must be decoded using the encoding associated with their source, and text must be encoded appropriately when it is written to a file, terminal, network connection, database, or external API. The encoding of the CREXX source file does not automatically determine the encoding of such external data.

Operations on text should consequently be understood in terms of the CREXX string model rather than assumed to be operations on an arbitrary sequence of bytes. This distinction is particularly important for non-ASCII characters, which may occupy more than one byte in UTF-8.

14.1.3 Binary values

A `.binary` value represents byte-oriented data that must not be interpreted as text. It is appropriate for data such as executable code, compressed content, images, cryptographic material, protocol packets, and records whose layout is defined in terms of bytes.

Binary data has no inherent character encoding. A byte sequence becomes text only when it is decoded using a specified encoding. Conversely, converting a string to binary data requires an encoding that defines how its characters are to be represented as bytes.

Keeping text and binary data distinct prevents accidental character conversion. It also makes the programmer's intention explicit: strings hold text, whereas `.binary` values preserve byte values.

14.1.4 Platform considerations

On older[7] or specialist platforms, source conversion may be included in the build, transfer, or packaging workflow. For example, a development system may store source as UTF-8 while a platform tool expects another native encoding. Such conversion concerns the representation of the source program and should not be confused with conversions performed by the running program on its input and output data.

A conversion process must preserve all characters used by the source. This is straightforward for the ASCII-based language syntax, but comments and string literals may contain Unicode characters that are not representable in a legacy character set. Conversion should therefore be controlled explicitly and should not rely on an implicit system locale.

Unless stated otherwise, CREXX release documentation, source distributions, and examples assume UTF-8 source files.

[7] This can be EBCDIC for builds for historic mainframe operating systems like MVS and VM/CMS. Modern mainframe operating systems can handle UTF-8.

15 General Syntax

A program is built up out of a series of *clauses* that are composed of: zero or more blanks (which are ignored); a sequence of tokens (described in this section); zero or more blanks (again ignored); and the delimiter "**;**" (semicolon) which may be implied by line-ends or certain keywords. Conceptually, each clause is scanned from left to right before execution and the tokens composing it are resolved.

Identifiers (known as symbols) and numbers are recognized at this stage, comments (see *Comments* on page 79) are removed, and multiple blanks (except within literal strings) are reduced to single blanks. Blanks adjacent to operator characters and special characters are also removed.

15.1 Blanks and White Space

Blanks (spaces) may be freely used in a program to improve appearance and layout, and most are ignored. Blanks, however, are usually significant

- within literal strings (see below)
- between two tokens that are not special characters (for example, between two symbols or keywords)
- between the two characters forming a comment delimiter
- immediately outside parentheses ("**(**" and "**)**") or brackets ("[" and "]").

Tabulation (tab) and form feed characters outside of literal strings are treated as if

they were a single blank; similarly, if the last character in a program is the End-of-file character (EOF, encoded in ASCII as decimal 26), that character is ignored.

15.2 Tokens

The essential components of clauses are called *tokens*. These may be of any length, unless limited by implementation restrictions, [^1] and are separated by blanks, comments, ends of lines, or by the nature of the tokens themselves.

The tokens are:

To illustrate how a clause is composed out of tokens, consider this example:

```
'REPEAT'   B + 3;
```

This is composed of six tokens: a literal string, a blank operator (described later), a symbol (which is probably the name of a variable), an operator, a second symbol (a number), and a semicolon. The blanks between the ”**B**” and the ”**+**” and between the ”**+**” and the ”**3**” are removed. However one of the blanks between the **‘REPEAT’** and the ”**B**” remains as an operator. Thus the clause is treated as though written:

```
'REPEAT' B+3;
```

15.3 Implied semicolons and continuations

A semicolon (clause end) is implied at the end of each line, except if:

1. The line ends in the middle of a block comment, in which case the clause continues at the end of the block comment.
2. The last token was a hyphen. In this case the hyphen is functionally replaced by a blank, and hence acts as a *continuation character*.

This means that semicolons need only be included to separate multiple clauses on a single line.

Notes:

1. A comment is not a token, so therefore a comment may follow the continu-

ation character on a line.

2. Semicolons are added automatically by after certain instruction keywords when in the correct context. The keywords that may have this effect are , , , ; they become complete clauses in their own right when this occurs. These special cases reduce program entry errors significantly.

15.4 The case of names and symbols

In general, CREXX is a *case-insensitive* language. That is, the names of keywords, variables, and so on, will be recognized independently of the case used for each letter in a name; the name ”**Swildon**” would match the name ”**swilDon**”.

16 Comments

Commentary is included in a CREXX program by means of comments. CREXX has single-line comments and multiline[8] comments.

16.1 Multiline comments

A multiline comment is started by a sequence of special characters /*, and ended by the inverse character combination */ . Within these delimiters any characters are allowed. Comments may be nested, which is to say that /* and */ must pair correctly. Comments may be anywhere, and may be of any length. They have no effect on the program, except that they do act as separators (i.e., two tokens with just a comment in between are not treated as a single token). These multiline comments do not need to straddle lines; it is ok to have them on one line only.

Multiline ('box') comments can be simple, or of the *flower pot* variety:

```
/************************************
 *  This is a flower pot comment.
 *  Every line starts with '*'.
 *  Often used with headers.
 ************************************/
```

Also often seen is the lighter style where the asterisks are replaced by dashes in lines:

[8] Also called block comments.

```
/*-----------------------------------*
 *  Module: parser.crexx             *
 *  Purpose: expression parsing      *
 *-----------------------------------*/
```

or even

```
/*************************************************************\
*                                                             *
*           C R E X X   C O M P I L E R                       *
*                                                             *
\*************************************************************/
```

16.2 Doc (Documentation) Comments

Doc comments are a special form of multiline comment which starts with `/**` and ends in `*/`. It is not processed by the CREXX compiler but can aid in the producing of documentation by external tools. In doc commments special tags as `@parm`, `@result` and `@author` can be used, purely for documentation purposes and with no consequence for the behaviour of the program.

The doc comment style used in the CREXX class library is:

```
/**
* method insert
 * Inserts an element at the specified index.
 *
 * Elements at, and after the index are shifted to the right.
 *
 * @param index  1-based index where the element will be inserted
    .
 * @param item   The element to insert.
 */
```

for readability purposes.

16.3 Single line comments

For CREXX level B, the *octothorpe* (#), also know as poundsign or hashmark, is the default for single line comments. Other single line comment styles can be chosen by the `option` statements `comments_dash` and `comments_slash` which enable the

SQL, NetRexx and Object Rexx `--` double dash style, and `comments_slash` which enable the C++ style `//` line comments.

16.3.1 Shebang

The # comment convention is a natural fit for the Unix, Linux and macOS convention where the first line of a program (or shell script) indicates the language processor to be used. This mechanism (sometimes called *shebang*, after the earlier *hash-bang*, which received its name after the colloquial names for the characters '#' (*hash*) and the '!' (*bang*) has its roots in the way an executable file is loaded on these operating systems. To execute a script `hello.crexx` (which needs to be made executable with the command `chmod +x hello.crexx`) we need to make sure that the *shebang* on its first line resembles this:

```
#!/usr/bin/env crexx
say 'hello shebang!'
```

It can then be invoked with `./hello.crexx` from the current directory.

16.4 Summary

TABLE 1: Comment options.

option	comment type	symbol
default	multiline	`/* */`
default	single line	`#`
options comments_dash	single line	`--`
options comments_slash	single line	`//`

16.5 Example

```
options levelb comments_dash comments_slash
/* rexx - slash star start slash
   multiline comments always work  */
# a hash comment (single line also always works
import rxfnsb
/**
 * this is a so-called doc comment
```

```
 * which has a special role in documenting code
 */
say 'comment on comments'
// this comment is enabled by option comments_slash
-- this comment is enabled by comments_dash
say 'no further comment.'
```

17

Data Types

Level B is statically typed. Every expression has a type known to the compiler, and assignments, arguments, returns, method calls, and factory calls are checked against those types.

17.1 Built-In Types

The built-in Level B value types are:

Type	Purpose
`.void`	No usable value. Used for procedures that do not return a value.
`.boolean`	Boolean truth value.
`.int`	Integer value.
`.float`	Binary floating-point value unless the source uses decimal float options.
`.decimal`	Decimal numeric value.
`.string`	Character string value.
`.binary`	Binary byte sequence.
`.object`	Object value. Interfaces and classes are object-shaped contracts.

The canonical integer spelling in source is `.int`. In Release 1 it is a signed 64-bit value on every supported desktop architecture, with the inclusive range `-9223372036854775808` through `9223372036854775807`. It is not sized from the host C `long`, pointer, or

native word width.

17.2 Constructors and Type Literals

Type names can be used as constructors:

```
count = .int(0)
name  = .string("Ada")
ok    = .boolean(1)
payload  = .binary()
```

Class and interface names also use the dotted form. A factory call creates an object through the selected class or interface provider:

```
item  = .cacheentry("abc")
asset = .asset("log.txt")
```

Without the call brackets, a class name denotes the typed default value for that class. For object classes this is a typed but uninitialized object value; it can be tested or cast for object/class/interface compatibility, but method calls or attribute access require an initialized instance and raise `OBJECT_NOT_INITIALIZED` otherwise. Use the factory form when an initialized object is wanted:

```
pending  = .cacheentry       /* typed, not initialized */
ready = .cacheentry("abc")  /* initialized by the factory */
say initialized(pending)    /* 0 */
say initialized(ready)      /* 1 */
```

Namespace-qualified contracts use a double dot:

```
client = .net..httpclient("example.com", 443, 1)
```

The left side of `namespace..symbol` must name an imported namespace. `namespace::symbol` remains accepted as a compatibility alias.

17.3 Arrays

Arrays are declared from a base type:

```
words = .string[]
numbers  = .int[10]
grid  = .int[10, 10]
window   = .int[0 to 10]
grow  = .int[-2 to *]
```

These are declarations, not calls that produce fresh arrays on each execution. An exposed array keeps its existing elements across calls to a procedure that declares its type. Use `call arraydrop words` (from `rxfnsb`) to clear an existing growable array. This differs from an object factory expression such as `.stem()`, which constructs an object when called.

An array value carries its element type and dimensions. Procedure signatures can accept arrays in the same way:

```
main: procedure = .int
  arg args = .string[]
```

Array elements can be accessed with bracket notation or Rexx-style dotted notation:

```
items[1] = "alpha"
items.2   = "beta"
```

Simple growable arrays are often used with one-based element indexes, but Level B arrays are not inherently limited to one-based indexing; explicit bounds such as `.int[0 to 10]` and `.int[-2 to *]` are part of the current source surface. For classic Rexx compound-variable style keyed string data, use the Level B `.stem` class from `rxfnsb`.

17.4 References

Reference values are explicit weak aliases to storage. A reference does not keep its target alive; if the target storage is destroyed, later use raises `REFERENCE_INVALID`. Use `reference` as a type modifier anywhere a Level B type is accepted:

```
count_ref = reference .int
items_ref = reference .string[]

read_count: procedure = .int
  arg r = reference .int
```

Use `reference target` to create a reference to aliasable storage, `local = dereference ref` when a local should become a scoped live link to the referenced target, and `snapshot ref` when an explicit deep copy is required:

```
count   = 1
count_ref = reference count
linked     = dereference count_ref
copy    = snapshot count_ref
```

`dereference` is only valid as the right side of an assignment to a local variable in the current procedure or block scope. Assigning a dereference into an object/class attribute, array element, global, exposed argument, or arbitrary expression is a compile-time error. The compiler emits `unlink` when the local's scope exits, and the VM also resets linked locals when a frame exits.

Reference values are not assignment-compatible with their target type. Passing a `.T` where `reference .T` is expected is an error, and passing `reference .T` where `.T` is expected is also an error; spell `reference target`, `local = dereference ref`, or `snapshot ref` at the boundary.

Reference values do not define value equality or ordering. Applying an ordinary comparison operator such as = or == to a reference is a compile-time error rather than an implicit comparison of target values or storage identity. The explicit `left <refsame> right` operator accepts compatible reference types and tests whether both retain the same storage-identity cell without dereferencing them. Copied references therefore remain `<refsame>` after their common target expires; cleared or empty references compare false. Use `<refvalid>(ref)` to test live-target validity, and explicitly `dereference` or `snapshot` when the target value itself is to be compared.

Nested reference containers, reference casts, reference type tests, and implicit member/index access through a reference are not part of the current Level B source surface. These are reserved for possible Level G convenience features. Level B code should keep reference boundaries explicit with `reference`, `dereference`, `snapshot`, `<refvalid>`, and `<refsame>`.

17.5 Numeric Values

Level B supports integer, float, and decimal arithmetic. The file-level `options` instruction selects the parser's arithmetic standard, and a procedure-level `numeric` instruction can set numeric context such as digits, form, fuzz, case, and standard.

Integer literals and conversions outside the signed 64-bit range are rejected. Runtime integer add, subtract, multiply, power, increment, decrement, negation, and the `INT64_MIN / -1` and `INT64_MIN % -1` edges raise `OVERFLOW_UNDERFLOW`; integer division or modulo by zero raises `DIVISION_BY_ZERO`.

The compiler performs type validation before bytecode emission. Numeric conversions are explicit where precision or representation could otherwise be surprising:

```
i = .int(42)
f = .float(i)
d = .decimal("42.50")
```

An ordinary dotted literal remains binary `.float` when no surrounding type requires decimal. When a decimal assignment, argument, return, cast, constructor, or decimal expression establishes the expected type, the compiler parses the literal's original source spelling directly as `.decimal`; it does not round through binary64 first. For example, `d = 0.1` is exact when `d` is declared `.decimal`. An explicit `.float(...)` boundary and `options floats_binary` retain binary treatment. The `d` suffix is also an explicit decimal spelling, such as `0.1d`.

The checked cast form can also be used for scalar conversions:

```
f  = 1 as .float
i  = "42" as .int
d  = "42.50" as .decimal
s  = 42 as .string
ok = "1" as .boolean
```

Scalar casts use the same conversion rules as the corresponding constructor or promotion opcode. A cast is still type checked; for example, `.binary` values only cast back to `.string` when the cast is explicit and the bytes are valid UTF-8.

17.6 Strings and Binary Values

`.string` values are character data. `.binary` values are byte data. Keep the two distinct when working with sockets, files, encodings, or native payloads: string operations are text operations, while binary operations preserve bytes.

In UTF builds, `.string` source values are valid UTF-8 text. Converting a string to `.binary` stores the exact UTF-8 bytes currently held by the string; the conversion does not normalize, transcode, or reinterpret the text:

```
payload = "alpha" as .binary
```

Converting `.binary` to `.string` validates the byte sequence as UTF-8:

```
payload  = "ceb1"x as .binary
text  = payload as .string     /* "□" */
```

An invalid binary-to-string conversion raises `UNICODE_ERROR` at runtime. If the invalid bytes are visible as a constant literal in the cast, the compiler rejects the program with `CANNOT_CAST_BINARY`.

Invalid UTF-8 byte sequences are only valid in an explicit binary context:

```
payload  = .binary
payload  = 'ffff'x

other = 'ffff'x as .binary
```

A first untyped assignment such as `payload = 'ffff'x` is treated as a text assignment and is rejected when the decoded bytes are not valid UTF-8. That rule keeps accidental invalid text out of string operations; use `.binary` when the program is handling bytes.

Binary concatenation is byte concatenation. If either operand of `||` is `.binary`, the expression result is `.binary`; string operands in that binary expression are converted to their exact UTF-8 bytes. Blank concatenation remains a text operation and should not be used for binary payload assembly.

```
prefix = "ff"x as .binary
packet = prefix || "OK"      /* bytes ff 4f 4b */
```

Runtime-owned `.binary` storage is aligned for the host's native `.int` and `.float` representations. Level B can use that guarantee through zero-based `<packed..int>(item)` and `<packed..float>(item)` reads and writes. These are explicit host-native views over bytes, not new value types and not portable encodings. See Binary Memory. Release 1 Level G provides explicit `.packedint` and `.packedfloat` owner classes in `rxfnsg`; those classes wrap this same representation and do not change ordinary arrays. Automatically packed ordinary `.int[]` and `.float[]` arrays remain a post-release Level G roadmap item.

Level B string length, indexing, slicing, search, and reversal use Unicode codepoints. They do not use UTF-8 bytes and do not imply grapheme boundaries. Normalization and full case folding are explicit future Level G services; no string conversion or ordinary Level B operation normalizes implicitly.

Level C direct BIFs retain the `.string`/`.binary` distinction but apply the character profile held by their call context. `BYTE` treats exact bytes as Classic character units; `UTF8` requires valid UTF-8 for text operations and uses codepoint units. A value's binary/text cache flags do not select that profile. See Unicode.

The `rxfnsb` library provides byte-oriented helpers for common binary work: `binlength`, `binbyte`, `binsetbyte`, `binsubstr`, `binconcat`, `binoverlay`, `bininsert`, `bindelstr`, `binpos`, `bincompare`, `bin2x`, and `x2bin`. For packed binary layouts, use the binary-memory intrinsics and helpers in Binary Memory; those use zero-based byte offsets and can operate directly on binary constants.

The same boundary applies outside source literals. Native RXVML string setters, `CREXXSAA ADDRESS` variable setters, RXPA native return / argument trees, command-line arguments passed through RXVML, `ADDRESS` callback text, text file reads, socket text reads, and explicit binary-to-string casts validate UTF-8 in normal Level B builds. Invalid bytes should be read or carried as `.binary` first, then decoded to `.string` only when the program has a valid encoding.

17.7 Object Values

Classes and interfaces are object contracts. A concrete class instance can be assigned to an interface it implements:

```
shape = .box()
```

Level B supports:

- `expr is .type` for boolean type tests
- `expr as .type` for checked casts
- `typeof(expr)` for concrete type introspection
- `initialized(expr)` for testing whether an object value has completed factory initialization; non-object values are considered initialized

Objects can also carry native payloads when exposed through the plugin API, but ordinary Level B code should interact with objects through factories, methods, and interfaces.

Level B does not define implicit object-to-string promotion. Statements such as `say value`, string concatenation, and string comparison operate on values whose types are already string-compatible under the Level B type rules; they do not automatically call a `toString()` method on arbitrary objects. A future Level G object-promotion capability is still undesigned. The likely direction is an explicit contract, such as a supported interface for string rendering, rather than a convention based only on a method name.

17.8 Type Inference

The compiler infers a variable type from the first binding in many common cases:

```
total = 0        /* .int */
label = "ready" /* .string */
```

Use explicit constructors or declarations when the intended type is not obvious from the initializer, when a signature is being published, or when a value must be an object or array contract.

A bare type expression can also declare a typed local before the value is known:

```
slot = .int
if found then slot = existing
else slot = created
```

Use this form when the declaration is the important point. Use a literal initializer, such as `slot = 0`, only when that literal value is part of the program logic.

18

Binary Memory

This chapter specifies the Release 1 binary-memory source surface for Level B. It builds on the RXAS binary-memory instruction baseline and defines the forms that the compiler lowers directly for packed lookup and record code.

Binary memory treats a `.binary` value or `.binary` constant as an indexed byte space. It is intended for packed lookup structures, indexes, parsers, protocol records, and other cases where repeatedly copying strings or binary slices would dominate runtime cost.

18.1 Core Rules

- Offsets are zero-based byte offsets.
- Lengths are byte counts unless the form explicitly says otherwise.
- `.binary` variables and `.binary` constants use the same read syntax.
- Writes require a mutable `.binary` variable; constants are read-only.
- Fixed-width numeric fields use canonical little-endian storage.
- Fixed-width numeric fields do not use host byte order. If a file, protocol, or network payload uses a different byte order, the program must use the correct layout convention or conversion helper for that format.
- `.string` fields are UTF-8. Invalid UTF-8 raises `UNICODE_ERROR` when a string value is materialized or compared as a string field.

- Ordinary = compares ordinary materialized REXX values. Zero-copy binary memory compare uses explicit compare intrinsics.

Do not use `[]` to address binary bytes. Brackets remain array indexing. Binary memory uses intrinsic forms so an array of `.binary` values and a byte offset in one `.binary` value remain unambiguous.

18.2 Storage Types

Fixed-width storage types are layout encodings, not separate REXX variable types. A read returns a normal REXX value; a write converts a normal REXX value into the selected storage encoding.

Storage type	Width	REXX value	Notes
`.u8`	1	`.int`	Unsigned byte.
`.i8`	1	`.int`	Signed byte.
`.u16`	2	`.int`	Unsigned little-endian 16-bit integer.
`.i16`	2	`.int`	Signed little-endian 16-bit integer.
`.u32`	4	`.int`	Unsigned little-endian 32-bit integer.
`.i32`	4	`.int`	Signed little-endian 32-bit integer.
`.i64`	8	`.int`	Signed little-endian 64-bit integer.
`.int`	8	`.int`	Convenience spelling for `.i64` in binary memory.
`.f32`	4	`.float`	IEEE binary32 storage.
`.f64`	8	`.float`	IEEE binary64 storage.
`.float`	8	`.float`	Convenience spelling for `.f64` in binary memory.

`.u64` is not part of the Release 1 source surface. Use `.i64`/`.int` or an explicit byte layout until unsigned 64-bit conversion and ordering rules are settled.

Variable-size storage views are:

View	Length argument	Result
`.binary`	Bytes	A copied `.binary` value.
`.string`	Omitted for NUL-terminated fields	A copied `.string` value after validation.
`.decimal`	Omitted for NUL-terminated	A `.decimal` value parsed from decimal text.

For ordinary packed `.string` fields, the starting position is a byte offset to zero-terminated UTF-8 text. The terminator is not part of the string value. Generated lexer/parser code should normally keep source lexemes as binary offset/length pairs and use zero-copy binary compare; materialize strings only for diagnostics or for formats that explicitly store NUL-terminated text.

For `.decimal`, the starting position is a byte offset to zero-terminated decimal text. The terminator is not part of the decimal value. Decimal reads scan to the first zero byte, validate the selected bytes as UTF-8 text, and parse that text through the active decimal plugin.

18.3 Size And Length Intrinsics

`<sizeof..type>` returns the fixed storage width in bytes:

```
layout_size: procedure = .int
  constant NODE_LEFT = 0
  constant NODE_RIGHT = NODE_LEFT + <sizeof..u32>
  constant NODE_BALANCE = NODE_RIGHT + <sizeof..u32>
  constant NODE_SIZE = NODE_BALANCE + <sizeof..i8>
  return NODE_SIZE
```

`<sizeof>` is valid only for fixed-width storage types. It is a compile-time operator.

`<blen>(memory)` returns the logical byte length of a `.binary` variable or `.binary` constant:

```
bytes = <blen>(arena)
header_bytes = <blen>(HEADER_TEMPLATE)
```

18.4 Fixed-Width Reads And Writes

The fixed-width read form is:

```
value = <at..type>(offset) memory
```

`memory` may be a `.binary` variable or `.binary` constant. `offset` is a zero-based byte offset.

The fixed-width write form is:

```
<at..type>(offset) memory = value
```

`memory` must be a mutable `.binary` variable. The field must fit completely inside the current logical byte length. Binary-memory writes do not resize the buffer.

```
options levelb

main: procedure = .int
  constant NODE_LEFT = 0
  constant NODE_RIGHT = 4
  constant NODE_BALANCE = 8

  node = .binary
  call binresize node, 16

  <at..u32>(NODE_LEFT) node = 12
  <at..u32>(NODE_RIGHT) node = 44
  <at..i8>(NODE_BALANCE) node = -1

  left = <at..u32>(NODE_LEFT) node
  balance = <at..i8>(NODE_BALANCE) node
  return 0
```

18.5 Host-Native Packed Numeric Items

Release 1 Level B also provides a deliberately host-native numerical view over the same `.binary` storage:

```
values = .binary
call binresize values, count * <sizeof..float>
<packed..float>(0) values = 100.0
first = <packed..float>(0) values
third = <packed..float>(2) values
```

The only accepted forms are `<packed..int>(index) binary` and `<packed..float>(index) binary`, including use as assignment targets. Indexes are zero-based native-item positions: `2` means the third item. `<blen>` and `binresize` remain byte-based, and stores never grow the binary value.

`packed..int` uses exactly the host representation of `.int` (`rxinteger`), and `packed..float` uses exactly the host representation of `.float` (`rxfloat`). Runtime-owned binary storage is aligned for both native types. Access outside the complete logical byte extent raises `OUT_OF_RANGE` before a write changes the value.

This is a numerical-kernel and native-provider surface, not an interchange format. It performs no endian conversion, width conversion, or portability check. The access spelling supplies the interpretation and `.binary` carries no hidden item-kind tag. Use `<at..type>` for files, protocols, persistence, unaligned data, fixed-width values, or selected byte order.

Only `int` and `float` are valid packed suffixes. Other scalar, object, string, decimal, and encoded-width suffixes are compiler errors. The Level G `rxfnsg` library wraps these instructions with explicit `.packedint` and `.packedfloat` owner classes. They retain zero-based item access through `get` and `set` and store their payload directly in the class object's binary component. Ordinary `.int[]` and `.float[]` arrays keep their current representation in Release 1; automatic packed ordinary arrays and custom `x[index]` indexing remain a post-release Level G roadmap item.

18.6 Variable-Size Reads And Writes

Variable-size reads materialize ordinary Rexx values:

```
key_bytes = <at..binary>(key_offset, key_len) arena
name = <at..string>(name_offset) arena
amount = <at..decimal>(amount_offset) arena
```

The `.binary` length argument is a byte count. The `.string` and `.decimal` forms do not accept a length argument for ordinary packed fields; both are zero-terminated text.

Release 1 implements zero-terminated string and decimal writes and rejects variable-size span writes:

```
<at..binary>(key_offset, key_len) arena = key_bytes
<at..string>(name_offset) arena = name
```

These span-write forms raise `BINARY_MEMORY_SPAN_WRITE_UNSUPPORTED`. Programmers should use `binupdate`, `bincopy`, `binfillat`, `binmakegap`, and `bindrop` for Release 1 byte-span mutation. If span writes are added later, the complete destination span must already fit inside the current binary length. Binary-memory writes do not resize the binary buffer.

18.7 Zero-Terminated UTF-8 Fields

Some packed formats store UTF-8 text as bytes terminated by a zero byte. The program must know which fields use that convention. The terminator is not part of the logical string.

The Release 1 source spelling is `<at..string>(offset) memory`, with no length argument:

```
text = <at..string>(offset) memory
<at..string>(offset) memory = text
amount = <at..decimal>(offset) memory
<at..decimal>(offset) memory = amount
```

The read form scans from the byte offset to the first zero byte, validates the selected bytes as UTF-8, and copies those bytes into a `.string`. The write form writes the string bytes followed by one zero byte. It does not resize the binary buffer; the complete string plus terminator must already fit.

The decimal forms use the same zero-terminated field convention. Reads parse the selected text as `.decimal`. Writes convert the decimal value to decimal text through the active decimal plugin, then write those bytes followed by one zero byte. They do not resize the binary buffer.

The fixed-codepoint read form, `<at..string>(offset, codepoints) memory`, remains available as a specialized extraction form when the program already knows a source span in codepoints. It is not the preferred representation for packed record fields, and Release 1 does not support the matching fixed-codepoint write form.

The compare intrinsic `<compare..string>` uses this zero-terminated field contract.

18.8 Zero-Copy Compare Intrinsics

Use compare intrinsics when the binary memory location itself is part of the operation and copying would be the wrong cost model.

The preferred compare family is:

```
result = <compare..binary>(memory, offset, needle)
result = <compare..string>(memory, offset, string)
result = <compare..u32>(memory, offset, value)
```

The result is an integer:

Result	Meaning
`-1`	The selected memory field is less than the comparison value.
`0`	The selected memory field is equal to the comparison value.
`1`	The selected memory field is greater than the comparison value.

`<compare..binary>(memory, offset, needle)` compares bytes at `memory + offset` with the whole binary needle. The source length is the byte length of `needle`. `memory` and `needle` may each be a variable or a constant.

There is no Release 1 explicit source-length binary compare form. For variable-length fields, compare the stored length with `<blen>(needle)` first, then use `<compare..binary>(memory, offset, needle)`. An explicit-length zero-copy compare can be added later when RXAS has a matching instruction.

`<compare..string>(memory, offset, string)` compares a zero-terminated UTF-8 field in binary memory with a string variable or string constant. It validates the source field as UTF-8 but must not materialize a temporary string when a direct compare is available.

Fixed-width typed compare forms such as `<compare..u32>` compare the stored field with an ordinary Rexx value. They lower through a typed read plus a tri-state comparison, avoiding any binary slice copy.

```
options levelb

main: procedure = .int
  arg arena = .binary, key_offset = .int, key_len = .int, wanted_key =
      .binary, text_offset = .int
```

```
constant MAGIC = "4352584944583031"x as .binary

if <compare..binary>(MAGIC, 0, "43525849"x as .binary) = 0 then say "
    CRXI"
if key_len = <blen>(wanted_key) then
  if <compare..binary>(arena, key_offset, wanted_key) = 0 then say "
     hit"
if <compare..string>(arena, text_offset, "select") = 0 then say "
    keyword"
return 0
```

Ordinary = remains ordinary REXX comparison. For example, this materializes a binary slice before comparing:

```
if <at..binary>(key_offset, key_len) arena = wanted_key then say "
    copied compare"
```

Use that form only when materialization is acceptable or wanted.

18.9 Constants And Scope

Layout constants should be compile-time constants declared once in an explicit procedure scope. Free-floating top-level `constant` declarations are not part of the Release 1 surface; because `constant` is an instruction form, that shape can create an implicit `main()` rather than a pure module surface.

Use a declaration procedure with `procedure expose` to give related constants a single home. A private declaration procedure may be used to give the constants an explicit home without making that procedure part of the public API. Release 1 generated-code examples should use this pattern so layout values are not redeclared in every helper procedure.

Release 1 constants are source-module local. A declaration procedure's `procedure expose` list makes the constants available to other procedures in the same source module, but constants are not imported across source, RXAS, or RXBIN module boundaries. Cross-module constants and wildcard expose forms such as `TINY_TOK_*` are Release 2 ergonomics/design items, not Release 1 syntax.

If the same file also has executable script code, put that code in an explicit `main: procedure`. A declaration procedure is a real procedure boundary; later statements belong to it until the next procedure/class boundary, not to a fresh implicit `main()`.

```
options levelb
namespace packedindex expose find_node

packedindex_layout: procedure expose NODE_LEFT NODE_RIGHT NODE_SIZE
    EMPTY_NODE
  constant NODE_LEFT = 0
  constant NODE_RIGHT = 4
  constant NODE_SIZE = 32
  constant EMPTY_NODE = "00000000000000000000000000000000"x as .binary
  return

find_node: procedure = .int
  arg arena = .binary, key = .binary
  if <compare..binary>(arena, NODE_LEFT, key) = 0 then return NODE_LEFT
  return -1
```

Do not put executable setup or `constant` declarations at top level in a library-style module. Top-level executable statements before a procedure synthesize an implicit `main()` for script compatibility. Shared binary layout values should be constants in an explicit procedure scope, exposed to the procedures in that same source module that need them.

18.10 Ordinary Helper Functions

Intrinsics are for direct binary-memory access, constants, and zero-copy compare. Other buffer management should use ordinary functions from `rxfnsb`. The existing `BIN*` helpers are compatibility helpers that return copied values and use one-based positions. Packed-memory helpers use zero-based offsets, mutate exposed arguments, and may be direct-lowered by the compiler/inliner.

The Release 1 packed-memory helper surface is:

Helper	Effect
`BINRESIZE(data, length)`	Set `data`'s logical length to `length` bytes; growth zero-fills.
`BINCLEAR(data)`	Set `data` to the empty binary value.
`BINFILL(data, byte)`	Fill the whole current logical byte range.
`BINFILLAT(data, offset, length, byte)`	Fill a zero-based byte span.
`BINCOPY(dst, dst_offset, src, src_offset, length)`	Copy `length` bytes between different binary values.
`BINMEMMOVE(data, dst_offset, src_offset, length)`	Move `length` bytes within one binary value; overlapping ranges are safe.

Helper	Effect
`BINAPPEND(dst, src)`	Append all bytes from `src` to `dst`.
`BINUPDATE(dst, offset, src)`	Overlay all bytes from `src` into `dst` at `offset`.
`BINMAKEGAP(data, offset, length)`	Insert a zero-filled gap.
`BINDROP(data, offset, length)`	Delete a byte range.

Mutating helpers update the first binary argument through `arg expose` and return the new logical byte length unless the specific helper documentation says otherwise. The VM may keep a larger private physical capacity for a binary register, growing in blocks and reusing that capacity across shrink/grow cycles; programs can observe only the logical length through `<blen>/BINLENGTH`.

18.11 Diagnostics And Signals

Compiler diagnostics use stable catalog keys. Tests should normally assert raw diagnostic keys with `CREXX_DIAGNOSTICS=raw`, while user output is localized through `messages/diagnostics.*.msg`.

Existing keys that must continue to be used:

Key	English template	Required use
`BINARY_MEMORY_AT_REQUIRED`	Binary memory access requires an at intrinsic.	Binary memory access was parsed without an `at` head where one is required.
`BINARY_MEMORY_FIXED_FOR_NOT_ALLOWED`	Fixed-width binary memory access must not specify a length argument.	Fixed-width access has an illegal length argument.
`BINARY_MEMORY_INVALID_STORAGE_TYPE`	Binary memory access has an unsupported storage type.	The dotted storage type is not supported.
`BINARY_MEMORY_LENGTH_NOT_ALLOWED`	This binary memory access form does not accept a length argument.	A zero-terminated form such as `.decimal` was given a length argument.
`BINARY_MEMORY_LENGTH_REQUIRED`	Variable-length binary memory access requires a length argument.	A `.binary` variable-size form is missing its length argument.
`BINARY_MEMORY_OFFSET_REQUIRED`	Binary memory access requires a byte-position argument.	A binary-memory form is missing its byte offset.

Key	English template	Required use
`BINARY_MEMORY_READ_ONLY`	Binary memory access target is read-only.	A write targets a binary constant or other read-only storage.
`BINARY_MEMORY_SPAN_WRITE_-UNSUPPORTED`	Variable-length binary memory writes are not supported yet.	A variable-size write is not implemented in this compiler slice.
`BINARY_MEMORY_TARGET_NOT_-BINARY`	Binary memory access target must be a scalar .binary value.	The memory operand is not a scalar `.binary` value or constant.
`CANNOT_CAST_BINARY`	Binary data cannot be converted to text without an explicit encoding.	Binary bytes would become text without explicit valid UTF-8 conversion.
`INVALID_SIZEOF_SYNTAX`	SIZEOF intrinsic has invalid syntax.	`<sizeof>` is malformed.
`INTRINSIC_GENERIC_TYPES_-UNSUPPORTED`	Intrinsic type parameter lists are not supported in Release 1.	A parsed generic intrinsic type-list form is valid syntax but unsupported in Release 1.
`PACKED_INDEX_REQUIRED`	Packed numeric access requires a zero-based item-index argument.	A packed native item read or write omits its item index.
`PACKED_INVALID_STORAGE_TYPE`	Packed numeric access supports only native int and float items.	A `packed` intrinsic names any other suffix.

Compare diagnostics are:

Key	Required message template
`BINARY_MEMORY_COMPARE_ARGUMENTS`	Binary memory compare intrinsic has an invalid argument list.
`BINARY_MEMORY_COMPARE_TYPE`	Binary memory compare intrinsic has an unsupported comparison type.
`BINARY_MEMORY_COMPARE_NEEDLE_TYPE`	Binary memory compare needle has an incompatible type.

Runtime operations signal:

Signal	Required use
`OUT_OF_RANGE`	Negative offset, negative length, field does not fit, missing NUL terminator, or value outside selected field range.
`UNICODE_ERROR`	Bytes selected for a `.string` value or string compare are not valid UTF-8.
`CONVERSION_ERROR`	Decimal or numeric conversion from selected bytes fails.
`OVERFLOW_UNDERFLOW`	A Rexx numeric value cannot be represented in the selected binary field or reverse conversion overflows.
`FAILURE`	Allocation or VM failure during resize, append, or other buffer-changing operations.

18.12 Deferred Items

The Release 1 surface deliberately leaves a few items for later work:

- unsigned 64-bit storage syntax and ordering rules (`.u64`);
- variable-size span writes for `.binary` and fixed-codepoint `.string` fields;
- explicit source-length zero-copy binary compare;
- wildcard exposes for constant families and generated-layout conveniences;
- direct compiler or inliner lowering for selected packed-memory helpers when profiling proves the helper call overhead matters;
- read-only constant/view parameter passing, binary struct declarations, and mmap/shared-memory views.

18.13 Implementation Acceptance Tests

The implementation test surface covers:

- positive parsing and lowering for every intrinsic in this chapter;
- `.binary` constants and variables using the same read and compare forms;
- read-only writes to constants;
- each diagnostic key listed above, using raw diagnostic output;
- all runtime signals listed above;

- optimized RXAS for lookup examples, proving compare intrinsics do not emit `binsubstr`, binary slice materialization, string extraction, or helper calls in the hot path.

19

Literals

A literal is a fixed value that appears directly in a program's source code. It can be a string of characters, a numeric value, or a Boolean value. Literals are used to represent constant data that does not change during the execution of a program.

19.1 String Literals

A string literal is a sequence of characters enclosed in single or double quotes. It represents a constant value in a program.

19.2 Numeric Literals

Numeric literals can be integers, floating-point numbers, or hexadecimal numbers. Integers are whole numbers that do not have a decimal point. Floating-point numbers are numbers that have a decimal point.

Integer literals may also use a `0x` prefix for hexadecimal notation, for example `0x10000`. Hexadecimal integer literals are `.int` values and are intended for masks, flags, and low-level numeric constants.

Release 1 `.int` literals use the signed 64-bit range `-9223372036854775808..9223372036854775807`. The leading sign is an operator in the source grammar, but the compiler recognizes `-9223372036854775808` as the

single legal minimum-value edge. Decimal literals beyond either bound are reported as `OVERFLOW_UNDERFLOW` rather than being truncated to a host integer.

19.3 Named Constants

Level B supports named compile-time constants:

```
flags: procedure = .int
  constant FLAG_STRING = 0x00010000
  constant FLAG_TEXT = 0x00200000
  constant TEXT_FLAGS = FLAG_STRING + FLAG_TEXT
  constant SAMPLE_BYTES = "4142"x as .binary
  return TEXT_FLAGS
```

Constants must be declared inside an explicit procedure, method, or factory scope. A file-body `constant` declaration is not part of the Release 1 surface.

The initializer must be a compile-time constant expression. A named constant is immutable, can be used in ordinary expressions in its visible scope, and can be used where inline assembler expects a literal operand. Runtime payload constants such as strings, decimals, floats, and binary byte sequences are stored through the RXBIN constant pool. Object, reference, and array constants are not part of this Level B surface.

Assigning to a named constant after declaration is a compile-time error. A constant declaration also cannot reuse an already declared variable name in the same scope.

19.4 Hex/Binary Literals

A string literal that concludes with `x`, such as `"4142"x`, is decoded as hexadecimal bytes. A string literal that concludes with `b`, such as `"0100000101000010"b`, is decoded as binary bits. Hex literals use two hex digits per byte. Binary literals use eight bits per byte.

Whitespace inside a hex or binary literal is ignored, so generated byte tables may group digits for readability:

```
payload = "00 ff 41"x as .binary
bits = "0100 0001 0100 0010"b
```

Adjacent string literal tokens separated only by a physical line break and ordinary indentation are treated as one literal. The line break contributes no character; include a trailing or leading blank inside a chunk when a blank is part of the value:

```
message = "large "
          "text block"
```

For suffixed hex and binary literals, the chunks are joined before decoding. Only the final chunk may carry the `x` or `b` suffix:

```
payload = "00 ff "
          "41"x as .binary

bits = "0100"
       "0001"b
```

A suffix on an earlier chunk is rejected. Same-line adjacent literals keep the ordinary Rexx whitespace-concatenation meaning; this continuation rule applies only across a physical line break.

If the decoded bytes are valid UTF-8, the literal is text by default:

```
text = "4142"x         /* "AB" */
also = "01000001"b     /* "A" */
```

If the decoded bytes are not valid UTF-8, the literal must be used in an explicit `.binary` context:

```
payload = .binary
payload = "00ff41"x

other = "ffff"x as .binary
```

A first untyped assignment such as `payload = "ffff"x` is rejected because the compiler treats the first binding as a text context unless `.binary` is made explicit. The same is true for `payload = "ffff"x as .string`.

Converting ordinary text to `.binary` stores the string's UTF-8 bytes exactly, without normalization:

```
bytes = "□" as .binary     /* ce b1 */
```

Converting binary bytes back to text requires an explicit `.string` cast and valid UTF-8:

```
text = ("4142"x as .binary) as .string     /* "AB" */
```

If the bytes are not valid UTF-8, a constant cast is rejected by the compiler and a runtime binary-to-string cast raises `UNICODE_ERROR`.

20 Variables

Level B variables are typed. The compiler can infer many local variable types, but once a variable has a type, later assignments must be compatible with that type.

20.1 Keywords

Keywords, instruction names, and operators cannot be used as variable names.

20.2 Declaration by Assignment

A variable is often declared by its first assignment:

```
count = 0
price = 1.25
name = "Ada"
ready = .boolean(1)
```

The inferred types are based on the assigned expression. Use constructors when the intended type needs to be explicit:

```
count = .int(0)
name = .string("Ada")
ratio = .float(1.25)
money = .decimal("1.25")
payload = .binary()
```

The canonical integer type name is `.int`.

20.3 The System Variable `rc`

`rc` has the integer type `.int`, including with `options numeric_classic`. The ADDRESS statement stores the command's integer status in it. It can also hold an integer result assigned by the program, but it is not a general-purpose string result variable.

Use a separate variable for a function that returns text:

```
result_text = MyFunc()
say result_text
```

Assigning nonnumeric text to `rc` requires an invalid string-to-integer conversion. The compiler reports `BAD_CONVERSION` when it can establish the invalid value during compilation; a value obtained at runtime raises `CONVERSION_ERROR`. The same conversion rules apply to an ordinary `.int` variable. Declaring `rc = .string` does not change its system type.

20.4 Block Scope

A first assignment inside a `DO ... END` block creates a binding in that block unless the variable already exists in an enclosing scope. Sibling blocks do not share newly created bindings. This differs from Classic Rexx's procedure scope and matters when computing a result in alternative branches.

Declare the shared result before the conditional:

```
choose: procedure = .string
  arg flag = .string
  text = .string
  if flag = "1" then do
    text = "from-if"
  end
  else do
    text = "from-else"
  end
  return text
```

Without `text = .string`, the assignments and final read create separate bindings. The compiler warns `NOT_IN_SAME_SCOPE`, and the final read returns the uninitialized variable's name instead of either branch's string. A bare type declaration is sufficient; no dummy initial value is needed.

20.5 Arrays

Array variables are declared with a typed array expression:

```
args = .string[]
scores = .int[10]
grid = .int[10, 10]
```

These bare type expressions declare storage; they do not create and assign a fresh empty array whenever execution reaches the declaration. In particular, repeating `args = .string[]` in a procedure that exposes `args` does not clear its existing module-global elements. With `import rxfnsb`, use `call arraydrop args` when the program needs to empty an existing array.

Array arguments use the same notation in procedure signatures:

```
main: procedure = .int
  arg args = .string[]
```

20.6 Object Variables

Class and interface values are object-shaped. Factories use dotted class or interface names:

```
asset = .asset("log.txt")
box = .box()
```

When a class implements an interface, a class instance can be assigned to that interface contract. Use `expr is .type`, `expr as .type`, and `typeof(expr)` when code needs runtime type checks or concrete type information.

20.7 Globals and Expose

Top-level values in a namespace are global to that module. Exposed globals can be imported by other modules. Procedures have their own local scope unless state is deliberately exposed through the current Level B expose mechanisms.

Prefer explicit arguments and return values for ordinary application code. Use global exposed state for library constants, runtime integration points, and cases where shared module state is genuinely the simplest contract.

21 Operators

21.1 Expression Operators

Level B incorporates the Classic Rexx operators, which function similarly to their counterparts in other programming languages. However, as a language with a strong typing system, crexx Level B introduces rules for type promotion that differ significantly from Classic Rexx's string-based model.

Note: Influence of `OPTIONS` and `NUMERIC`

The precise behaviour of operators is governed by two key instructions:

- **`OPTIONS`**: A file-wide instruction that sets **parser** rules. It determines operator precedence, associativity, and which symbols (e.g., `%`, `//`) are used for certain operations.
- **`NUMERIC`**: A procedure-scoped instruction that controls the **semantic** behaviour of arithmetic, such as precision, rounding, and the `FUZZ` setting for comparisons.

21.2 Arithmetic operators

TABLE 2: Arithmetic Operators

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide (normal or integer, see note)

Operator	Description
`%`	Integer Division or Remainder (see note)
`//`	Remainder (see note)
`**`	Raise a number to a whole-number power (exponentiation)
Prefix -	Negate the next term
Prefix +	Take the next term as-is

21.2.1 Influence of `OPTIONS` on Arithmetic

The `OPTIONS {NUMERIC_CLASSIC|NUMERIC_COMMON}` setting determines which symbols are used for certain operations for the entire source file.

- Under **`OPTIONS NUMERIC_CLASSIC`**:
  - **`%`** performs **integer division**.
  - **`//`** performs the **remainder** operation.
- Under **`OPTIONS NUMERIC_COMMON`**:
  - **`%`** performs the **remainder** operation.
  - **`/`** performs **integer division** if both operands are integers; otherwise, it performs normal division.
  - **`//`** is not strictly a valid operator, however, to support legacy code, it is treated as an alias for `%`.

21.3 Comparison operators

21.3.1 Strict Operators renamed to String Comparison Operators

CREXX Level B refers to what other REXX dialects call strict comparison (e.g., ==, ») as string comparison.

This terminology is used intentionally to make the operator's semantics explicit within a typed language. The core principle of a string comparison is that it unconditionally promotes both of its operands to the `.string` data type before performing a direct, character-for-character comparison. This ensures that the operation is always a test of exact textual identity, with no possibility of numeric interpretation or padding, which clarifies its behaviour and purpose.

21.3.2 Comparison Operator Categories

CREXX provides comparison operators that fall into two main categories: **loose** and **string**.

- **Loose** operators (like =) perform padding to make strings of unequal length comparable and can be numeric-aware.
- **String** operators (like ==) require strings to be identical in length and always operate on the string representation of terms.

The complete set of equality operators is summarised below:

TABLE 3: Comparison Operator Categories

Operator	Type	Padding?	Description
=	Loose	**Yes**	Standard REXX equality (numeric or padded string).
==	**String**	No	**String** character-for-character equality.

21.3.3 Standard Comparison (Case-Sensitive, Loose)

These operators perform a numeric comparison only when both operands are numeric by value after any required type conversion. If either operand is not numeric, they fall back to a character comparison where the shorter string is padded with blanks. For example, `"09" = 9` is true numerically, while `"a" > 1` is a padded string comparison and does not raise a numeric conversion error.

- The numeric comparison is affected by the **`NUMERIC FUZZ`** setting.

cRexx Level B Unicode String Comparison

To ensure accuracy with Unicode, CREXX Level B enhances the standard string comparison. Before the padding and comparison steps, both strings are first brought into a consistent representation by applying Unicode Normalization Form C (NFC).

TABLE 4: Standard Comparison Operators

Operator(s)	Description
=	Equal (numerically or when padded)
¬=, \=, /=	Not equal (inverse of =)
>	Greater than
<	Less than

Operator(s)	Description
< >	Not equal (same as ¬=)
>=	Greater than or equal
¬<	Not less than
<=	Less than or equal
¬>	Not greater than

21.3.4 String Comparison (Case-Sensitive) a.k.a. Strict Comparison

These operators perform what is known in other rexx dialects as a **strict** comparison. CREXX refers to them as **string comparisons** to emphasize their semantics: they *always* operate on the string representation of the terms.

Before the comparison, both operands are **promoted to the `.string` type**. This forces a character-by-character comparison with no padding.

The compiler applies this promotion before both optimization and code emission, so folded and non-folded expressions follow the same string-comparison rules. Numeric literals are compared by their numeric value after stringification, not by their source spelling, so `01 == 1` is true because both sides become the string `"1"`.

- **No Padding**: If the strings have different lengths, they are not equal.
- **No Numeric Interpretation**: `1.0 == 1` may be **false** because the operands are first converted to the
- strings `'1.0'` and `'1'`, which are not identical. Numeric types are converted to their string representation before comparison,
- using the current `NUMERIC FORM`, `NUMERIC CASE`, and `NUMERIC DIGITS` settings.

TABLE 5: String Comparison Operators

Operator(s)	Description
==	String equal (identical)
¬==, \==, /==	String not equal (inverse of ==)
>>	String greater than
/>>	String not greater than
<<	String less than
\<<	String not less than
>>=	String greater than or equal
<<=	String less than or equal

21.4 String Concatenation

TABLE 6: String Concatenation Operators

Operator	Description
`||`	Concatenate terms (with no blank between them)
space ()	Concatenate with a single space added between terms
abuttal	Concatenate without a space (i.e., no space between a literal and variable)

21.5 Logical operators

TABLE 7: Logical Operators

Operator	Description
`&`	AND (returns `1` if both terms are true)
`|`	Inclusive OR (returns `1` if either term is true)
`&&`	Exclusive OR (returns `1` if either term is true, but not both)
Prefix `¬`	Logical NOT (negates; `1` becomes `0` and vice versa)

21.6 Level B named integer operators

Level B also provides a fixed set of system-programmer named operators for integer bit and flag work. The syntax is a single contiguous `<id>` token; blanks or comments are not allowed inside the angle brackets. Operator names are case-insensitive.

These operators are Level B syntax and do not change Classic REXX / Level C operator semantics.

Operator	Description
`<and>`	Bitwise integer AND
`<or>`	Bitwise integer inclusive OR
`<xor>`	Bitwise integer exclusive OR
Prefix `<not>`	Bitwise integer NOT
`<idiv>`	Integer division, independent of numeric mode operator-token rules
`<mod>`	Remainder, independent of numeric mode operator-token rules
`<rem>`	Alias for `<mod>`

Operator	Description
`<shl>`	Integer shift left
`<shr>`	Integer shift right
`<has>`	Test whether any masked bit is present
`<set>`	Set masked bits, equivalent to `<or>`
`<clear>`	Clear masked bits, equivalent to `<and> <not>`

Examples:

```
flags = flags <set> RV_FLAG_INT
if flags <has> RV_FLAG_TEXT then say "text"
flags = flags <clear> RV_FLAG_BINARY
mask = 1 <shl> bit
```

Table: Level B Named Integer Operators

21.7 Term Operators

TABLE 8: Term Operators

Operator	Description
`()`	Parenthetical grouping
`[]` or `.`	Stem / Array index

21.8 Reference Expressions

Level B reference operations use word-form expressions:

Form	Description
`reference target`	Create a weak reference to aliasable storage.
`local = dereference ref`	Link a current-scope local variable to the current reference target until scope exit.
`snapshot ref`	Make an explicit deep copy of the current reference target.
`<refvalid>(ref)`	Return whether the weak reference currently has a valid target.
`left <refsame> right`	Return whether two compatible references retain the same storage-identity cell, without dereferencing them.

The operand of `reference` must be storage such as a local, exposed argument or global, method `self`, an object/array attribute, or an array element. It must not

be a temporary expression. `dereference` and `snapshot` raise `REFERENCE_INVALID` if the target has expired.

`<refsame>` compares reference identity, not referenced values. Two copies of the same reference remain identical after their target expires; two cleared or empty references compare false because neither retains an identity. Use `<refvalid>` separately when live-target validity matters. Ordinary = and == do not accept reference operands.

`dereference` must be used as the right side of an assignment to a local variable in the current procedure or block scope. It cannot assign through an attribute, array element, exposed global, argument, or expression. Use `snapshot ref` when a value copy is needed.

Level B does not implicitly treat a reference value as the referenced target for member calls or index operations. Code must write `local = dereference ref` for a scoped live link, `snapshot ref` for a deep copy, or pass a `reference .T` value to code that expects a reference. Convenience forms such as `itemsRef[i]`, `itemsRef[i] = value`, and `listRef.add(value)` are reserved for possible Level G live-access syntax.

21.9 Level G Object Equivalence

Level G provides `left <eq> right` as opt-in sugar for canonical object equivalence. Both operands must be object values and the static class of the left operand must implement `data.ObjectEquatable`:

```
ObjectEquatable: interface
  equivalent: method = .boolean
    arg other = .object
```

The compiler evaluates each operand once and lowers the expression to the equivalent of `left.equivalent(right as .object)`. The implementation defines the supported object pairs and should provide a reflexive, symmetric, and transitive relation. The operator name is case-insensitive and has comparison precedence.

`<eq>` does not overload = or ==, does not compare storage identity, and does not invoke an `ObjectKeyStrategy`. Use `<refsame>` only for explicit reference-cell identity and use a key strategy when a collection needs policy-specific equivalence and hashing.

21.10 Operator Precedence

The order of priority of the operators (from highest to lowest). Note that the `OPTIONS` instruction affects the precedence and associativity of certain operators.

TABLE 9: Operator Priorities

Priority	Operators	Description	Notes
1	`() [] .`	Term operators (grouping, indexing)	
2	Prefix `+ - ¬ <not>`	Unary operators	[9]
3	`**`	Exponentiation	[10] [11]
4	`* / % // <idiv> <mod> <rem>`	Multiply and divide	[12]
5	`+ -`	Add and subtract	
6	`<shl> <shr>`	Level B named integer shifts	
7	`<and> <has> <clear>`	Level B named integer AND/flag-test	
8	`<xor>`	Level B named integer XOR	
9	`<or> <set>`	Level B named integer OR/flag-set	
10	`||` (space, abuttal)	Concatenation	
11	`= > < ==... <refsame> <eq>`	All comparison operators (`<eq>` is Level G)	
12	`&`	Logical AND	
13	`| &&`	Logical OR and exclusive OR	

[9] **`OPTIONS NUMERIC_CLASSIC`**: Prefix - has higher priority than **.

[10] **`NUMERIC_CLASSIC`**: Left-associative and has lower priority than Prefix -.

[11] **`NUMERIC_COMMON`**: Right-associative and has higher priority than Prefix -.

[12] Which operators are valid depends on `OPTIONS`.

22

Classes and Interfaces

Level B implements the core class/interface model:

- interfaces and classes
- classes implementing one or more interfaces
- abstract interface methods
- final/default interface methods
- interface-centred default and named factories
- factory provider selection through class-side `match`
- checked casts with `expr as .type`
- boolean type tests with `expr is .type`
- concrete type introspection with `typeof(expr)`
- namespace-qualified contract references such as `.pkg..thing()`

Level B does **not** currently implement interface inheritance, interface attributes/state, interface factory bodies, overloads, singleton declarations, class/interface constants, or destructor/finalizer syntax.

Class and interface constants are a Release 2 roadmap item. The expected starting point is private constants owned by the class or interface body, consistent with the current member privacy model. Whether such constants also need a controlled public view should be investigated as part of that Release 2 design.

22.1 Core Model

Interfaces define callable contracts. Classes implement those contracts. Factory declarations do not carry an explicit return type. The return contract is implied by the owner: an interface factory returns that interface contract, and a class factory returns that concrete class. In a class factory, bare `return` returns the constructed object; there is no source-level `self` value to return explicitly.

```
vehicle: interface
  *: factory
  arg name = .string
  describe: method = .string

car: class implements .vehicle
  _name = .string

  *: factory
    arg name = .string
    _name = name
    return

  describe: method = .string
    return "car:" || _name
```

Instances are normally created through the interface:

```
current = .vehicle("mini")
say current.describe()
```

Each class also has an intrinsic interface of its own name, so `.car()` remains valid when you want to work directly with the concrete class contract.

22.2 Defining Interfaces

An interface may declare:

- abstract methods
- final/default methods with bodies
- the default `*` factory
- named factories

An interface method with a body is final in Level B. A class may rely on that body, but it may not override it.

```
shape: interface
  *: factory

  describe: method = .string
    return prefix() || ":" || summary()

  prefix: method = .string
    return "iface"

  summary: method = .string
```

In this example `describe` and `prefix` are final/default methods, while `summary` remains abstract.

22.3 Defining Classes

Classes implement one or more interfaces:

```
box: class implements .shape
  *: factory
    return

  summary: method = .string
    return "box"
```

The class must implement every abstract member from its effective interface set. If an interface already provides a final/default method body, the class may omit that member.

Factories are the exception to the ordinary `name: callable = .type` spelling: write `*: factory` or `name: factory`, then declare arguments with `arg` as usual. A factory return type written after `=` is not part of the Level B source syntax.

Assignment compatibility in the current Level B implementation flows from a concrete class value to an implemented interface. In practice that means a statement such as `iface = .widget(...)` is supported, while interface-to-interface assignment should not be relied on.

The factory call brackets are semantically significant. `.widget()` calls the default factory and returns an initialized object. Bare `.widget` is the typed default object value for that class and is not initialized. It can still satisfy type-contract checks such as `value is .object`, but using it as a receiver raises the catchable `OBJECT_NOT_INITIALIZED` signal. Use `initialized(value)` when code needs to distinguish

these states explicitly.

22.4 Class State

Class attributes are declared in the class block:

```
box: class implements .shape
  _label = .string
  _count = .int
```

For ordinary classes, let the compiler allocate storage automatically and keep external callers on factories and methods.

Explicit physical layout should be reserved for genuine low-level interop. When that is needed, append `with register.N` to the attribute definition, where `N` is the one-based VM object-attribute slot:

```
raw_event: class
  _code = .int with register.1.int
  _module = .int with register.2.int
  _address = .int with register.3.int
  _name = .string with register.4.string
```

The optional suffix after the index is the VM register value view to use for the slot. Valid views are `.int`, `.float`, `.decimal`, `.binary`, `.string`, and `.object`:

```
  _message = .string with register.5.string
  _payload = .object with register.5.object
```

System classes can also expose masked status-flag views using `register.N.flags.<partition>`. These views are `.int` attributes over the VM status word rather than value-payload views:

```
  _cache_flags = .int with register.0.flags.library
  _vm_flags = .int with register.0.flags.vm
```

The supported partitions are

- `.vm`,
- `.compiler`,
- `.library`,
- `.user`,
- `.public`, and

- `.readable`.

`.vm`, `.compiler`, and `.readable` are read-only. `.library` and `.user` are writable. `.public` is writable but covers only the library and user bands, not compiler call-ABI flags. Assigning a writable flag view replaces only that masked band; other status-word bits are preserved. Flag views must be declared as `.int`; unknown partitions are rejected during source validation.

The compiler emits the attribute linking code for methods that read or write these attributes. Source code should still access them through methods, not through hand-written assembler. It is valid for VM-integration classes to define more than one typed view over the same physical slot, as shown for a signal payload/message slot above. Ordinary application classes should not use explicit register mappings unless they are matching a fixed VM or native object layout.

`register.0` is reserved for a typed view of the containing value itself. This is a Rexx source-level convention, not RXAS attribute zero. The compiler lowers it to a direct receiver/factory link, while `register.1` and above continue to name one-based child attributes. `register.0` and duplicate typed mappings to the same `register.N` slot are treated as complex attributes: compiler-generated reads copy the selected view into a local register before expression code manipulates it, and writes copy the selected payload view back through the physical slot. Status/cache flag updates for these typed views are explicit runtime code, not hidden compiler side effects. This is a low-level system-programmer facility for runtime and VM-integration classes; ordinary programs should use normal class attributes and methods.

Flag views are the exception to the complex typed-view copy rule: reads access the status word directly and do not copy the register's string, binary, numeric, or object payload. Writes through writable flag views replace only the selected flag partition.

22.5 Receiver Storage

Inside a method, unqualified attribute and method names are resolved against the current receiver where the member exists. For example, `call clear()` calls the receiver's `clear` method, and `rc = append(value)` calls the receiver's `append`

method and stores its result.

`self` names the receiver storage explicitly. It is mainly useful when a method needs to pass or return a live reference to its receiver:

```
iterator: method = .StringIterator
  return .StringArrayListIterator(reference self)
```

Bare `self` is not an implicit reference. Use `reference self` when a formal requires `reference .Class`, and use ordinary unqualified attribute access or method calls for normal receiver access.

The standard class-library collection direction uses that explicit receiver reference for live iterators: `iterator()` returns an unsynchronized live iterator, and `snapshotIterator()` names the explicit factory-time snapshot variant.

22.6 Factory Selection with `match`

Factory members are declared on the interface and implemented by the class using the same name.

- `*` is the default factory member
- `name: factory` is a named factory member
- `name: match` is the optional class-side selector for that same factory

`match` is class-side only. Its signature must match the paired factory and it must return `.int`.

Selection rules:

1. Every candidate provider is scored through its effective `match`, even when there is only one candidate.
2. If `match` is omitted, the candidate behaves as if it returned `1`.
3. Scores `<= 0` reject that candidate.
4. The highest positive score wins.
5. Ties are broken alphabetically by concrete class name.

22.6.1 Example

```
asset: interface
  *: factory
  arg spec = .string
  from_size: factory
  arg size = .int

  describe: method = .string
    return kind() || ":" || name() || ":" || size()

  kind: method = .string
  name: method = .string
  size: method = .int

fileasset: class implements .asset
  _name = .string
  _size = .int

  *: match
    arg spec = .string
    if spec = "log.txt" then return 100
    return 0

  *: factory
    arg spec = .string
    _name = spec
    _size = 8
    return

  from_size: match
    arg size = .int
    if size = 8 then return 50
    return 0

  from_size: factory
    arg size = .int
    _name = "sized-file"
    _size = size
    return

  kind: method = .string
    return "file"

  name: method = .string
    return _name

  size: method = .int
    return _size

cacheasset: class implements .asset
```

```
  _name = .string
  _size = .int

  *: factory
    arg spec = .string
    _name = spec
    _size = 1
    return

  from_size: factory
    arg size = .int
    _name = "cache-" || size
    _size = size
    return

  kind: method = .string
    return "cache"

  name: method = .string
    return _name

  size: method = .int
    return _size

selected = .asset("log.txt")
fallback = .asset("memo")
say selected.describe()   /* file:log.txt:8 */
say fallback.describe()   /* cache:memo:1 */
```

This complete example is mirrored by the test `compiler/tests/rexx_src/interface_showcase_same_module.crexx`.

22.7 Multiple Interfaces

A class may implement more than one interface as long as the public member surface remains coherent.

```
named: interface
  *: factory
  arg label = .string, length = .int
  name: method = .string

measured: interface
  size: method = .int

widget: class implements .named .measured
  _label = .string
  _length = .int
```

```
  *: factory
    _label = label
    _length = length
    return

  name: method = .string
    return _label

  size: method = .int
    return _length

item = .widget("gear", 4)
by_name = item
by_size = item
say by_name.name()
say by_size.size()
```

This example is mirrored by the test `compiler/tests/rexx_src/interface_multi_-interface_same_module.crexx`.

22.8 Namespace Qualification

If two imported namespaces expose contracts with the same name, qualify the reference with `namespace..`

```
import qifa
import qifb

left = .qifa..vehicle("one")
right = .qifb..vehicle.from_name("two")
```

The token to the left of `..` must be a namespace name that has already been imported. Qualification does not bypass `import`. `namespace::symbol` remains accepted as a compatibility alias, but `namespace..symbol` is the canonical form.

22.9 Same-File Contract and Provider Pattern

An interface and one or more implementation classes may live in the same source file and namespace. This is often the clearest shape when the contract and providers are developed together.

```
options levelb
namespace automotive expose vehicle mycar

vehicle: interface
  *: factory
  arg name = .string
  describe: method = .string

mycar: class implements .vehicle
  _name = .string

  *: factory
  arg name = .string
  _name = name
  return

  describe: method = .string
  return "dep:" || _name
```

The interface owns the public factory contract. The class supplies the implementation with the same factory name and argument signature; its concrete class return is accepted because the class implements the interface.

This same-file shape is mirrored by `compiler/tests/rexx_src/interface_dep_contract.crexx`.

22.10 Split-File Contract and Provider Pattern

When an interface and its provider class live in different source files, each file is still compiled as a separate module. Multiple files may contribute symbols to the same namespace, just as the Level B standard library contributes many `rxfnsb` functions from separate files, but the provider compile must still be able to find the interface contract through the import path.

The provider may use the same namespace as the contract:

```
/* vehicle.rexx */
options levelb
namespace automotive expose vehicle

vehicle: interface
  *: factory
  arg name = .string
  describe: method = .string
```

```
/* mycar.rexx */
options levelb
namespace automotive expose mycar

mycar: class implements .vehicle
  _name = .string

  *: factory
  arg name = .string
  _name = name
  return

  describe: method = .string
  return "dep:" || _name
```

It is also valid to put provider classes in a separate provider namespace when that better matches packaging or ownership:

```
options levelb
namespace automotive_provider expose mycar
import automotive

mycar: class implements .vehicle
  /* implementation as above */
```

Pure contract modules are valid: a source file that contains only interface or class/interface metadata can compile and assemble into a metadata-only `.rxbin`. A provider that consumes that binary contract must compile with the contract's directory on the binary import path, for example `rxc -i build-dir provider.rexx`. Keeping source roots and binary roots separate prevents stale `.rxbin` side products beside edited source from silently satisfying interface imports.

The class factory must have the same argument signature as the interface factory. The return contract is implicit on both sides: the interface factory returns the interface, and the class factory returns the concrete class. The compiler checks that the concrete class value is assignable to the interface return type.

If a provider reports `#INTERFACE_NOT_FOUND` or `#INTERFACE_MEMBER_SIGNATURE_MISMATCH`, check these first:

- the provider compile can find the source module (`.crexx`, `.crx`, `.rexx`, or the initial file's arbitrary extension in a source root) or binary module (`.rxbin` in a directory passed with `-i`) that exposes the interface
- the class factory argument list exactly matches the interface factory argu-

ment list
- a binary contract is intentional; sibling `.rxbin` files are not searched unless their directory is a binary root
- the provider module is loaded or linked into any program that calls the interface factory

This example is mirrored by:

- `compiler/tests/rexx_src/qualified_interface_main.crexx`
- `compiler/tests/rexx_src/qualified_interface_dep_a.crexx`
- `compiler/tests/rexx_src/qualified_interface_dep_b.crexx`

22.11 Casts and Type Tests

Level B supports explicit object casts and runtime type inspection:

```
vehicle = .car("roadster") as .vehicle
current = vehicle as .car

say typeof(current)
say current is .vehicle
say current is .car
say current is .truck
```

Rules:

- `expr as .interface` succeeds when the concrete class implements that interface
- `expr as .class` succeeds only for that exact concrete class
- failed object casts raise `CONVERSION_ERROR`
- `expr is .interface` checks interface implementation
- `expr is .class` checks the exact concrete class
- `typeof(expr)` returns the concrete runtime class for objects and the canonical built-in type name for scalars

These operations are mirrored by:

- `compiler/tests/rexx_src/type_ops_showcase.crexx`
- `compiler/tests/rexx_src/type_ops_fail.crexx`

22.12 Object Equivalence and Key Strategies

The class library separates canonical object equivalence from collection key policy:

- `.ObjectEquatable` is an opt-in object contract. In Level G, `left <eq> right` lowers to `left.equivalent(right as .object)`.
- `.ObjectKeyStrategy` owns both `hash(key)` and `equivalent(left, right)` for a particular map or set policy.
- `.ObjectComparator` remains the canonical total-order contract for ordered object collections. The older `.Comparator` name remains for compatibility.

An `ObjectKeyStrategy` must return equal hashes whenever it reports two keys as equivalent. It may choose case-sensitive, case-insensitive, namespace-scoped, or domain-specific rules; there is deliberately no universal default object hash and no raw-address hash.

For a copyable handle to a global resource, keep immutable logical fields such as scope, namespace, resource type, value, and generation in the handle. Hash a length-framed representation of exactly those fields and compare the same fields in `equivalent()`. This identifies the resource across handle copies; it does not turn the handle into a live cross-worker reference. See `examples/classes/global_object_keys.crexx` for a complete strategy and map example.

22.13 Notes and Current Boundaries

- Interface default methods are final in Level B.
- Factory providers are selected at runtime from the linked modules.
- Interface method dispatch works for both interface-typed and concrete-class receivers.
- Namespace qualification is a disambiguation mechanism, not a second global symbol path.
- Singleton declarations, external-object adoption syntax, and destructor hooks are intentionally deferred to higher levels.

23

Namespace

In CREXX programming, the related `namespace` and `import` instructions play an important role in organizing and structuring modules within an application. It allows you to define a unique identifier for a module, enabling multiple modules to coexist within the same program while maintaining their own identity and scope.

The namespace instruction is used after the `options` instruction in a CREXX program. By specifying a namespace, you define the logical group or category to which a particular module belongs. This helps in organizing related modules together and facilitates their identification and usage from other modules within the program.

As an example, the built-in-functions package, has a namespace of `rxfnsb`, where the `b` suffix indicates the language level. The `namespace` statement on line 3 identifies this function as part of the `rxfnsb` namespace, which, in CREXX level B, needs to be `import`ed into a program which makes use of it.

```
options levelb
import rxfnsb

say insert("abc","def",2,1)
```

```
deaf
```

This works this way because the built-in function insert (written in Rexx) defines the imported namespace:

```
/* rexx */
```

```
options levelb
namespace rxfnsb expose insert

/* insert(insstr,string,position,length,pad)
inserts string into existing string at certain position and length */

insert: procedure = .string
arg insstr = .string, string = .string, position = -1, len = -1, pad =
    " "


slen=length(string)
  if pad='' then pad=' '

  if len>0 then insstr=substr(insstr,1,len,pad) /* was there an insert
      string length?      */
                                                /* format insert string
                                                     to requested
                                                    length*/
  else if len=0 then insstr=""
  if position<=0 then return insstr||string

  if position>slen then string=substr(string,1,position,pad)  /* is
      position>string length, extend string */

  str1=substr(string,1,position)             /* extract first part of
      string       */
  if position>=slen then str2=''             /* nothing remains of
      string          */
  else str2=substr(string,position+1)        /*    else create str2
                    */
 return str1||insstr||str2                   /* return newly
     constructed string    */
```

One of the key benefits of using namespaces is that it allows you to control the exposure of procedures, members, and globals from one module to another. When you define a namespace, you can specify which elements of that module will be visible and accessible to other modules. This enhances modularity and encapsulation by limiting the scope of variables and procedures to specific namespaces, preventing potential conflicts and promoting code reusability.

If a module does not explicitly include a namespace instruction, CREXX automatically assigns it to a default namespace which is derived from the file name without the `.rexx` file type.

By leveraging namespaces, CREXX programmers can create modular and well-organized programs, enhancing code readability, maintainability, and reusability.

Namespaces promote a structured approach to program and library design, allowing developers to group related functionality together and control the visibility of elements across different modules.

23.1 Qualified references

When two imported namespaces expose the same procedure, class, or interface name, qualify the reference with `namespace..`.

The token to the left of `..` must be a namespace name that has already been made visible through `import`. Qualification does not bypass the import system; it only disambiguates among imported namespaces. `namespace::symbol` remains accepted as a compatibility alias, but `namespace..symbol` is the canonical form.

```
options levelb
namespace qualified_interface_main
import qifa
import qifb

main: procedure
  left = .qifa..vehicle("one")
  right = .qifb..vehicle.from_name("two")

  say left.describe()
  say right.describe()
  return
```

This same form also applies to procedures:

```
answer = qifa..helper()
```

and to type references:

```
current = .qifa..vehicle
```

24

Controlling Arithmetic Behaviour

Rexx decimal arithmetic is based on numbers whose working precision is controlled by the program. This differs from languages in which an operation is determined primarily by a fixed machine type such as a 32-bit integer or a binary double-precision value. A Rexx procedure handling the `.decimal` type instead operates within a numeric context that defines how many significant digits are retained, how numbers are compared, how exponential results are written, and which arithmetic conventions apply.

The `NUMERIC` instruction establishes this context for a procedure. It controls:

- the working precision through `DIGITS`;
- tolerance in numeric comparisons through `FUZZ`;
- scientific or engineering exponential notation through `FORM`;
- the spelling of special numeric values through `CASE`; and
- the selected set of arithmetic rules through `STANDARD`.

In crexx Level B, these settings are fixed at the beginning of each procedure. This restriction allows the compiler to determine the numeric environment before it generates the procedure's code. Settings can either be constants, known at compilation time, or be explicitly inherited from the calling procedure.

Numeric behaviour must be distinguished from numeric *syntax*. The file-level instruction:

```
OPTIONS {NUMERIC_CLASSIC | NUMERIC_COMMON}
```

affects how the parser interprets operators throughout the source file. It can determine such matters as precedence, associativity, and the accepted remainder operator. `NUMERIC STANDARD`, by contrast, selects runtime arithmetic semantics for a procedure. It does not change the syntax tree already constructed by the parser.

Keeping these two responsibilities separate is essential when a source file uses one parsing convention but an individual procedure selects another arithmetic standard.

24.1 Syntax

The `NUMERIC` instruction must be the first instruction in a procedure, immediately following its label. This position permits the compiler to establish the numeric context before compiling the executable body.

```
NUMERIC [ DIGITS [ <constant_value> | INHERITED ] ]
        [ FORM [ SCIENTIFIC | ENGINEERING | INHERITED ] ]
        [ FUZZ [ <constant_value> | INHERITED ] ]
        [ CASE [ UPPER | LOWER | INHERITED ] ]
        [ STANDARD [ CLASSIC | COMMON | INHERITED ] ]
```

The operands have the following meanings:

- `<constant_value>` is a literal whole number, such as `18` or `0`;
- `SCIENTIFIC` and `ENGINEERING` select an exponential notation;
- `UPPER` and `LOWER` select the case used for exponent markers and special numeric values;
- `CLASSIC` and `COMMON` select a set of arithmetic semantics; and
- `INHERITED` obtains the specified setting from the caller's numeric context.

More than one component of the numeric context may be declared by the instruction. Each component may occur only once in a procedure. A setting given as a constant cannot subsequently be changed dynamically.

For example, a procedure can establish an 18-digit context with engineering notation and two ignored comparison digits:

```
calculate:
numeric digits 18 form engineering fuzz 2
```

```
-- procedure body
```

Alternatively, it can inherit selected properties while fixing others:

```
calculate:
numeric digits inherited fuzz 0 standard common

-- procedure body
```

In this example the precision comes from the caller, while comparison fuzz and the arithmetic standard are local constants.

If a component is omitted, its CREXX default is used. Omission does not imply inheritance. Inheritance must always be requested explicitly.

24.2 Procedure Scope

The numeric context applies to all numeric operations performed within the procedure. It remains in effect for the lifetime of that invocation and is not altered by ordinary executable statements.

A called procedure does not automatically acquire every numeric setting from its caller. Unless it requests `INHERITED`, it uses its own declared or default settings. This gives a procedure a stable arithmetic environment independent of the code that happens to call it.

That stability is valuable for library routines. A calculation written and tested for 18 significant digits should not silently run at a caller's lower precision unless the procedure explicitly permits that behaviour. Conversely, `INHERITED` is useful for a general-purpose routine intended to participate in the caller's chosen numeric environment.

24.3 NUMERIC DIGITS

`NUMERIC DIGITS` sets the number of significant digits used for calculations. It defines the working precision of the procedure rather than merely the number of digits displayed in its output.

```
numeric digits 24
```

The value must be a positive whole number and must be greater than the current `NUMERIC FUZZ` value.

The defaults are:

- `18` digits for CREXX Level B; and
- `9` digits in classic Rexx.

A larger value allows calculations to retain more significant information, but generally requires more work and storage. A smaller value can be sufficient for ordinary calculations but may cause earlier rounding during a sequence of operations.

The current setting can be obtained with the `DIGITS()` built-in function:

```
say digits()
```

When inheritance is required, write:

```
numeric digits inherited
```

The effective value is then taken from the caller when the procedure is invoked.

24.4 NUMERIC FORM

`NUMERIC FORM` selects how a number is written when exponential notation is required. It does not change the underlying value or working precision.

The available forms are:

- `SCIENTIFIC`, the default, which places one non-zero digit before the decimal point; and
- `ENGINEERING`, which uses an exponent that is a multiple of three.

For example, a value may be represented conceptually in scientific form as:

```
1.2345E+7
```

Engineering notation adjusts the exponent to a multiple of three and permits one, two, or three digits before the decimal point:

```
12.345E+6
```

Engineering notation is often convenient when values are read together with SI prefixes, because powers of one thousand correspond to prefixes such as kilo, mega, and giga.

The form is selected with:

```
numeric form engineering
```

or:

```
numeric form scientific
```

A procedure that should follow its caller's convention can use:

```
numeric form inherited
```

The effective setting is returned by the `FORM()` built-in function.

24.5 NUMERIC FUZZ

`NUMERIC FUZZ` reduces the number of significant digits considered when numeric values are compared. It provides a controlled tolerance for results that may differ in their least significant positions.

```
numeric digits 18 fuzz 2
```

With this context, calculations still use 18 significant digits, but numeric comparisons disregard the least significant two digits for comparison purposes. `FUZZ` does not reduce the precision at which the calculations themselves are performed.

The value must be zero or a positive whole number smaller than `NUMERIC DIGITS`. A fuzz value equal to or greater than the working precision would leave no significant digits on which to base a comparison and is therefore invalid.

The default is:

```
0
 rexx */
```

With a zero fuzz value, all digits available under the current numeric precision participate in comparisons.

A procedure can inherit the caller's tolerance:

```
numeric fuzz inherited
```

The current value is returned by the `FUZZ()` built-in function.

`FUZZ` should be selected deliberately. It can be useful for comparisons of calculated values, but an unnecessarily large setting may cause values with a meaningful difference to compare as equal.

24.6 NUMERIC CASE

`NUMERIC CASE` controls the case used when numeric conversion produces an exponent marker or a special numeric value. It affects spellings such as:

```
e
inf
nan
```

The available settings are:

- `LOWER`, the default, which produces `e`, `inf`, and `nan`; and
- `UPPER`, which produces `E`, `INF`, and `NAN`.

For example:

```
numeric case upper
```

selects uppercase numeric text for the procedure. This can be useful when generated output must conform to an external convention or when a body of report text uses a consistent uppercase notation.

The setting affects the textual representation, not the mathematical meaning of the value. A procedure can follow the caller by specifying:

```
numeric case inherited
```

The current setting is returned by the `NUMCASE()` built-in function.

24.7 NUMERIC STANDARD

`NUMERIC STANDARD` selects a predefined collection of arithmetic semantic rules. It allows a procedure to request classic Rexx behaviour or the common behaviour adopted by crexx Level B.

```
numeric standard classic
```

or:

```
numeric standard common
```

This setting affects the meaning and constraints of arithmetic operations. It does not alter operator precedence or associativity; those are parser choices made by the file-level `OPTIONS` instruction.

24.7.1 CLASSIC

`CLASSIC` selects the arithmetic conventions defined by ANSI Rexx X3.274-1996 where classic-compatible mode is selected.

Its principal rules include:

- **Remainder (`//` or `%`)** – division is calculated at the current precision and truncated, after which the remainder is computed as:

```
a - (TRUNC(a / b) * b)
```

- **Integer magnitude and precision** – the integer quotient used for `%` and `//` must fit within `NUMERIC DIGITS` without exponential notation. Otherwise, `SYNTAX Error 26.11` is raised.
- **Rounding** – values use the traditional round-half-up rule.
- **Integer division (`%`)** – the operands are divided at the current precision and the result is truncated towards zero.

The magnitude constraint is a characteristic part of classic Rexx numeric semantics. `NUMERIC DIGITS` limits not only the precision of a calculation but also the acceptable representation of the integer quotient involved in these operations.

24.7.2 COMMON

`COMMON` selects semantics closer to those of C-like languages. It is the default standard for crexx Level B.

Its principal rules include:

- **Remainder (`%` or `//`)** – division is calculated at the current precision and truncated, after which the remainder is computed as:

```
a - (TRUNC(a / b) * b)
```

- **Integer magnitude and precision** – the classic quotient constraint is not applied. Quotients may exceed `NUMERIC DIGITS` and may use exponential notation.
- **Rounding** – values use round-half-even. When a value lies exactly halfway between two possible rounded results, the result whose final retained digit is even is selected.

- **Division (`/`)** – when both operands are integers, integer division is performed and any fractional part is truncated towards zero. No signal is raised merely because the mathematical quotient is not integral. If either operand is a float or decimal value, normal float or decimal division is performed.

The last rule is particularly important:

```
numeric standard common

say 5 / 2
```

produces:

```
2
 rexx */
```

not `2.5`, because both operands are integers. Similarly:

```
say 180945931154 / 100
```

produces:

```
1809459311
```

When a fractional result is intended, at least one operand must explicitly be a decimal or floating value:

```
say 5d / 2
say 180945931154d / 100
say .decimal("5") / 2
```

This makes the desired kind of division visible in the source and avoids an implicit change from integer to decimal arithmetic.

24.7.3 Inheriting the Standard

A procedure can use the caller's arithmetic semantics:

```
numeric standard inherited
```

This is suitable for utility routines that are deliberately intended to work within the numeric conventions of their caller. It should not be used merely as a substitute for choosing the standard under which a procedure is meant to operate.

The effective setting is returned by the `STANDARD()` built-in function.

24.8 Interaction with OPTIONS

The distinction between `OPTIONS NUMERIC_CLASSIC` or `OPTIONS NUMERIC_COMMON` and `NUMERIC STANDARD` arises because parsing occurs before procedure execution.

The `OPTIONS` instruction determines how the parser constructs expressions for the entire source file. Once an expression has been parsed, changing a procedure's numeric standard cannot give that expression a different tree.

For exponentiation, the two parser modes produce different results:

24.8.1 Classic parsing

```
say -3**2
```

is interpreted as:

```
(-3)**2
*/
```

and produces `9`.

Similarly:

```
say 2**2**3
```

is interpreted as:

```
(2**2)**3
```

and produces `64`.

24.8.2 Common parsing

Under common parsing:

```
say -3**2
```

is interpreted as:

```
-(3**2)
*/
```

and produces `-9`.

Exponentiation is associated from the right:

```
say 2**2**3
```

is interpreted as:

```
2**(2**3)
```

and produces `256`.

The file-level option can also determine which token is used for remainder, such as `//` or `%`. These are lexical and syntactic decisions and remain constant throughout the file.

`NUMERIC STANDARD CLASSIC` inside a procedure does not make a file parsed with common precedence rules behave as though it had been reparsed in classic mode. It changes only the arithmetic semantics applied after the expression has been constructed.

24.9 Defaults and Inheritance

Each omitted numeric setting receives its defined CREXX default. It does not automatically take the corresponding value from the caller.

This procedure:

```
worker:
numeric digits 24
```

uses 24 digits and the defaults for `FORM`, `FUZZ`, `CASE`, and `STANDARD`. Those other components do not depend on the caller.

By contrast:

```
worker:
numeric digits 24 form inherited standard inherited
```

uses a fixed 24-digit precision but obtains its exponential form and arithmetic standard when it is called.

Selective inheritance permits a routine to protect the settings essential to its calculation while remaining adaptable in areas that affect presentation or calling conventions.

24.10 Runtime Optimization

CREXX can generate more efficient arithmetic code when the numeric context is known during compilation. Requiring `NUMERIC` at the beginning of the procedure and restricting its operands to constants allows the compiler to:

- embed numeric context values directly in generated bytecode;
- avoid repeated runtime lookup of numeric settings;
- select specialized arithmetic operations; and
- reason about the procedure under one stable set of numeric rules.

For example:

```
calculate:
numeric digits 18 fuzz 0 standard common
```

gives the compiler a complete, fixed description of several important arithmetic properties.

`INHERITED` necessarily postpones part of that decision until the procedure is called:

```
calculate:
numeric digits inherited standard inherited
```

The effective settings cannot then be known solely from the procedure's source. This provides flexibility, but it can prevent the corresponding compile-time optimizations.

The choice is therefore intentional:

- use constants when a procedure has defined arithmetic requirements and predictable performance is important; and
- use `INHERITED` when adapting to the caller's numeric environment is part of the procedure's contract.

If a setting is omitted, the default is used and remains known to the compiler. Programmers must explicitly request `INHERITED` when caller-dependent behaviour is required.

25 Statements

25.1 ADDRESS

The ADDRESS instruction is used to effect a temporary or permanent change to the destination of commands. Commands are strings sent to an external environment, and may be sent by clauses consisting of just an expression as well as by the ADDRESS instruction.

To send a single command to a specified environment, an environment name followed by an expression is given. The expression is evaluated, and the resulting command string is submitted to the given environment. After execution of the command the previously selected environment will be unchanged.

In cRexx, ADDRESS sends commands or function requests to a named external environment. It is implemented through the current compiler-exit and VM environment protocol. If no environment is selected, the default command environment is CREXX.

Basic command form:

```
address crexx "echo hello"
```

Command output and error streams can be captured:

```
address crexx "echo #42" output out error err
say out
```

The capture destination determines the representation:

- a `.string` receives the stream text, including line-ending characters as delivered by the selected environment;
- a `.string[]` receives one element per newline-delimited record, without the newline; an empty line is retained as an empty element; and
- `OUTPUT` and `ERROR` are independent, so either or both may be a string or string array.

After the command returns, `rc` contains its integer status independently of whether either stream was captured. A successful command may produce no output; with fresh destinations that is an empty string or an array whose count is zero. Captured Unicode text remains Level B text. Quotes, brackets, semicolons, pipes, and similar delimiters in emitted text are data and receive no special interpretation from capture.

```
input_lines = .string[]
input_lines[1] = "echo first"
input_lines[2] = "echo second"

output_lines = .string[]
error_text = .string
address crexx "batch" input input_lines output output_lines error
    error_text

if rc = 0 then do
  do i = 1 to output_lines.0
    say output_lines[i]
  end
end
```

Array destinations are mutable append targets. Redirection adds each captured record after the array's current last element; it does not clear or replace existing elements. An empty stream adds nothing, so any existing elements remain. Command failure does not change this rule: emitted records are appended to the corresponding `OUTPUT` or `ERROR` array, while a stream that emits nothing leaves its array unchanged.

Use `arraydrop` immediately before each command when the array should contain only that command's records. `OUTPUT` and `ERROR` are independent arrays and must each be dropped when replacement-style reuse is required:

```
output_lines = .string[]
error_lines = .string[]

call arraydrop output_lines
```

```
call arraydrop error_lines
address crexx "echo current" output output_lines error error_lines
```

A fresh array needs no preliminary `arraydrop`. Reusing without a drop is the supported accumulation form, not an implicit reset operation.

When `OUTPUT` or `ERROR` is omitted, that stream is written to the normal CREXX standard stream and flushed as the command emits it. This includes the corresponding child stream from `ADDRESS CREXX "run ..."`: no hidden capture is inserted before the inherited stream. Long-running child or later work does not delay already-emitted CREXX command output until command or task completion. A stream with an explicit `OUTPUT` or `ERROR` destination retains the requested string/array capture semantics.

25.1.1 Built-In Command Environments

The built-in environments enable a specific use of the underlying operating system functionality.

Environment	Purpose
`CREXX`	The default CREXX command environment. It is cREXX-specific and OS-independent, implemented by CREXX rather than by a shell.
`SYSTEM`	The platform command processor. On POSIX this is standard `sh -c`; on Windows this is `%COMSPEC% /D /S /C` with a `cmd.exe` fallback.
`COMMAND`, `CMD`	Compatibility aliases for `SYSTEM`.
`PATH`	Direct executable dispatch through the platform process API. It resolves executables through process `PATH` where needed and calls them without shell syntax.
`SHELL`	Explicit configured shell dispatch. Set `CREXX_ADDRESS_SHELL` to the shell executable and optionally `CREXX_ADDRESS_SHELL_ARGS` to the argument list used before the command text. If unset, it falls back to the platform command processor defaults.

```
address system "echo one && echo two" output out error err
address cmd "cd ."
address path "rxas -h" output out error err
```

25.1.2 CREXX Command Environment

`CREXX` is the default addressable command environment, but it is not a shell implementation. It is a CREXX specific command environment with stable command names and CREXX defined return-code behaviour across supported operating systems. It does not interpret shell punctuation such as `;`, `&&`, `||`, or pipes. Use multiple `ADDRESS` statements, or send newline-separated commands to `ADDRESS CREXX "batch"`. Blank batch lines and lines whose first non-blank characters are `--` are skipped; batch stops at the first non-zero return code.

`cd`, `pushd`, and `popd` change the current VM worker's logical working directory. It persists for later `ADDRESS CREXX` commands in that VM and is copied into children launched by `run`, `PATH`, `SYSTEM`, or `SHELL`; it never temporarily changes the host process working directory. File commands resolve relative paths against the same logical directory. In contrast, `ADDRESS SYSTEM "cd path"` changes only that child command processor and does not update the worker's logical directory after the child exits.

`setenv` and `unsetenv` likewise update a worker-owned environment overlay. `env`, `which`, configured-shell lookup, and child launch see the overlay, while unrelated VM workers and native host threads continue to see the unchanged process environment. A child receives an immutable merged environment snapshot.

The command set is useful but bounded. CREXX command names are literal. Host-variable anchors are supported only in command operands. In the table below, `anchorable` means an operand may be a scalar anchor such as `:name` or `${name}`. An operand ending in `...` may also be supplied by a stem anchor such as `:name[]`, `:name.`, `${name[]}`, or `${name.}`, which expands to zero or more operands. For fixed-arity commands, a stem anchor is valid only when it expands to exactly the number of operands required at that point.

Command	Anchorable operands	Behavior
`help`	None.	Print the command list.
`echo [text...]`	`text...` may use scalar or stem anchors.	Write text followed by a newline.
`pwd`	None.	Print the current VM worker's logical working directory.

Command	Anchorable operands	Behavior
`cd [path]`	`path` may use a scalar anchor, or a one-item stem anchor.	Change the worker's logical working directory; no path means the user's home directory where known.
`pushd path`	`path` may use a scalar anchor, or a one-item stem anchor.	Push the current directory and change to `path`.
`popd`	None.	Return to the most recent pushed directory.
`ls [path...]`, `dir [path...]`	`path...` may use scalar or stem anchors.	List directory entries, excluding . and …
`exists path...`	`path...` may use scalar or stem anchors.	Print `1 path` or `0 path` for each path; returns non-zero if any are missing.
`stat path...`	`path...` may use scalar or stem anchors.	Print type, size, and modification time for each path.
`mkdir [-p] path...`	`-p` may be literal or scalar-anchored; `path...` may use scalar or stem anchors.	Create directories; `-p` creates missing parents.
`rmdir path...`	`path...` may use scalar or stem anchors.	Remove empty directories.
`rm [-r] path...`, `del [-r] path...`	`-r` may be literal or scalar-anchored; `path...` may use scalar or stem anchors.	Remove files, or recursively remove paths with `-r`.
`copy source target`, `cp source target`	`source` and `target` may use scalar anchors, or one stem anchor that expands to both operands.	Copy a file.
`move source target`, `mv source target`, `rename source target`	`source` and `target` may use scalar anchors, or one stem anchor that expands to both operands.	Rename or move a path.
`touch path...`	`path...` may use scalar or stem anchors.	Create files if missing and update modification times.
`cat path...`, `type path...`	`path...` may use scalar or stem anchors.	Write file contents to the command output stream.
`head [-n count] path`	`count` and `path` may use scalar anchors; a stem anchor must expand to the exact option/value/path shape.	Write the first lines of a file; default count is 10.

Command	Anchorable operands	Behavior
`tail [-n count] path`	`count` and `path` may use scalar anchors; a stem anchor must expand to the exact option/value/path shape.	Write the last lines of a file; default count is 10.
`lines [path]`	`path` may use a scalar anchor, or a one-item stem anchor.	Count lines in a file, or in redirected command input when no path is supplied.
`write path text...`	`path` may use a scalar anchor; `text...` may use scalar or stem anchors.	Replace a file with the supplied text.
`append path text...`	`path` may use a scalar anchor; `text...` may use scalar or stem anchors.	Append the supplied text to a file.
`which command`	`command` may use a scalar anchor, or a one-item stem anchor.	Resolve an executable through the worker environment's `PATH`.
`now [local|utc]`, `date [local|utc]`	`local`/`utc` may be literal or scalar-anchored.	Print an ISO-like timestamp.
`sleep seconds`	`seconds` may use a scalar anchor.	Sleep for the requested duration.
`platform`, `os`	None.	Print operating-system and architecture details.
`env [name...]`	`name...` may use scalar or stem anchors.	Print all environment variables, one variable's value, or `name=value` lines for multiple names.
`setenv name value...`	`name` may use a scalar anchor, or a stem anchor whose first item is the name. `value...` may use scalar or stem anchors and is joined with spaces.	Set a variable in the current worker's environment overlay.
`unsetenv name...`	`name...` may use scalar or stem anchors.	Hide one or more variables in the current worker's environment overlay.
`pid`	None.	Print the current `CREXX` process id.
`ps [pid]`	`pid` may use a scalar anchor.	Print current process details, or check whether a process id is alive.
`kill pid [signal]`	`pid` and `signal` may use scalar anchors.	Terminate or signal a process.

Command	Anchorable operands	Behavior
`resolve host`	`host` may use a scalar anchor.	Resolve host names to numeric addresses.
`tcp host port`	`host` and `port` may use scalar anchors, or one stem anchor that expands to both operands.	Check that a TCP connection can be opened.
`batch`	None in the `batch` command itself; input lines are runtime text and are not compiler auto-exposed.	Read commands from input and execute them in order.
`run executable [arg...]`	`executable` may be literal or scalar-anchored. `arg...` may use scalar or stem anchors. `run :argv[]` is also accepted: the first stem item is the executable and the remaining items are arguments. The word `run` itself must be literal.	Execute a direct `PATH` command and forward its output and error streams.

Anchors must occupy a whole parsed operand. For example, `cat :file` expands `:file`, but `--file=:file` is a literal operand. Build combined operands in Rexx first, then pass the result with a scalar anchor. Stem anchors preserve each item as one operand, even when an item contains spaces or shell punctuation such as `&&`.

25.1.3 Argument-Vector Execution

The argv-preserving command path is `ADDRESS CREXX "run :argv[]"`. The first array element is the executable and every later element is exactly one child argument. Empty elements remain empty arguments; whitespace, quotes, Unicode, wildcards, variable-like text and shell metacharacters are not split, expanded or executed. The child is launched directly through the platform process API.

```
argv = .string[]
argv[1] = executable
argv[2] = "value with spaces"
argv[3] = ""
argv[4] = "; && | * ?"

out = .string[]
err = .string[]
```

```
address crexx "run :argv[]" output out error err
say rc
```

This is intentionally distinct from `ADDRESS COMMAND`, `ADDRESS SYSTEM`, and `ADDRESS CMD`: those environments accept one command string and invoke the platform command processor. Use them only when shell parsing is required. The `PATH` environment is direct process dispatch too, but accepts parsed command text; use `CREXX run :argv[]` when exact argument boundaries matter.

25.2 ARG

The ARG statement receives the arguments to programs or procedures and specifies the number of (required and optional) arguments and the expected types. In CREXX level B, it is *not* a short form of PARSE UPPER ARG but a seperate statement.

See the Procedures and Arguments section on page 183 for more information.

25.3 CALL

CALL routine [parameter] [, [parameter] …]

`CALL` invokes a procedure, function, or method for its side effects and discards any returned value. In a class method, an unqualified method name is a call on the current receiver, in the same way that `rc = append(value)` calls the current receiver and keeps the result:

```
fromArray: method = .int
  arg list = .object[]
  call clear()
  loop i = 1 to list[0]
    rc = append(list[i])
  end
  return 0
```

A qualified receiver can also be used:

```
call list.add("red")
```

The receiver may be a postfix expression, including an indexed value, a parenthesized expression, a factory or function result, or an earlier method result:

```
call lists[index].add("red")
call makeList().add("blue")
call makeContainer().list().clear()
```

The receiver expression and each argument are evaluated once in normal call order. If the method mutates a variable-like receiver such as `lists[index]`, the changed object is written back through that same selected location. The method's returned value, if any, is discarded by `CALL`.

Use an expression call when the returned value is required.

25.4 Array Mutation

Level B supports core statement forms for mutating one-dimensional dynamic raw typed arrays:

```
append items with value
insert items with value at index
remove items at index
remove items at index for count
remove items at first to last
clear items
```

The target must currently be a raw dynamic array such as `.string[]` or `.int[]`. Appended or inserted values must be assignable to the array element type, and index/count/range expressions must be integers. Fixed-size arrays, multi-dimensional arrays, and object/interface collection targets are not part of this first implementation phase.

```
items = .string[]
append items with "red"
append items with "blue"
insert items with "green" at 2
remove items at 1
clear items
```

`append`, `insert`, `remove`, `clear`, and `at` are contextual in this statement surface; method calls such as `list.append(value)` and variables named `at` remain ordinary symbols outside these statement forms.

25.5 CONSTANT

Level B supports named compile-time constants:

```
flags: procedure = .int
  constant FLAG_STRING = 0x00010000
  constant PAYLOAD = "41424344"x as .binary
  return FLAG_STRING
```

Constants must be declared inside an explicit procedure, method, or factory scope. A `constant` declaration in the file body is a design defect and is not part of the Release 1 surface, because it is an instruction form and can accidentally imply an implicit `main()` in a library-shaped module.

When a script needs shared constants in a separate declaration procedure, put the script body in an explicit `main`. A procedure body continues until the next callable boundary, so executable statements after a declaration procedure belong to that procedure unless a new `main: procedure` boundary is present.

The initializer must be a compile-time constant expression. The declared name is immutable after declaration and can be used in ordinary expressions in its visible scope or as an inline assembler literal operand. Integer constants used as assembler immediates are substituted directly; string, decimal, float, and binary payload constants are stored through the normal RXBIN constant-pool path.

25.6 DO/END

DO [repetitor] [conditional] ; [clauses]

expr ::= DO ; [clauses] END

END [symbol] ;

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

conditional : = WHILE exprw UNTIL expru

The DO/END statement is the command employed to iterate and group multiple statements into a singular block. This instruction consists of multiple clauses.

When `DO ... END` appears where an expression is expected, it is parsed as a block expression. In that form the body must yield a value using `LEAVE WITH expr`.

Simple `DO ... END` groups may also carry block-scoped signal handlers using `ON SIGNAL` clauses. The handler clauses are valid only on a simple `DO` group, not on counted, conditional, forever, or expression-form `DO`. To protect code inside a loop, nest a simple signal-handling `DO ... END` group inside the loop body.

25.7 EXIT

EXIT [expr] ;

Causes the Rexx program to cease execution and, optionally, returns the expression expr to the calling program.

25.8 IF/THEN/ELSE

IF expr [;] THEN [;] statement

[ELSE [;] statement]

This provides the standard conditional statement structure.

25.9 ITERATE

ITERATE [symbol] ;

The ITERATE instruction will execute the innermost, active loop in which the ITERATE instruction is situated repeatedly. If a symbol is specified, it will execute the innermost, active loop having the symbol as the control variable repeatedly.

25.10 LEAVE

LEAVE [symbol] ;

LEAVE WITH expr ;

This statement terminates the innermost, active loop. If symbol is specified, it terminates the innermost, active loop having symbol as control variable.

`LEAVE WITH expr` is distinct from loop-control `LEAVE`. It exits the innermost enclosing expression-form `DO ... END` block and returns the value of `expr` to the parent expression.

25.11 NOP

NOP ;

The NOP instruction is the *No Operation* directive; it executes without performing any operation. It is syntactically valid but intentionally does nothing. It exists primarily as a placeholder, in the following cases:

- as a placeholder while developing or debugging;
- in generated code where an empty statement is needed;
- to make an intentionally empty branch explicit rather than accidental

This example shows how a branch can be temporarily disabled while developing by inserting a `nop` statement:

```
if retries > 0 then
    nop               /* retry logic temporarily disabled */
else
    call abortTransfer
```

25.12 OPTIONS

OPTIONS expr ;

The OPTIONS instruction is used to set various interpreter-specific options. See Language Level and Options Section

25.13 PARSE

PARSE [option] [CASELESS] type [template] ;

Current implementation status:

- `PARSE VALUE ...`, `PARSE VAR ...`, and `PARSE ARG ...` are implemented through the certified `PARSE` exit.

- `PARSE VERSION template` uses the result of the `version()` built-in function as its source and applies the normal PARSE template rules. For example, `PARSE VERSION one` keeps the complete version string in `one`, while `PARSE VERSION one two` assigns successive words to `one` and `two`.
- `PARSE SOURCE template` uses the result of the `sourceinfo()` built-in function as its source and applies the normal PARSE template rules. The source string contains the system, invocation mode, and source file name.
- These forms are supported in both Level G and Level B source. Their internal certified lowering may use Level B instructions; that compiler detail does not permit an authored `ASSEMBLER` statement in Level G source.
- `PARSE ARG` uses the current procedure's `arg()` compatibility view.
- In implicit `main`, that means command-line arguments.
- In other procedures, that means the `...` tail if present, or an empty source string if there is no `...` tail.

25.14 PROCEDURE

`PROCEDURE` starts a named local routine and optionally declares its return type and the module-global variables that routine can see.

Common forms:

```
name: procedure
name: procedure = .int
name: procedure = .void expose state count
```

Procedure-level `expose` is local to that procedure declaration. The listed names are bound to module-global storage shared with other procedures that also expose the same names. A procedure that does not list the name does not see that exposed storage unless the name is also exposed by the file-level `namespace ... expose ...` declaration.

```
proc1: procedure = .void expose var
  var = "Hello World"
  return

proc2: procedure = .void expose var
  say "var is" var
  return
```

This is distinct from `ARG expose`, which exposes a call argument by reference.

25.15 INITIALISER

`INITIALISER` declares a private procedure that the VM runs automatically before its mutable module instance becomes executable.

```
name: initialiser
name: initialiser expose state cache
```

A module may contain zero or more initializers. They run once per mutable module instance in declaration order. An initializer has no arguments and an implicit `.void` return type. It may use a bare `return`, but it cannot declare `arg` parameters or return a value.

The label has the module's normal namespace-qualified identity in metadata and diagnostics, but it is not a callable or exportable procedure. A direct call to the label and a file-level `namespace ... expose` of the label are compile errors. The optional initializer-level `expose` clause binds module-global variables into the initializer body; it does not expose the initializer itself.

The VM runs initializers after native-provider resolution, linking, and execution-image preparation, and before `main` or a public host call. A call from an initializer into an unready module initializes the target module first; an initialization cycle fails rather than entering a partially initialized module.

25.16 SAY

SAY [expr] ;

Evaluates the expression expr and prints the resulting string onto the standard output stream.[13]

25.17 MSAY

MSAY mask, value-1, value-2, …

Example:

```
options levelb comments_dash numeric_classic
```

[13] with an added newline. For cases where no newline is wanted, the conventional way of using `call lineout` can be used, or the `sayx` assembler instruction.

```
import rxfnsb
numeric digits 15
b=.decimal;c=.decimal;p=.decimal
n=.decimal;mn=.decimal
b = 963807195502/100
c = 180945931154/100
p = 0.081090
n = 30
-- COMPUTE MN = (1 + P) ** N
mn = (1+p) ** n
-- COMPUTE B = (B + C) * MN + C * ( (MN - 1) / P - 1) .
b = (b+c) * mn + c * (( mn - 1) / p - 1)
msay "$$$,$$$,$$$,$$$,$$9.99.",b
```

```
$326,047,467,304.71
```

`MSAY` is a convenience syntax that formats values using `fmtmask` and immediately outputs the resulting line. The facility is intended for report-style output where fixed-width text and numeric alignment are desirable. See MSAY and fmtmask on page 195 for the complete template syntax.

25.18 FSAY

`FSAY` is a convenience syntax that formats the template using `fsayfmt()` and outputs the resulting line.

Unlike `fmtmask` and `MSAY`, which use COBOL-inspired picture masks, `fsayfmt` uses embedded placeholders similar to modern string interpolation systems.

```
FSAY template
```

The template may contain one or more placeholders enclosed in braces.

Example:

```
name  = "Fred"
qty   = 12
price = 64.31

FSAY "Name: {name:<10} Qty: {qty:>3} Price: {price:8.2}"
```

```
Name: Fred       Qty:  12 Price:       64.31
```

See fsay and fsayfmt on page 203 for the complete template syntax.

25.19 SELECT/WHEN/OTHERWISE

SELECT [expression] [;] WHEN expression [, expression …] [;] THEN [;] instruction [;] [WHEN expression [, expression …] [;] THEN [;] instruction [;]] … [OTHERWISE [;] [instruction] [;] …] END [;]

The SELECT statement allows you to conditionally evaluate multiple expressions and execute corresponding instructions based on the first expression that evaluates to true (1).

There are two styles of the SELECT statement in cRexx:

1. **Classic SELECT:** Does not include an initial `expression` after the `SELECT` keyword. Each `WHEN` expression is evaluated as a standalone boolean condition.
2. **C-Style SELECT (SWITCH):** Includes an initial `expression` after the `SELECT` keyword. The `expression` is evaluated once, and its result is implicitly compared for equality (=) against each `WHEN` expression.

If a `WHEN` condition is met, its associated `THEN` instruction is executed, and control exits the `SELECT` block. If no `WHEN` condition is met, the `OTHERWISE` block (if present) is executed. If no `WHEN` condition is met and an `OTHERWISE` block is absent, the `SELECT` statement acts as a `NOP` (null operation) and does nothing.

25.20 SIGNAL

Signals are Level B error/condition objects implementing `.signal`. Rexx-created signals can be raised with a signal object or with the compact named forms:

```
signal .signal("error", "message")
signal error
signal error "message"
```

Procedure-scoped handlers are installed with `SIGNAL ON` and removed with `SIGNAL OFF`:

```
signal on conversion_error call handle_conversion
signal on error, syntax call handle_problem
signal off conversion_error

handle_conversion: procedure = .signalaction
  arg problem = .signal
  say problem.source()
  return .signalaction.skip()
```

The handler procedure receives one `.signal` argument and returns a `.signalaction`: `.signalaction.skip()` or `.signalaction.fail()`. Retrying work requires an explicit source-level loop; instruction-level retry is not a handler action.

Block-scoped handlers use `ON SIGNAL` clauses on a simple `DO ... END` group:

```
do
  risky_work()
on signal conversion_error as problem
  say problem.source()
on signal error, syntax
  call cleanup()
on signal
  call log_unhandled_signal()
end
```

The statements before the first `ON SIGNAL` clause are the protected body. Normal completion skips the handlers. A handler that completes normally leaves the `DO` block. `ON SIGNAL` with no names catches all maskable signals. `AS name` binds the current `.signal` object; if `AS` is omitted, no signal object is available to that handler.

Only a simple `DO ... END` group can carry `ON SIGNAL` clauses. Counted, conditional, forever, and expression-form `DO` loops do not carry handlers directly. To protect code inside a loop, put a simple signal-handling `DO ... END` group inside the loop body.

25.21 TRACE

`TRACE` enables or disables VM breakpoint-backed tracing for the current call frame and procedures called from it.

Supported forms are:

```
trace off
trace normal
trace results
trace rexx
trace asm
trace as
trace llm
trace ll
trace env
trace value expr
trace suppress namespace name
trace unsuppress namespace name
trace add suppressed namespace name
trace remove suppressed namespace name
trace reset namespaces
trace results to stderr
trace llm to file "trace.jsonl"
```

The standard Rexx option letters `A`, `C`, `E`, `F`, `I`, `L`, `N`, `O`, and `R` are accepted, including a leading `?` prefix and signed integer settings. Options use a minimum-abbreviation rule: the spelling must be a left-prefix of the full option word. For example, `TRACE R`, `TRACE RE`, `TRACE RES`, `TRACE RESULT`, and `TRACE RESULTS` all select Results, while `TRACE RAS` is invalid. The crexx extensions use `AS` as the minimum abbreviation for `ASM` and `LL` as the minimum abbreviation for `LLM`; `ENV` is an exact crexx extension spelling. `TRACE REXX` remains supported as an exact legacy crexx source-trace spelling; it is not abbreviated because `R` and `RE...` belong to Results. This is not yet full semantic compatibility, but the noninteractive output shape follows the standard prefix vocabulary for implemented events:

```
  >  >   escaped-source-file
5 *-* escaped-source
  >=>   "escaped-assignment-result"
  +++   RC=-3 ENVIRONMENT escaped-command
```

`TRACE N` is the quiet/default mode: it does not trace ordinary statements and emits `+++` only for failing ADDRESS commands. `TRACE C`, `TRACE E`, and `TRACE F` are ADDRESS-command driven. `TRACE A` traces source clauses. Classic `TRACE R` traces source clauses, variable substitutions, and assignment or expression results. crexx emits source clauses from `.srcstep` metadata and semantic value records from `.traceevent` metadata, including initial `>=>` assignment, `>V>` variable, `>L>` literal, and `>O>`/`>P>` operation coverage where the compiler can point at an available register or constant. `TRACE I` uses the same metadata path and adds intermediate-event visibility as coverage grows. `TRACE L` is accepted, but label-pass events are

not emitted yet. `O`/`OFF` disables breakpoint tracing. `TRACE ASM` traces VM/RXAS instruction information and includes source text where metadata is available.

When traced execution moves between source files, CREXX emits a source-file transition line:

```
> > helper.crexx
```

The first visible source file is not printed, so single-file traces keep their classic shape. Later file changes are printed before the next `*-*` source line, including when execution returns to the original file. This is a CREXX extension; classic Regina-style output does not provide an equivalent filename record.

This TRACE implementation is still beta. Source reporting now uses self-contained source-step metadata, and text TRACE no longer guesses assignment results from source text. Result coverage is still deliberately partial: optimized-away or folded values may have no trace event, and some compound-variable details such as final resolved-name reporting remain a compiler/ runtime coverage task.

TRACE reports the executable program after optimisation. Folding, fusion, inlining, elimination and code motion or loop hoisting can therefore change, remove or relocate trace events. This does not affect ordinary program semantics, and retained events must still read the correct available value, but optimised trace output is not required to match the unoptimised event stream. For the closest correspondence to authored execution, use `crexx --nooptimise`, or disable both stages explicitly with `rxc -n` and `rxas -n`. The beta coverage limits above still apply.

`TRACE LLM` is a CREXX extension that emits one JSON-lines-style trace record per event. It is intended for debugger automation and for validating emitted `.srcstep` metadata; source text is escaped so control characters and backslashes remain visible. `TRACE VALUE expr` evaluates `expr` at runtime and normalizes it using the same trace option rules.

Trace output defaults to stdout. Add `TO STDERR`, `TO STDOUT`, `TO FILE expr`, or `TO expr` to choose a sink. `TO FILE` opens the selected file in append mode for each trace record.

TRACE normally hides events from system library and debugger namespaces so a user trace follows the program being debugged instead of the machinery that implements tracing. The default suppressed namespaces are `rxfnsb`, `_rxsysb`, `rxfnsg`, `_rxsysg`, `rxfnsl`, `_rxsysl`, `rxfnsc`, `_rxsysc`, `rxcp`, `rxcpexits`, `rxcptest`,

`rxdb`, `rxdbgui`, `runtime_signal`, `signalaction`, and `library`. The TRACE runtime internals themselves are always hidden.

Use namespace controls when you need to include or exclude a library while debugging:

```
trace results
trace unsuppress namespace rxfnsg
trace suppress namespace myframework
trace reset namespaces
```

`TRACE SUPPRESS NAMESPACE name` and `TRACE ADD SUPPRESSED NAMESPACE name` suppress a namespace. `TRACE UNSUPPRESS NAMESPACE name` and `TRACE REMOVE SUPPRESSED NAMESPACE name` make that namespace visible again. `TRACE RESET NAMESPACES` restores the default suppression list and clears per-session changes. Namespace names may be bare identifiers or string literals; matching is by namespace or path component, not by arbitrary substring, so suppressing `rxfnsg` does not suppress `myrxfnsghelper`.

`TRACE ENV` explicitly checks two environment variables at that point in the program. `CREXX_TRACE` supplies the mode using the same option rules as `TRACE VALUE`, and `CREXX_TRACE_TO` supplies the sink using the same rules as `TO`. For example, `CREXX_TRACE=results CREXX_TRACE_TO=stderr` can switch tracing on at a compiled `TRACE ENV` marker without editing the source. If `CREXX_TRACE` is unset or empty, `TRACE ENV` turns tracing off. If `CREXX_TRACE` has an invalid value, `TRACE ENV` turns tracing off and emits a `+++` trace message naming the invalid value.

TRACE is implemented as a certified compiler exit. It requires normal compiler exit loading; compiling with exits disabled rejects the statement rather than treating it as an implicit command.

The CREXX standard-library/BIF build deliberately compiles most built-in function source files with compiler exits disabled (`rxc -x`) to avoid bootstrap and circular-dependency problems while building the library that the exits themselves use. Adding `TRACE RESULTS`, `TRACE R`, or another explicit TRACE instruction directly to a BIF source file such as `abs.crexx` therefore produces `#CERTIFIED_EXIT_DISABLED`. Debug BIF or library behavior from a normal test program instead: call the BIF from a fixture that compiles with exits enabled, use `TRACE UNSUPPRESS NAMESPACE rxfnsb` if you need to see library frames, and keep linked/native images unstripped with `--link-keep-source` when source-level TRACE metadata is needed.

Implementation status and compatibility requirements are tracked in `docs/ai-context/CREXX_TRACE_REQUIREMENTS.md`.

26 Concurrency

This chapter defines the current CREXX task and parallel-block language surface. It is an initial Level G extension. The Level B class contracts used by its lowering are specified in the library reference.

26.1 Language level and contextual words

Task declarations, task-target expressions and both forms of `DO PARALLEL` are accepted only when the source selects:

```
options levelg
```

`TASK` and `PARALLEL` are case-insensitive contextual words. They are not globally reserved. A variable or ordinary routine named `task` remains valid when the token is not in one of the grammar positions defined below; `parallel` remains an ordinary symbol except immediately after the opening `DO` of a parallel block.

The Level B classes `.taskpool`, `.taskscope`, `.task`, `.tasktarget`, `.taskwork`, `.taskcontext`, `.completion`, `.channel`, `.channelrequest`, `.channelvalue`, `.channelcodec`, `.byteendpoint`, `.serviceref` and `.transferbuffer` do not require Level G.

26.2 Task callable declarations

The callable forms are:

```
task-procedure ::= symbol ":" "task" [ "=" type ] callable-body
task-method    ::= symbol ":" "task" [ "=" type ] callable-body
```

At module scope the form declares a task procedure. Within a class it declares a task method. Arguments use the normal typed `ARG` statement. Omitting the result type declares a `.void` task.

```
options levelg
namespace task_example

checksum: task = .int
  arg text = .string
  return length(text)

notify: task
  arg text = .string
  say text
  return
```

The task modifier controls submission when the callable is invoked from a controlling execution. The body itself remains an ordinary callable body executed in one worker-owned Rexx execution.

26.2.1 Transferable signature types

The current typed task-call lowering accepts these scalar argument/result types:

- `.boolean`;
- `.int`;
- `.string`;
- `.binary`; and
- `.void` as an omitted result only.

A `.channelvalue` is accepted as a direct task argument. A concrete class is accepted as an argument, result or task-method receiver only when the exact static class declares the transfer contract in the next section.

Current task signatures do not accept arrays, references, varargs, `ARG EXPOSE`, `.float`, `.decimal`, an interface result, untyped `.object`, or another type without the exact transfer contract. RXCV can represent more value kinds than the Level G typed-call sugar currently lowers; the broader Level B `.channelvalue` and `.taskwork` path remains available when needed.

26.3 Transfer contract for concrete classes

A transferable concrete class declares exactly resolved conversion members:

```
from_channel: factory
  arg encoded = .channelvalue

to_channel: method = .channelvalue
```

`to_channel` has no arguments. `from_channel` has exactly one ordinary `.channelvalue` argument. Both members must be declared on the exact concrete class known at the task call site; an interface is not enough to select unique reconstruction code.

The receiver and argument direction is controller `to_channel`, worker `from_channel`. The result direction is worker `to_channel`, controller `from_channel`. Reconstruction creates new receiver-owned objects. No live object identity or reference crosses the boundary.

This complete task-method example uses the transfer contract for its receiver, argument and result:

```
numberbox: class
  _value = .int

  *: factory
    arg value = .int
    _value = value
    return

  from_channel: factory
    arg encoded = .channelvalue
    _value = encoded.as_integer()
    return

  to_channel: method = .channelvalue
    return .channelvalue.integer_value(_value)

  value: method = .int
    return _value

calculator: class
  _base = .int

  *: factory
    arg base = .int
    _base = base
    return
```

```
  from_channel: factory
    arg encoded = .channelvalue
    _base = encoded.as_integer()
    return

  to_channel: method = .channelvalue
    return .channelvalue.integer_value(_base)

  add: task = .numberbox
    arg amount = .numberbox
    return .numberbox(_base + amount.value())
```

Its ordinary-looking method call is a task call:

```
calculator = .calculator(40)
answer = calculator.add(.numberbox(2))
```

The complete executable program is `../crexx_programming_guide/examples/concurrency_transferable_objects.crexx`.

26.4 Calling a task

A call resolved to a task callable uses the existing Rexx procedure/function or method-call grammar. No `ASYNC` call prefix is added.

```
answer = checksum("hello")
call notify "complete"
answer = calculator.add(.numberbox(2))
```

Outside `DO PARALLEL`, a task-valued expression has an implicit expression scope that closes before its containing statement completes. A `CALL` task has an implicit statement scope and completes before the next statement.

26.4.1 Expression evaluation

For an expression containing task calls, the implementation:

1. evaluates ordinary receiver and argument expressions left to right;
2. encodes each task's inputs after those inputs are available;
3. submits ready tasks in source evaluation order;
4. permits independent submitted bodies to overlap;
5. evaluates ordinary calls and operators in their existing controller order; and

6. applies an operator after the task operands it needs have materialized.

Thus:

```
answer = first() + second()
```

submits `first` before `second`, permits the two task bodies to overlap, waits for both successful values, and then performs ordinary addition. It does not make `+` asynchronous.

In `first() + ordinary()`, the controller may call `ordinary()` while `first()` runs, but `ordinary()` is not moved to a worker. Existing short-circuit behavior is retained. A task in an unevaluated right branch is not submitted.

The complete positive expression/block program is `../crexx_programming_guide/examples/concu` `basic.crexx`.

26.5 Parallel blocks

The grammar is:

```
parallel-do ::= "do" "parallel" [ "using" expression ]
                clauses
                "end" [ symbol ]
```

The form is available wherever the corresponding ordinary `DO ... END` form is available. In expression position the body must produce a result through the existing `LEAVE WITH expression` statement.

```
do parallel
  first_value = first()
  second_value = second()
  say "controller is still running ordinary clauses"
end
```

Only calls to task callables and explicit Level B submissions create child work. Other clauses execute sequentially in the controlling execution.

All task calls in one block belong to one scope. Without `USING`, the compiler creates a scope over the execution-local default pool. `USING expression` is evaluated once before the block opens and must yield a fresh open `.taskscope`:

```
pool = .taskpool.local(2, 8)
scope = .taskscope.collectall(pool, 60000)
```

```
answer = do parallel using scope
  left = first()
  right = second()
  leave with left + right
end

call pool.close()
```

Block exit consumes and closes the scope, including exit by `RETURN`, `EXIT`, `LEAVE`, controller signal or failed ordinary expression. The exit requests child cancellation where required and accounts for all children before control escapes. It does not close the pool. A consumed scope cannot be reused.

The full local/process example is `../crexx_programming_guide/examples/concurrency_providers.crexx`.

26.5.1 Pending result bindings

Within `DO PARALLEL`, assignment directly from a task call creates a typed pending binding:

```
do parallel
  left = first()
  right = second()

  say left       /* materializes left here */
end              /* accounts for right even if it was never read */
```

Reading the binding materializes it and waits if necessary. It then becomes an ordinary value. Before materialization the following are invalid:

- reassignment;
- passing or taking it by reference;
- `ARG EXPOSE` use;
- indexed or attribute mutation through it; and
- returning or otherwise escaping it from the scope.

For example, this is rejected with `#PENDING_TASK_RESULT_MUTATION`:

```
do parallel
  value = first()
  value = 8
end
```

Expression-form blocks return only a materialized value:

```
answer = do parallel
  left = first()
  right = second()
  leave with left + right
end
```

26.6 Task-target expressions

The grammar is:

```
task-target ::= "task" callable-reference
              | "task" class-factory-expression
```

The callable form requires a statically resolved task procedure or generated task-method adapter:

```
target = task checksum
```

The factory form requires a statically resolved concrete factory whose class implements `.taskwork`:

```
target = task .imagework("thumbnail")
```

Factory arguments are evaluated on the controller and reconstructed in the worker. The current factory-target lowering accepts `.boolean`, `.int`, `.string`, `.binary` and direct `.channelvalue` arguments. It does not use the general concrete-object `to_channel` contract for factory arguments.

Dynamic procedure names and strings are invalid:

```
selected = "my_module.checksum"
target = task selected              /* #TASK_DYNAMIC_TARGET */
```

The compiler, assembler and linker seal the selected callable, signature and linked-image identity. `.tasktarget.name()` is diagnostic only; dispatch never uses the user-authored name string.

26.7 The `.taskwork` adapter

An advanced work class implements:

```
run: method = .channelvalue
  arg request = .channelvalue, context = .taskcontext
```

For example:

```
adderwork: class implements .taskwork
  _delta = .int

  *: factory
    arg delta = .int
    _delta = delta
    return

  run: method = .channelvalue
    arg request = .channelvalue, context = .taskcontext
    if context.cancellation_requested() then return .channelvalue.
        null_value()
    return .channelvalue.integer_value(request.as_integer() + _delta)

main: procedure = .int
  pool = .taskpool.local(1, 4)
  scope = .taskscope.collectall(pool, 60000)
  target = task .adderwork(2)
  child = scope.submit(target, .channelvalue.integer_value(40))
  completion = child.wait(-1)
  answer = completion.value().as_integer()
  call scope.finish()
  call pool.close()
  return answer <> 42
```

The complete example, including the process provider, is `../crexx_programming_guide/examples/concurrency_taskwork.crexx`.

The request above is already an application `.channelvalue`; the compiler does not wrap it in the internal typed-object argument marker. The taskwork object is constructed in the receiving execution from the target's factory arguments.

26.8 Recursion and nested waits

A task body calling the same task callable is a normal synchronous recursive call in that worker:

```
factorial: task = .int
  arg value = .int
  if value <= 1 then return 1
  return value * factorial(value - 1)
```

It does not create a child task. A call from one task body to a different task callable, including a mutual-recursion edge, is rejected with `#TASK_NESTED_WAIT`. This prevents a bounded worker from blocking while waiting for work that may require the same pool.

```
inner: task = .int
  return 1

outer: task = .int
  return inner()          /* #TASK_NESTED_WAIT */
```

26.9 Compile-time diagnostics

The following diagnostic names are the current stable identifiers for the implemented checks:

Diagnostic	Condition
`#TASK_ONLY_LEVELG`	task declaration or task call below Level G
`#TASK_TARGET_ONLY_LEVELG`	task-target expression below Level G
`#PARALLEL_ONLY_LEVELG`	statement or expression `DO PARALLEL` below Level G
`#TASK_EXPOSED_ARGUMENT`	`ARG EXPOSE` in a task signature
`#TASK_REFERENCE_TYPE`	reference argument or result
`#TASK_NONTRANSFERABLE_TYPE`	unsupported scalar, array, interface or class without an exact transfer contract
`#TASK_NONTRANSFERABLE_RECEIVER`	task-method receiver lacks the exact concrete transfer contract
`#TASK_DYNAMIC_TARGET`	target is a dynamic string/procedure value
`#TASKWORK_ADAPTER_REQUIRED`	factory target's class does not implement `.taskwork`
`#TASK_NESTED_WAIT`	one task body calls a different task callable
`#PARALLEL_SCOPE_TYPE`	`USING` does not yield `.taskscope`
`#PARALLEL_SCOPE_REUSED`	one consumed scope is used by another parallel block
`#PENDING_TASK_RESULT_MUTATION`	pending binding is reassigned, referenced, mutated or escaped
`#TASK_RESULT_CYCLE`	the static task-result dependency graph contains a cycle

26.10 Current exclusions

The language does not expose a future type, `async`/`await`, detached task, thread identity, affinity, locks, atomics or shared writable VM objects. Services and `.taskscope.ask()` are reserved for a later single-owner state model. Pool telemetry and cross-host providers are also not part of the current language surface.

Library interfaces may still use the implemented task-target form. For example, `.httpserver.serve(task .handler())` receives a statically sealed `.taskwork` factory target and launches independent request tasks. That does not create a language service identity or make `.taskscope.ask()` available; the distinction is specified in the HTTP client and server guide.

27 Procedures and Arguments

27.1 Procedure Boundaries

A Level B procedure body continues until the next top-level procedure, class, interface, method, factory, or match boundary. There is no separate `END` instruction for a procedure body.

This matters when a file has declaration procedures before executable script code. Once the compiler sees an explicit top-level procedure, later statements belong to that procedure until the next callable boundary; they do not start a new implicit `main()`.

Use an explicit `main` when a script also needs a declaration procedure:

```
layout: procedure expose FLAG
  constant FLAG = 1
  return

main: procedure = .int
  say FLAG
  return 0
```

27.2 Procedure-Level Expose

`procedure expose` is the local procedure form for sharing module-global state:

```
main: procedure
  call proc1
```

```
  call proc2
  say "but var in main is" var
  return

proc1: procedure = .void expose var
  var = "Hello World"
  return

proc2: procedure = .void expose var
  say "var is" var
  return
```

Here `proc1` and `proc2` share the exposed global `var`. `main` does not list `var`, so its bare `var` reference is not the same exposed variable. To let `main` read or write the shared value, declare `main: procedure expose var` as well.

The `expose` list follows the return type if a return type is present. The items in the procedure-level list are bare variable names:

```
worker: procedure = .int expose state errors
```

For globally published module variables, prefer file-level `namespace name expose var`; those namespace-exposed globals auto-bind into procedures in the same source file.

27.3 Function Arguments

Arguments can be passed to a procedure by reference or by value. When an argument is passed by reference, the procedure can modify the original variable that was passed to it. When an argument is passed by value, a copy of the variable is passed to the procedure, and any changes made to the copy do not affect the original variable.

The user-visible rules are:

- Plain `ARG name = type` is pass by value.
- `ARG expose name = type` is pass by reference.
- Pass-by-value semantics are defined by caller visibility, not by the VM calling convention. If the callee writes to a by-value formal, the caller must still observe its original value after the call.

- This applies equally to simple values, arrays, and class/object references. Rebinding or mutating a by-value formal must not leak back to the caller's variable.
- The compiler is allowed to optimise away an internal defensive copy only when that cannot change caller-visible behaviour, for example when the formal is provably read-only or when the actual value is a temporary expression that has no caller-side symbol to preserve.
- If the caller wants the callee to update the original variable, the parameter must be declared with `expose`.

By example:

```
ARG a1 = 0, a2 = .int, expose a3 = .aclass, ?a4 = .aclass, a5 = .string
    \[\]
```

- Arg a1 is an optional integer (and 0 if not specified in the call)
- Arg a2 is a mandatory integer (pass by value)
- Arg a3 is a mandatory class aclass pass by reference
- Arg a4 is an optional class aclass pass by value; the default expression is the bare typed class value `.aclass`, not a factory call
- Arg a5 is an array of strings and is one way to allow an arbitrary number of strings to be passed to the procedure (see also Ellipsis later)

Optional defaults evaluate exactly as written. Use `?x = .SomeClass` for the bare typed class value and `?x = .SomeClass()` to call the default factory.

Examples:

```
bump: procedure = .int
  arg value = .int
  value = value + 1
  return value

x = 10
say bump(x)
say x           /* still 10 */

bumpref: procedure = .void
  arg expose value = .int
```

```
  value = value + 1
  return

x = 10
call bumpref(x)
say x           /* now 11 */
```

27.4 Ellipsis (...)

The last arguments declaration can be an ellipsis ('…'), this is used to show that 0 or more arguments can be provided. For example:

```
ARG a1 = 0, a2 = .int, ... = .string
```

- The '…' shows that an arbitrary number of .string arguments can be added to the end of the call.
- The ? operator exist to access & query arguments:
- ?a1 returns true if the optional arg a1 was specified.
- ?a2 will always be true as a2 is not optional.

Pseudo Array arg allows access to the '…' arguments. Also see the Arrays section.

- arg[1] or arg.1 gives the first '…' argument. These can signal `OUTOFRANGE`
- arg[0], arg[], arg.0 or arg. return the number of '…' arguments
- In a procedure without a `...` tail, the count forms return `0`

The type of this Pseudo is the type of the '…' argument

27.5 arg() Operator

The compatibility arg() operator is designed to provide some compatibility with Classic Rexx; by example:

- arg() is equivalent to arg.0 etc. Type Integer.

- arg(1) is equivalent to arg.1 etc. The type of this operator is the same as the '…' argument and like arg.1 can signal `OUTOFRANGE`

- arg(4,E), arg(4,"E"), arg(4,Exxx), arg(4,"Exxx") etc. all return 1 (true) if there were 4 or more '…' arguments given or 0 (false) otherwise. E is Exists.

- Likewise arg(4,'O') etc. (O is Omitted) is equivalent to ~arg(4,'E').
- In a procedure without a `...` tail, `arg()` returns `0` and the `E/O` probe forms operate on that empty tail

27.6 Implicit Main Procedure

In the event that a module file contains instructions preceding a `PROCEDURE` instruction, an implicit procedure named main() is automatically generated within the namespace of the module file. The arguments for this procedure can be accessed through the pseudo array arg or arg() operator. This implicit main() case is the compatibility bridge that maps classic `arg(n)` access onto command-line arguments when no explicit signature is present. The return type of the implicitly defined main() procedure is automatically set to either int or void.

28

Standard Library

The CREXX standard library is built from a mix of Level B source, rxas, and native support. The release model is explicit: source imports name the namespace to use, and runtime images or linked artifacts provide the bytecode and native pieces needed by the VM. The complete content of the library is documented in the *Library Reference*; here is a short impression of how to use it and what it offers in addition to the classic set of built-in functions.

28.1 Core Level B Library: `rxfnsb`

`rxfnsb` is the Level B built-in-function library. It contains many REXX-familiar functions such as string, numeric, date/time, argument, and conversion helpers. It also contains the supported Level B array helper surface: `arrayinsert`, `arraydelete`, `arrayappend`, `arrayprepend`, `arraypop`, `arrayshift`, `arrayget`, `arrayset`, `arraycontains`, `arrayindexof`, `arrayreverse`, `arrayjoin`, and the older copy, move, sort, format, dump, find, high-water, and drop helpers. New code should prefer these standard BIFs over the deprecated native arrays plugin.

`rxfnsb` also exposes the `.stem` class for classic REXX compound-variable style string-to-string keyed data. Stems support dotted tails and bracket keys, for example `s.name` and `s["customer.id"]`.

Use it explicitly in reusable Level B source:

```
options levelb
import rxfnsb
```

```
say date("w")
say length("hello")
```

The `crexx` driver imports `rxfnsb` automatically only for headerless top-level scripts.

Many `rxfnsb` functions are written in CREXX itself under `lib/rxfnsb/rexx/`. Low-level functionality is provided through RXAS or native runtime support where needed. The mutating array helpers use VM array attribute instructions for insert, delete, shrink, and clear operations, so common list-like operations can adjust the pointer array without a Rexx-level per-element copy loop.

28.1.1 Testing and Debugging REXX BIFs

The library build is a bootstrap build, not an ordinary application build. Most REXX BIF source files in `lib/rxfnsb/rexx/` are compiled with compiler exits disabled (`rxc -x`). That means certified-exit statements such as `TRACE`, `PARSE`, and `ADDRESS` are rejected in those files during the BIF build with `#CERTIFIED_EXIT_DISABLED`.

Do not add `TRACE RESULTS` directly to a BIF source file to debug it. Use one of these routes instead:

- Write or extend a functional test under `lib/rxfnsb/tests_functional/` that calls the BIF from normal REXX code.
- Build a small scratch program with exits enabled, import `rxfnsb`, and call the BIF under `TRACE R`, `TRACE I`, `TRACE ASM`, or `TRACE LLM`.
- If the trace needs to include standard-library frames, add `TRACE UNSUPPRESS NAMESPACE rxfnsb` before the call. The default TRACE filter hides `rxfnsb` and `_rxsysb` so ordinary user traces are not dominated by runtime-library internals.
- For native or linked-image debugging, keep source/TRACE debug metadata in the linked image. The `crexx -native` driver strips that metadata by default; use `--link-keep-source` for a debuggable linked intermediate.

The functional BIF test target is `testbifs`, and the individual tests are named `ts_*_noopt` and `ts_*_opt`. The system plugin smoke test is named `test_system`.

28.2 JSON, Sockets, and HTTP

The Level B library includes small, stable building blocks for integration work:

- `rxjson`: string-oriented JSON validation, path lookup, quoting, arrays, and objects
- `rxsocket`: VM-backed TCP sockets with optional TLS depending on build configuration

The private Level B `_rxhttpcore` module provides binary HTTP framing, parsing and codecs for higher layers; it is not a user client. User-facing HTTP belongs to `rxfnsg`: its initial Level G client provides pooled task requests, policy, buffered content decoding and bounded response streaming, while its bounded server dispatches complete request values to task classes. The Level G LLM providers use the same client and private backend. See Concurrent HTTP client and server for the complete surface and examples.

28.3 Binary SHA-256

The standard/default native provider `rx_hash` exposes one-shot, canonical-lowercase-hexadecimal, immutable incremental-state, and synchronous bounded-memory file SHA-256 procedures to both Level B and Level G. Binary digest results are 32 raw bytes; hexadecimal results are 64 lowercase characters. The versioned serialized state is pointer-free and transferable, and malformed states signal before hashing. Import `rxhash` directly; no Rexx wrapper is needed. The compiler records the native provider dependency, ordinary VMs discover the installed provider, and native packaging selects its static archive automatically. See Binary hashing with `rxhash` for every exact signature, state layout, file/error contract, and examples.

28.4 Mathematics

The Level G standard mathematics family separates native scalar `rxfloat`, checked `rxint`, and precision-preserving `rxdecimal`. Provider metadata makes the native float library automatic for interpreted and native packaging; the integer and dec-

imal modules are Level-B-authored library code. See Mathematics for the complete surface and availability boundary.

28.5 Packed vectors

The Level G standard/default `rxvector` native provider supplies explicit `f32le` conversion, exact cosine similarity, and deterministic bounded top-k search over `.packedfloat` and `.packedint` owners. Portable persistence stays separate from host-native computation. See Packed vectors.

28.6 SQLite

The standard `rxsqlite` native provider exposes typed SQLite connections, prepared statements, exact NULL/int64/real/text/blob access, WAL and bounded maintenance operations. Provider metadata makes the dynamic plugin automatic for the VMs and selects its bundled static archive for `crexx --native`. `rxsqlite_address` is a separate installed Level G `ADDRESS SQLITE` façade over the same provider, not a second driver. See SQLite with `rxsqlite`.

28.7 ADDRESS and Trace Support

`_address.rexx` contains the Rexx-side ADDRESS protocol support used by command dispatch, redirects, sandboxes, function calls, and host-variable binding helpers.

`trace.rexx` contains the runtime trace/debugger support used by `TRACE` and by the experimental debugger work. User-facing traces suppress standard library, compiler-exit, runtime-support, and debugger namespaces by default, while `TRACE SUPPRESS NAMESPACE`, `TRACE UNSUPPRESS NAMESPACE`, and `TRACE RESET NAMESPACES` let a debugging session adjust that filter.

`rxdb` remains experimental for the Release 1 beta line. Treat it as a smoke-tested debugging aid, not as a supported full debugger contract.

28.8 Class Library

`classlib` is loaded by the `crexx` driver by default and is part of the beta surface. Its initial concurrency classes include `.taskpool`, `.taskscope`, `.task`, `.completion`, `.tasktarget`, `.taskwork`, `.channel`, `.channelvalue`, `.byteendpoint` and `.transferbuffer`. They provide explicit control beneath Level G syntax while preserving one provider-neutral contract. See Concurrency classes for lifecycle rules and examples. Unsupported telemetry and service declarations fail explicitly; they do not return invented values.

28.9 RexxScript

RexxScript is delivered as a first-class runtime product with `bin/rexxscript.rxbin` and the standalone `bin/rexxscript` runner. It is an interpreted, sandboxed Rexx-family scripting surface for rules and generated code execution, not the Level C compiler path.

The product documentation lives with the runtime source:

- RexxScript user guide
- RexxScript developer guide

28.10 Level G Libraries

`rxfnsg` contains the LLM client modules used by demos and the initial concurrent HTTP client/server implementation. Level G task/parallel syntax and the classlib concurrency surface are implemented and tested on the current development baseline, but they are not the stable Level B contract. Start with the concurrent programming guide, which includes complete checked examples and explains when ordinary work stays on the controlling execution.

The `rxunicode` module supplies Unicode 17.0.0 normalization and predicates, full default upper/lower conversion, full/simple/Turkic case folding, default extended grapheme-cluster operations, and strict-by-default typed codecs between `.string` and `.binary`. Supported codecs include UTF-8, explicit-endian UTF-16/UTF-32, ASCII, ISO-8859-1, Windows-1252, IBM437, IBM850, and IBM1047. It does not

change ordinary `.string` equality or codepoint indexing. Level B `readbinary` and `writebinary` provide complete-file byte I/O for explicit codec composition. See Unicode text services for the exact surface and its relationship to TUTOR.

28.11 Level L Generated-Output Work

`rxfnsl` contains early Level L language-engineering examples. The first module is `tinyexpr`, a tiny arithmetic lexer/parser written in the shape that a future generator might emit. It uses packed binary constants, fixed-size binary token records, an exposed declaration procedure for token/layout constants, direct `<at..type>` reads/writes, and zero-copy source-slice compare to test whether the binary-memory surface is pleasant enough for generated language tooling.

This is intentionally not a public parser-generator API yet. Its job is to teach the project what emitted Rexx/RXAS should look like before deciding whether to port re2c, alter a generator backend, or design a narrower Level L generator.

28.12 Native Plugins

Native functions use the RXPA plugin architecture. Dynamic plugins can be loaded by the VM, and selected plugins can be statically linked into packaged executables depending on build configuration.

From Level B source, calls look the same whether the implementation is written in cRexx, rxas, or native code. The import and runtime library path decide which module or plugin is available.

29

MSAY and FMTMASK

29.1 Overview

`fmtmask` is a lightweight picture-mask formatter. It combines literal text with fixed-width text and numeric fields, making it convenient for producing aligned messages, summaries, and simple reports. Its mask notation is inspired by the picture clauses and report-writing facilities of COBOL, but provides a much smaller and simpler set of features.

The formatter is available in two forms. The `fmtmask()` function returns the formatted text to the program:

```
line = fmtmask(mask, value-1, value-2, ...)
```

The `MSAY` compiler-exit statement uses the same formatting engine and writes the resulting line immediately:

```
MSAY mask, value-1, value-2, ...
```

The choice between them depends on what must happen to the result. `fmtmask()` is appropriate when the text must first be stored, modified, logged, returned, or passed to another routine. `MSAY` provides a more concise form when the only purpose of the formatted text is to display it.

For example:

```
line = fmtmask(
    "Name: XXXXXXXXXX  Price: $$$$9.99  Qty: 999",
    "Fred",
```

```
    64.31,
    12
)

say line
```

produces:

```
Name: Fred        Price:   $64.31  Qty:  12
```

The same operation can be written with `MSAY`:

```
MSAY "Name: XXXXXXXXXX  Price: $$$$9.99  Qty: 999", ,
     "Fred", 64.31, 12
```

This is equivalent to:

```
say fmtmask(
    "Name: XXXXXXXXXX  Price: $$$$9.99  Qty: 999",
    "Fred",
    64.31,
    12
)
```

29.2 How a Mask Is Applied

A mask is a character string containing literal text and one or more field descriptions. Literal text is copied to the result unchanged. A sequence of `X` characters describes a text field, while a numeric field is described by `9` characters, optionally combined with a decimal point and a leading currency symbol.

The formatter scans the mask from left to right. Whenever it encounters a field, it takes the next value argument, formats that value according to the field description, and inserts the result. The values therefore correspond to fields by position rather than by name.

Consider the following mask:

```
mask = "Name: XXXXX  Qty: 999  Price: 999.99"
```

When it is used with these values:

```
"Tea", 7, 12.45
```

the fields and values are associated as follows:

```
XXXXX   -> "Tea"
999     -> 7
999.99  -> 12.45
```

The words `Name`, `Qty`, and `Price`, together with their punctuation and spacing, are literal text. They appear in the result exactly as written in the mask. Only the three picture fields consume values.

This positional processing makes masks compact and predictable. It also means that the order and number of values must agree with the fields in the mask. Changing the order of the fields requires the corresponding values to be reordered as well.

29.3 Text Fields

A contiguous sequence of `X` characters defines a text field. The number of characters in the sequence determines the width of the field:

```
XXXXXXXXXX
```

This mask describes a text field ten characters wide. Text is aligned on the left. A value shorter than the field is padded with spaces on the right, while a value longer than the field is truncated to the available width. The mask therefore fixes both the position at which the text begins and the amount of space it may occupy.

For example, formatting `Fred` with a ten-character field produces:

```
Fred
xx */
```

The displayed text is followed by six spaces. Those spaces may not be visible on their own, but they ensure that any later fields begin at a fixed position. This is particularly useful for columns:

```
MSAY "Name: XXXXXXXXXX  Department: XXXXXXXX", "Fred", "Sales"
```

If a value does not fit, it is shortened rather than allowed to disturb the remainder of the line. Thus, a ten-character field containing `Christopher` produces the first ten characters:

```
Christophe
```

29.4 Numeric Fields

Numeric fields are right-aligned so that values of different lengths line up naturally in a column. Spaces are inserted on the left when a value occupies less than the available width. Unlike text fields, numeric fields are not silently truncated. If a numeric result cannot fit, the complete field is replaced by asterisks, making the error visible in the output.

29.4.1 Integer Fields

A contiguous sequence of `9` characters without a fractional part defines an integer field. For example:

```
999
exx */
```

defines a field three characters wide. The value is converted using `TRUNC()` and then aligned on the right. Formatting the value `12` consequently gives:

```
 12
exx */
```

The leading space occupies the unused position. Because `TRUNC()` is used, this form discards any fractional part rather than rounding it for display.

If the value is too large for the field, the field is filled with asterisks:

```
Mask   : 999
Value  : 12345
Result : ***
```

29.4.2 Fixed-Decimal Fields

A numeric mask containing a decimal point followed by one or more `9` characters defines a fixed-decimal field:

```
9999.99
*/
```

The number of `9` characters after the decimal point determines the number of decimal places in the result. Formatting is performed using `FORMAT()`, after which the result is aligned on the right within the complete field.

For example:

```
Mask   : 9999.99
Value  : 64.31
Result :   64.31
```

The decimal point is part of the field width. In the preceding example the complete field is seven characters wide: four positions before the point, the point itself, and two positions after it. The mask fixes the displayed precision as well as the alignment.

A decimal point is recognized as part of a numeric field only when it is followed by at least one `9`. Consequently:

```
999.99
 */
```

is one fixed-decimal field, whereas:

```
999.
xx */
```

is an integer field followed by a literal period. This distinction permits ordinary punctuation to be placed directly after an integer field.

29.4.3 Currency Fields

A numeric field may be preceded by repeated currency characters. The following symbols are supported:

```
$€

£
¥
```

For example:

```
$$$$9.99
/
```

describes a currency field whose total width includes all the currency positions, the digit position, the decimal point, and the fractional positions. The numeric value is formatted normally, after which one currency symbol is placed immediately before it. Unused leading positions remain spaces:

```
Mask   : $$$$9.99
```

```
Value  : 64.31
Result :   $64.31
```

Repeating the currency character does not cause the symbol itself to be repeated in the output. It reserves sufficient room for the symbol and for the value to move within the field while remaining right-aligned.

For example:

```
MSAY "Total: $$$$$9.99", 123.45
```

produces:

```
Total:   $123.45
```

Currency formatting is intentionally independent of locale. The symbol is taken directly from the mask; the formatter does not select a currency, change the decimal separator, or apply national grouping conventions.

29.5 Overflow Handling

Numeric information should not be made misleading merely to preserve the layout of a line. For this reason, a numeric value that does not fit is not truncated. Instead, every position in its field is replaced by an asterisk:

```
Mask   : 999
Value  : 12345
Result : ***
```

The same rule applies to fixed-decimal and currency fields:

```
Mask   : $$$9.99
Value  : 123333.3
Result : *******
```

Asterisks preserve the width of the report while drawing attention to the fact that the selected mask was too narrow. The program can correct such an overflow by increasing the field width or by otherwise constraining the value before it is formatted.

Text fields behave differently: an overlong text value is truncated to the field width. This is appropriate for labels and descriptions, where retaining the layout is generally preferable to treating the value as a numeric error.

29.6 Combining Fields in a Line

The principal benefit of picture masks becomes apparent when several fields are combined. Literal text supplies headings and punctuation, while each field reserves a predictable amount of space:

```
MSAY "Item: XXXXX  Qty: 999  Price: 999.99",
     "Tea", 7, 12.45
```

This produces:

```
Item: Tea    Qty:   7  Price:  12.45
```

Additional calls using the same mask retain the same column positions even when the values have different lengths. A mask can therefore be kept in a variable and reused for every detail line in a report:

```
detailMask = "Item: XXXXXXXXXX  Qty: 999  Price: $$$$9.99"

MSAY detailMask, "Tea",    7, 12.45
MSAY detailMask, "Coffee", 2, 64.31
```

Using one mask for a group of related lines also keeps their presentation consistent and makes later changes to the layout easier.

29.7 Choosing Between MSAY and FMTMASK

`MSAY` is intended for direct output. It expresses formatting and display as one operation and is consequently well suited to diagnostic messages, headings, and report lines:

```
MSAY "Customer: XXXXXXXXXX  Balance: $$$$$9.99",
     customer,
     balance
```

Use `fmtmask()` when the resulting string has a life beyond that immediate output operation. It can be assigned to a variable, passed to another routine, written through a logging interface, combined with other text, or returned to a caller:

```
line = fmtmask(mask, values...)
call logfile line
say line
```

Both forms use the same mask language and produce the same formatted text. The distinction is therefore one of program structure rather than formatting capability.

29.8 Scope of the Facility

`fmtmask` is designed as a small, practical facility for common fixed-width formatting tasks. It provides:

- fixed-width, left-aligned text fields;
- right-aligned integer and fixed-decimal fields;
- simple currency presentation;
- visible numeric-overflow handling; and
- a compact notation suitable for report-style output.

It is not intended to implement the complete COBOL `PIC` language or to act as a general report writer. In particular, it does not provide locale-aware currency conversion, national decimal and grouping conventions, or the range of editing characters and sign-placement rules found in more extensive formatting systems.

In this way, a mask remains easy to read beside the values it formats, and the result can usually be understood without consulting a large set of picture-string rules. For output that needs more sophisticated localization or presentation logic, a specialized formatting library probably remains the more appropriate choice.

30 FSAY and FSAYFMT

30.1 Overview

FSAY provides formatted string interpolation. It allows variable names and their formatting instructions to be written directly in a template, close to the surrounding text:

```
name = "Fred"
qty = 12
price = 64.31

FSAY "Name: {name:<10} Qty: {qty:>3} Price: {price:8.2}"
```

The result is:

```
Name: Fred       Qty:  12 Price:    64.31
```

Unlike a runtime interpolation facility, FSAY is a compiler-exit statement. The compiler exit transforms the template into an ordinary CREXX string expression and places that expression after SAY. The generated statement is then compiled in the normal way.

Conceptually, the preceding statement becomes:

```
say 'Name: ' || left(name,10) ||,
    ' Qty: ' || right(qty,3) ||,
    ' Price: ' || format(price,8,2)
```

This distinction is important. The template is analysed while the source program is being compiled, but the variables are evaluated when the generated statement

runs. FSAY therefore combines concise template notation with the normal name lookup, expression evaluation, and formatting functions of cRexx.

The transformation is performed by the Level B helper `fsayfmt`. That helper returns CREXX source text; it does not itself evaluate the variables or produce the final output string.

30.2 The FSAY Statement

The statement has the following form:

```
FSAY template
```

The template is a quoted string containing literal text and optional placeholders. Literal text appears in the output unchanged. Each placeholder names a variable whose value is inserted at that position. A placeholder can also specify a field width, alignment, or decimal precision.

A simple placeholder performs interpolation without imposing a field width:

```
name = "Fred"
FSAY "Hello, {name}"
```

This produces:

```
Hello, Fred
```

At compilation time, the compiler exit passes the quoted template source token to `fsayfmt`. The helper decodes one matching pair of outer quotes, scans the template, and constructs a CREXX expression from its literal and variable parts. The returned expression is inserted after SAY and compiled as part of the program.

Consequently, FSAY does not capture the value of a variable at compilation time. If a variable changes between two executions of the statement, the current value is used each time:

```
status = "starting"
FSAY "Status: {status}"

status = "complete"
FSAY "Status: {status}"
```

30.3 Placeholders

A placeholder is enclosed in braces. Its general form is:

```
{variable[:alignment][width][.decimals]}
```

Only the variable name is required. The remaining parts determine how its value is presented:

- `alignment` is <, >, or ^;
- `width` is a non-negative decimal integer; and
- `decimals` is a non-negative decimal integer following a period.

The colon separating the variable name from its format is optional when whitespace is used instead:

```
{name:<10}
{name <10}
```

Both forms describe the same ten-character, left-aligned field. The colon form is usually more compact, while the whitespace form may be more readable in dense templates.

The variable must be an identifier or compound variable. A placeholder is not a general CREXX expression: operators, function calls, and array subscripts cannot be placed inside it. Values requiring prior calculation should be assigned to a variable before the `FSAY` statement:

```
total = price * quantity
FSAY "Total: {total:10.2}"
```

Keeping placeholders restricted to names makes their parsing unambiguous and keeps templates readable.

30.4 Field Width and Alignment

When no format is present, the variable is inserted directly:

```
{name}
 */
```

Conceptually, this contributes `name` to the generated string expression. No padding or truncation is requested by the placeholder itself.

A width reserves a field of the specified number of characters. A plain width uses left alignment:

```
{name:10}
```

This generates:

```
left(name,10)
```

The same alignment can be stated explicitly with <:

```
{name:<10}
```

Right alignment uses >:

```
{qty:>3}
/
```

which generates:

```
right(qty,3)
```

Centred text uses ^:

```
{heading:^20}
```

which generates:

```
center(heading,20)
```

The alignment character therefore maps directly to the familiar CREXX string functions:

```
<   left
>   right
^   center
```

An explicit alignment character requires a width. Alignment has no useful meaning without a field in which the value can be positioned, so forms such as `{name:<}` are invalid.

30.4.1 Alignment Examples

The following placeholders illustrate the generated expressions:

Placeholder	Generated expression
`{name}`	`name`
`{name:10}`	`left(name,10)`
`{name:<10}`	`left(name,10)`
`{name:>10}`	`right(name,10)`
`{name:^10}`	`center(name,10)`

For example, a common field width can be used to align a sequence of names:

```
FSAY "Name: {name:<12} State: {state}"
```

The text following the name begins at the same position regardless of the length of the current name, subject to the behaviour of `LEFT()`.

30.5 Decimal Formatting

A decimal precision is introduced by a period. It requests numeric formatting through `FORMAT()`:

```
{price:8.2}
```

This generates:

```
format(price,8,2)
```

Here, `8` is the field width and `2` is the number of digits after the decimal point. Decimal formats default to right alignment, which is the customary presentation for numeric columns.

The width may be omitted while retaining the decimal precision:

```
{price:.2}
```

This generates:

```
format(price,,2)
```

The omission leaves the overall width to `FORMAT()` while still fixing the number of decimal places.

Explicit alignment can be combined with decimal formatting. Right alignment uses the width directly in `FORMAT()`:

```
{price:>8.2}
```

generates:

```
format(price,8,2)
```

For left or centred presentation, the number is first formatted to the requested precision without an overall numeric width. The resulting text is then aligned:

```
{price:<8.2}
```

generates:

```
left(format(price,,2),8)
```

and:

```
{price:^8.2}
```

generates:

```
center(format(price,,2),8)
```

The complete set of decimal examples is therefore:

Placeholder	Generated expression
`{price:8.2}`	`format(price,8,2)`
`{price:.2}`	`format(price,,2)`
`{price:<8.2}`	`left(format(price,,2),8)`
`{price:>8.2}`	`format(price,8,2)`
`{price:^8.2}`	`center(format(price,,2),8)`

Widths and decimal precisions must be non-negative decimal integers. They are part of the template notation and are fixed when the template is expanded; they are not variable expressions evaluated at runtime.

30.6 Literal Text and Braces

All characters outside placeholders are literal template text. During expansion, `fsayfmt` turns each literal part into canonical single-quoted REXX source. Apostrophes in the template are doubled in the generated source so that they remain literal apostrophes when the expression is compiled. Unicode text is preserved.

Because single braces delimit placeholders, doubled braces are used when an actual brace must appear in the output:

```
FSAY "Set {{name}} to {value}"
```

If `value` contains `Fred`, the result is:

```
Set {name} to Fred
```

Here, `{{` produces a literal opening brace and `}}` produces a literal closing brace. The text between those doubled braces is not treated as a variable reference.

This escaping rule allows templates to include notation that already uses braces, while keeping an isolated `{name}` available for interpolation.

30.7 Errors in Templates

The structure of an `FSAY` template is checked while the compiler exit expands the statement. Malformed templates therefore fail during compilation rather than producing partially interpolated output at runtime.

The following conditions signal `INVALID_ARGUMENTS`:

- an unmatched opening or closing brace;
- a nested opening brace;
- an empty placeholder;
- an invalid variable name;
- an empty format specification;
- an invalid field width;
- an invalid decimal precision; or
- an alignment character without a width.

Examples of invalid forms include:

```
{}
{name:}
{name:<}
{name:ten}
{price:8.two}
```

These checks protect the generated source expression from ambiguous or incomplete template notation. They also make errors local to the `FSAY` statement that contains them.

30.8 Direct Use of FSAYFMT

The typed Level B interface is:

```
fsayfmt(template = .string) = .string
```

A direct call supplies decoded template text and receives a CREXX source expression. For example:

```
fsayfmt("Hello {name}")
```

returns source equivalent to:

```
'Hello '||name
```

Similarly:

```
fsayfmt("{name:10}")
```

returns:

```
left(name,10)
```

This return value is source code, not the formatted result of evaluating `name`. Direct use is consequently most relevant to compiler exits, source generation, testing, and other tooling that needs to construct a CREXX expression.

For compiler-exit use, one matching outer quoted source token is decoded before the template is scanned. `fsayfmt` then makes a single scan over the decoded template. It does not invoke the general quote-aware library.

30.9 Choosing Between FSAY and MSAY

`FSAY` and `MSAY` both produce formatted output, but they express the layout in different ways.

`FSAY` uses named placeholders:

```
FSAY "Customer: {customer:<12} Balance: {balance:10.2}"
```

The name of each value appears at the point where the value is inserted. This makes the template easy to read when it contains descriptive text, when values appear in a different order from their declarations, or when a value appears more than once.

`MSAY` and `fmtmask` use picture fields and consume separately supplied values from left to right:

```
MSAY "Customer: XXXXXXXXXXX Balance: 9999999.99",
     customer, balance
```

This style is particularly useful for traditional fixed-width reports in which the shape of each column is more important than the name of the value.

As a practical rule, use `FSAY` for readable named interpolation and local formatting within prose-like messages. Use `MSAY` or `fmtmask` for picture-driven layouts and repeated report lines whose fields are naturally positional.

See the selector-level Level B contract for direct-call and test details.

31 File and Stream I/O

31.1 Overview

For input and output Rexx traditionally[14] provides a small set of built-in functions for reading and writing streams. The same functions can be used with ordinary files and with the standard input and output streams. They deliberately avoid exposing file descriptors, buffers, and other operating-system details to the program.

crexx Level B provides the familiar line-oriented functions:

- `LINEIN()` reads a line;
- `LINEOUT()` writes a line;
- `LINES()` reports whether another line is available;

and the corresponding character-oriented functions:

- `CHARIN()` reads one or more characters; and
- `CHAROUT()` writes text without adding a line terminator.

These functions form the standard stream-oriented part of the supported Level B File I/O API. crexx also provides an extended group of functions for more specific file-processing requirements; those functions are documented separately

[14] Except for the mainframe implementations on CMS and TSO where the product cutoff date precluded delivery of the stream I/O functions, and the PL/S stream implementation was available as an add-on (PRPQ). VM employed the EXECIO program for I/O, which was reproduced for the TSO/E implementation. The z/OS USS (Unix System Services) implementation of Rexx does include all stream I/O functions.

later in this chapter.

The implementation treats these facilities as UTF-8 text I/O. `LINEIN()` and `LINEOUT()` operate on text lines, and `CHARIN()` reads Unicode code points rather than arbitrary bytes. Programs that need to preserve arbitrary binary data should use the extended `READBINARY()` and `WRITEBINARY()` functions.

31.2 Streams and File Names

The first argument to each standard I/O function identifies a stream. For an ordinary file, this is its file name or path:

```
line = linein("customers.txt")
```

The special names:

```
stdin
stdout
stderr
```

refer to the standard streams of a process. The standard functions use `stdin` or `stdout` as appropriate when the file-name argument is omitted:

```
line = linein()          -- read from standard input
call lineout , line      -- write to standard output
```

A blank file name is treated in the same way as an omitted one. For input it selects `stdin`; for output it selects `stdout`:

```
line = linein("")
call lineout "", "ready"
```

The library manages the underlying open streams. A program normally names the same file on successive calls and lets the library retain its current position. An output or input stream can be closed explicitly through the output functions, as described under `LINEOUT()` and `CHAROUT()`.

31.3 Line-Oriented Input

Line-oriented input is the natural choice for text files in which records are separated by line terminators. The terminator identifies the end of the record, but it is not included in the value returned to the program.

31.3.1 LINEIN

`LINEIN()` reads the next line from a text stream.

```
line = linein([fileName])
```

The Level B signature is:

```
linein(fileName = "stdin") = .string
```

If `fileName` is omitted or blank, input is read from `stdin`. Otherwise, the named file is opened for reading when necessary and its current stream position is used. Standard input returns as soon as its line terminator is read; `linein` does not wait for or consume a character from the next line.

For example:

```
name = linein("names.txt")
say "First name:" name
```

Successive calls read successive lines:

```
do while lines("names.txt") > 0
    say linein("names.txt")
end
```

The returned string does not contain the line terminator. An empty physical line is therefore returned as an empty string. End of file can also yield an empty string, so a program that must distinguish an empty line from the end of the stream should use `LINES()` to test availability.

The CREXX Level B form implements sequential line input. Unlike some extended REXX stream interfaces, it does not accept a starting line number or a count; each call reads one line at the current position.

If the file cannot be opened, `LINEIN()` writes an error diagnostic. Programs that require explicit and structured open-error handling can instead use `READLINES()` with its error option.

31.3.2 LINES

`LINES()` reports whether a line is available from a text stream.

```
available = lines([fileName])
```

The Level B signature is:

```
lines(fileName = "stdin") = .int
```

It returns:

```
 1   at least one line is available
 0   the stream is at end of file
-1   the stream could not be opened
```

`LINES()` is therefore an availability test, not a count of all remaining lines. A positive result means that one subsequent `LINEIN()` call can obtain a line; it does not promise that exactly one line remains.

A conventional file-reading loop is:

```
file = "report.txt"

do while lines(file) > 0
    line = linein(file)
    say line
end

call lineout file
```

The final `LINEOUT()` call closes the named stream. Although this looks like an output operation, the omitted second argument gives it the traditional stream-closing meaning.

To determine whether data is available without losing it, the implementation may read one line ahead. That line is retained internally and returned by the next `LINEIN()` for the same file. Repeated calls to `LINES()` therefore do not skip records:

```
if lines(file) > 0 then do
    if lines(file) > 0 then
        say linein(file)
end
```

Only one availability value is needed because the prefetched line remains pending until `LINEIN()` consumes it or the stream is closed.

31.4 Line-Oriented Output

31.4.1 LINEOUT

`LINEOUT()` writes one line of text and appends a line terminator.

```
result = lineout([fileName], line)
```

The Level B signature is:

```
lineout(fileName = "stdout", line = "") = .int
```

The presence of the second argument is significant. If `line` is supplied, its value is written and an LF line terminator, byte `0A` hexadecimal, is appended:

```
call lineout "messages.txt", "first message"
call lineout "messages.txt", "second message"
```

An explicitly supplied empty string writes an empty line:

```
call lineout "messages.txt", ""
```

If the `line` argument is omitted, nothing is written. Instead, the named stream is closed:

```
call lineout "messages.txt"
```

This distinction between an omitted argument and an empty argument is important:

```
call lineout file, ""     -- write an empty line
call lineout file         -- close the stream
```

When the file name is omitted or blank, `stdout` is used:

```
call lineout , "Processing complete"
```

`LINEOUT()` flushes the stream after writing a line and checks both the flush status and the stream error status. It returns zero after a successful write or close. If the file cannot be opened, it writes a diagnostic and returns `1`. A non-zero error reported while flushing or checking the stream is returned to the caller.

A complete writing sequence can therefore test each operation:

```
file = "summary.txt"

rc = lineout(file, "Summary")
if rc = 0 then rc = lineout(file, "-------")
if rc = 0 then rc = lineout(file)

if rc <> 0 then
    say "Unable to write" file", return code" rc
```

31.5 Character-Oriented Input

31.5.1 CHARIN

`CHARIN()` reads text from the current position in a stream without treating line terminators as record boundaries.

```
text = charin([fileName] [, count])
```

The Level B signature is:

```
charin(fileName = "stdin", count = 1) = .string
```

`count` specifies the maximum number of Unicode code points to read. Its default is one:

```
character = charin("input.txt")
```

Several characters can be read in one call:

```
prefix = charin("input.txt", 12)
```

The result may contain fewer code points than requested if the end of the stream is reached. A count less than or equal to zero returns an empty string without reading the stream:

```
text = charin("input.txt", 0)       -- returns ""
```

`CHARIN()` is character-oriented rather than byte-oriented. A UTF-8 character encoded with more than one byte still counts as one code point for the purpose of `count`. This makes the operation suitable for Unicode text but unsuitable for reading a precise number of bytes from a binary file.

The Level B form reads sequentially from the current stream position. It does not provide the optional absolute starting position found in some other Rexx implementations.

If the file cannot be opened, `CHARIN()` writes an error diagnostic and returns an empty string.

31.6 Character-Oriented Output

31.6.1 CHAROUT

`CHAROUT()` writes text without appending a line terminator.

```
result = charout([fileName], text)
```

The Level B signature is:

```
charout(fileName = "stdout", text = "") = .int
```

This function is useful when a line is to be assembled in several operations, or when the program needs to write a prompt and leave the cursor on the same line:

```
call charout , "Name: "
name = linein()
```

Text from successive calls is written contiguously:

```
call charout "result.txt", "part one"
call charout "result.txt", " and part two"
```

No separator or newline is inserted between the calls. A line terminator can be supplied explicitly as part of the text when needed, or the final part can be written with `LINEOUT()`.

As with `LINEOUT()`, omission of the second argument closes the stream:

```
call charout "result.txt"
```

An explicit empty second argument is different. It performs a zero-length write rather than requesting a close:

```
call charout "result.txt", ""
```

`CHAROUT()` returns zero after a successful write or close. If the file cannot be opened, it writes a diagnostic and returns `1`.

31.7 Closing Streams

The standard Level B interface does not expose a separate `CLOSE()` built-in function. A stream is closed by calling `LINEOUT()` or `CHAROUT()` with its data argument omitted:

```
call lineout file
```

or:

```
call charout file
```

Closing an input stream through `LINEOUT()` also discards any line that `LINES()` has read ahead for that file. A later input operation starts with a newly opened stream.

It is good practice to close a file when a program has finished using it, especially when:

- output must be made visible to another process;
- the same file will later be reopened in another mode;
- the program uses many different files; or
- an availability test may have retained a pending input line.

The standard streams can be named and closed in the same way, although a program normally leaves their lifetime to the process environment.

31.8 Mixing Line and Character Operations

Line-oriented and character-oriented functions use the same underlying file position, but they apply different interpretations to the data. Mixing them on one stream can therefore be difficult to reason about.

There is an additional consideration in the Level B implementation: `LINES()` may read a complete line ahead and retain it for `LINEIN()`. `CHARIN()` does not consume that pending line. If a program calls `CHARIN()` after a successful `LINES()` but before the corresponding `LINEIN()`, the character read begins after the internally retained line. A later `LINEIN()` then returns the earlier pending line.

For predictable sequential processing, use either:

```
LINES() with LINEIN()
```

or:

```
CHARIN()
/
```

for a particular stream at a particular stage of processing. Close and reopen the stream before changing models if its position must be reset.

The same general advice applies to `LINEOUT()` and `CHAROUT()`: combining them is valid when their different newline behaviour is intentional, but the program must account for exactly which operation supplies each line terminator.

31.9 Standard I/O Examples

31.9.1 Copying a text file

The following example copies a text file one line at a time:

```
source = "input.txt"
target = "output.txt"
rc = 0

do while lines(source) > 0
    rc = lineout(target, linein(source))
    if rc <> 0 then leave
end

call lineout source
close_rc = lineout(target)

if rc = 0 then rc = close_rc
if rc <> 0 then
    say "Copy failed, return code" rc
```

Because `LINEOUT()` supplies a new LF terminator for every input line, this is a text copy rather than a byte-for-byte copy. It normalizes the output line terminators to the Level B convention.

31.9.2 Reading a prompt

The standard streams allow terminal interaction without special console functions:

```
call charout , "Enter a file name: "
file = linein()

if lines(file) < 0 then
    say "Cannot open" file
else
    say "The first line is:" linein(file)

call lineout file
```

`CHAROUT()` writes the prompt without a newline, while `LINEIN()` reads the reply as a complete line.

31.10 Extended File I/O Functions

In addition to the standard file I/O operations, CREXX provides several functions for more specific file-processing requirements. These functions are part of the supported Level B File I/O API; they are grouped separately because they provide higher-level or more specialised operations rather than basic file access.

They can be used like any other File I/O function when their behaviour matches the requirements of an application.

31.10.1 Record I/O with EXECIO

`EXECIO`, included in the IBM Classic REXX implementations for VM/CMS and TSO/E, offers an IBM-mainframe style record-oriented syntax for transferring lines between a text file and a string array.

`EXECIO` is especially convenient when a program reads or writes an entire group of records.

31.10.2 Syntax

```
EXECIO record-count mode file-name (STEM stem-name FINIS
```

The operands are:

- `record-count` specifies the maximum number of records to transfer. A positive whole number requests that number of records. An asterisk requests all applicable records.
- `mode` specifies whether records are read, written, or appended.
- `file-name` identifies the text file.
- `STEM stem-name` identifies the string array that supplies or receives the records.
- `FINIS` requests completion of the operation and closure of the file.

The supported modes are:

```
DISKR    Read records from a file
DISKW    Write records, replacing the file
DISKA    Append records to the file
```

The longer names `READ`, `WRITE`, and `APPEND` are also accepted by the underlying Level B function.

31.10.3 Writing Records

With `DISKW`, the array supplies the records written to the file:

```
out_stem = .string[]
out_stem[1] = "Line 1: Hello from Execio"
out_stem[2] = "Line 2: Testing the exit"
out_stem[3] = "Line 3: Goodbye"

EXECIO 3 DISKW 'test_execio_temp.txt' (STEM out_stem FINIS
```

This writes the first three elements of `out_stem`. Existing contents of the file are replaced.

Records are separated by LF characters. The implementation does not append an additional line terminator after the final record, thereby avoiding the appearance of an extra empty record when the file is subsequently read.

When the record count is `*`, the number of records is obtained from the array's item count:

```
EXECIO * DISKW 'output.txt' (STEM out_stem FINIS
```

A numeric record count cannot cause more records to be written than the array contains.

To add records without replacing the existing file, use `DISKA`:

```
EXECIO * DISKA 'output.txt' (STEM additional_lines FINIS
```

31.10.4 Reading Records

With `DISKR`, records are read from the file into the named string array:

```
EXECIO * DISKR 'test_execio_temp.txt' (STEM mystem FINIS
```

An asterisk reads all records through the end of the file. A numeric count limits the number of records:

```
EXECIO 10 DISKR 'input.txt' (STEM first_ten FINIS
```

The first record is placed in element `1`, the second in element `2`, and so forth:

```
say mystem[1]
say mystem[2]
say mystem[3]
```

Line terminators are not included in the array elements. Empty physical lines inside the file are preserved, while a final line terminator does not create an additional empty element.

Before reading begins, the existing item count of the receiving array is reset. The resulting array contains the records obtained by the current operation.

31.10.5 Automatic Stem Declaration

The `EXECIO` compiler exit recognizes the name in the `STEM` clause. If the receiving stem has not previously been declared, the compiler exit hoists an appropriate declaration into the procedure:

```
EXECIO * DISKR 'input.txt' (STEM records FINIS
```

It is therefore not necessary to precede the statement with:

```
records = .string[]
```

The generated declaration gives `records` the `.string[]` type required by the underlying `_execio()` function. This is particularly useful for input, where the array exists primarily to receive records.

An explicit declaration remains valid and may make the role of an output array clear:

```
records = .string[]
records[1] = "first record"
records[2] = "second record"

EXECIO * DISKW 'output.txt' (STEM records FINIS
```

31.10.6 Complete Example

```
options levelb
import rxfnsb
```

```
main: procedure

out_stem = .string[]
out_stem[1] = "Line 1: Hello from Execio"
out_stem[2] = "Line 2: Testing the exit"
out_stem[3] = "Line 3: Goodbye"

EXECIO 3 DISKW 'test_execio_temp.txt' (STEM out_stem FINIS

/* mystem is declared automatically by the compiler exit. */
EXECIO * DISKR 'test_execio_temp.txt' (STEM mystem FINIS

say "Read back from file:"
say "1:" mystem[1]
say "2:" mystem[2]
say "3:" mystem[3]
say "SUCCESS"

return
```

Relationship to `_execio()`

`EXECIO` is source-level convenience syntax. The actual transfer is performed by the `_execio()` Level B library function. The compiler exit:

- parses the record count, mode, file name, and `STEM` clause;
- supplies a `.string[]` declaration when the stem is undeclared;
- constructs the corresponding `_execio()` call; and
- lets the resulting CREXX code be compiled normally.

The statement consequently retains the familiar appearance of IBM `EXECIO` while using typed CREXX arrays and the Level B file-I/O implementation underneath. It should be regarded as an IBM-style CREXX facility rather than part of the portable ANSI REXX built-in-function set.

31.10.7 The _EXECIO Built-in function

`_EXECIO` provides an IBM-style record interface for transferring a group of text lines between a file and a stem. It is used in the implementation of the above mentioned `EXECIO` facility, but can also be used as a built-in function by itself.

Its signature is:

```
_execio(maxRecords, mode, fileName, stem) = .int
```

The arguments are:

- `maxRecords`, a record count or `*`;
- `mode`, selecting input, replacement output, or append output;
- `fileName`, the file to process; and
- `stem`, the string array used to receive or supply records.

The accepted modes are:

```
DISKR    READ
DISKW    WRITE
DISKA    APPEND
```

Mode names are stripped and converted to uppercase. `DISKR` and `READ` open the file for reading. `DISKW` and `WRITE` open it for replacement writing. `DISKA` and `APPEND` open it for append writing.

31.10.8 Reading records

For input, `*` requests all remaining records. Otherwise, `maxRecords` gives the maximum number of records to read:

```
records = .string[]
count = _execio("*", "DISKR", "input.txt", records)
```

The records are placed in:

```
records[1]
records[2]
...
```

and the function returns the number read. The array's item count is reset before input begins. Line terminators are not stored in the array, physical empty lines are retained, and a final newline does not create an extra empty record.

31.10.9 Writing records

For output, `*` writes the number of records indicated by `stem.0`. A numeric maximum writes no more than the smaller of that value and `stem.0`:

```
records = .string[]
records[1] = "first"
records[2] = "second"

count = _execio("*", "DISKW", "output.txt", records)
```

The function inserts LF separators between records but does not append a newline after the last record. This prevents the reader from interpreting a final separator as an additional empty record.

The return value is the number of records processed. Failure to open the file returns `-12`; a non-zero close error is returned instead of the count.

31.10.10 Reading Several Lines with READLINES

`READLINES()` reads a selected range of a text file into a dense string array. It is convenient when the complete selection is to be processed in memory rather than one line at a time.

```
lines = readlines(fileName [, fromLine [, maxRecords [, errorAction]]])
```

The effective signature is:

```
readlines(fileName = .string,
          fromLine = 1,
          maxRecords = 0,
          errorAction = "raise") = .string[]
```

`fromLine` is one-based. Values smaller than one are adjusted to one. `maxRecords` limits the number of returned lines; zero means all remaining lines. Negative values are adjusted to zero.

For example, this reads at most 25 lines beginning with physical line 101:

```
part = readlines("large.txt", 101, 25)
```

The result is a dense array indexed from one:

```
part[1]    physical line 101
part[2]    physical line 102
...
```

Line terminators are removed. Empty physical lines inside the selected range are preserved, but a trailing newline at physical end of file does not create an additional empty array item.

By default, failure to open or close the file raises condition `error` with code `40.27`. If `errorAction` is not `RAISE`, an open failure instead returns an array whose first item is:

```
-8 OPEN ERROR
```

Read errors raised by the underlying line operation propagate to the caller. The file is closed before a successful return.

`READLINES()` holds the selected records in memory. Sequential `LINEIN()` is preferable for very large files when the program does not need the entire selection at once.

31.10.11 Loading Text with LOADTEXT

`LOADTEXT()` reads a complete non-binary file and combines its lines into one string.

```
text = loadtext(fileName [, delimiter [, errorAction]])
```

Its effective signature is:

```
loadtext(fileName = .string,
         delimiter = '0D0A'x,
         errorAction = "raise") = .string
```

Each physical line is returned without its original terminator and is followed by `delimiter` in the resulting string. The default delimiter is CRLF:

```
text = loadtext("chapter.txt")
```

A caller can request another separator, including LF:

```
text = loadtext("chapter.txt", '0A'x)
```

or combine the lines without any separator:

```
text = loadtext("chapter.txt", "")
```

Because the delimiter is appended after every line read, the returned string also ends with the selected delimiter when the file contains at least one line.

By default, failure to open or close the file raises condition `error` with code `40.27`. If `errorAction` is not `RAISE`, an open failure returns:

```
-8 OPEN ERROR
```

The routine loads the complete text into memory and should therefore be used only when the expected file size is appropriate.

31.10.12 Reading and Writing Binary Files

`READBINARY()` and `WRITEBINARY()` provide whole-file byte I/O using the `.binary` type. They do not perform UTF validation, newline conversion, or text encoding.

```
data = readbinary("input.dat")
written = writebinary("copy.dat", data)
```

`READBINARY()` returns the exact file contents, including embedded NUL and invalid UTF-8 bytes. It raises `NOTREADY` if the file cannot be opened, read, or closed. `WRITEBINARY()` replaces the target with the supplied bytes and returns the number of bytes written. Empty data creates or truncates a file to zero bytes; open, write, flush, or close failures raise `NOTREADY`.

These operations load or write the complete value, so the caller is responsible for the memory required by the data.

31.10.13 Emptying a File with ERASEFILE

Despite its name, `ERASEFILE()` does not delete a file. It truncates the file to zero length while retaining the directory entry:

```
result = erasefile(fileName)
```

Its signature is:

```
erasefile(fileName = .string) = .int
```

The function first closes any cached instance of the file. It then opens the file in replacement-write mode, which truncates an existing file, and closes it immediately.

The return values are:

```
 0   success
-1   the file could not be opened or created
```

If the file does not exist, a new empty file is created. The operation is similar to `OPEN WRITE REPLACE` in classic Rexx stream implementations.

For example:

```
if erasefile("trace.log") <> 0 then
    say "Unable to empty trace.log"
```

Use an operating-system or file-system deletion facility when the file itself must be removed.

31.10.14 Closing a Cached Stream with CLOSEFILE

`CLOSEFILE()` explicitly closes a cached stream opened by the Level B text I/O functions:

```
result = closefile(fileName [, suppressMessage])
```

Its signature is:

```
closefile(fileName = .string, suppressMessage = 1) = .int
```

The function returns `0`. `suppressMessage` defaults to `1`, so closing a stream that is not currently cached produces no diagnostic. Pass `0` when a diagnostic is wanted for that case. `CLOSEFILE()` is a Level B helper, not a portable classic Rexx built-in. `LINEOUT()` and `CHAROUT()` remain convenient ways to close a stream when their data argument is omitted.

31.11 Selecting an I/O Facility

The appropriate operation depends on the structure of the data and the way it will be consumed:

- use `LINEIN()` with `LINES()` for sequential text records;
- use `LINEOUT()` to write complete lines;
- use `CHARIN()` for sequential Unicode code-point input;
- use `CHAROUT()` for text that must not receive an automatic newline;
- use `READLINES()` to load a selected line range into an array;
- use `LOADTEXT()` to obtain a complete text file as one string; and
- use `READBINARY()` and `WRITEBINARY()` for exact byte-oriented file I/O;
- use `ERASEFILE()` to retain a file while reducing it to zero length; and
- use `CLOSEFILE()` when a cached stream must be closed explicitly.

The line and character functions keep memory use small and are appropriate for streaming large inputs. The bulk routines are more convenient when random access or repeated processing justifies holding the data in memory.

All the functions described here operate on text. Their treatment of line terminators, Unicode characters, and record boundaries is intentional and should not be relied upon for a byte-exact copy of binary data.

32 Signal Handling

32.1 Overview

Signals report exceptional conditions that interrupt the ordinary execution of a program. A signal may originate in a CREXX statement, in a called routine, or in the runtime system. Examples include an explicit application failure, an invalid conversion, division by zero, an out-of-range value, or interruption by the operating environment.

CREXX Level B provides two complementary forms of signal handling:

- procedure-scoped handlers installed with `SIGNAL ON`; and
- block-scoped handlers written as `ON SIGNAL` clauses in a simple `DO ... END` group.

Procedure-scoped handlers are useful when one routine is responsible for a condition throughout a substantial part of a call tree. Block-scoped handlers place the protected operations and their recovery code together, making them better suited to a particular calculation or resource operation.

Signals are represented by `.signal` objects. Such an object identifies the condition, carries an optional message or payload, and can provide source information for runtime errors. Procedure handlers receive this object as an argument. A block handler can bind it to a local variable with an `AS` clause.

Level B signal handling is structured. It does not use the classic REXX `SIGNAL` in-

struction as an unrestricted branch to a label.

32.2 Signal Names

Every signal has a name and a numeric code. Programs normally use the name, which is clearer and does not depend on the numeric representation used by the runtime.

Runtime and application conditions include names such as:

```
FAILURE
ERROR
DIVISION_BY_ZERO
CONVERSION_ERROR
INVALID_ARGUMENTS
OUT_OF_RANGE
UNICODE_ERROR
UNKNOWN_INSTRUCTION
FUNCTION_NOT_FOUND
NOT_IMPLEMENTED
INVALID_SIGNAL_CODE
OTHER
```

The runtime also recognizes operating-system-style signals including:

```
QUIT
TERM
POSIX_INT
POSIX_HUP
POSIX_USR1
POSIX_USR2
POSIX_CHLD
```

`KILL` cannot be masked and therefore cannot be consumed by an ordinary Rexx handler. `BREAKPOINT` belongs to the tracing and debugging mechanism; it is not an ordinary application error condition.

Signal names written literally in a `SIGNAL` statement are checked by the compiler. An unknown literal name is a compile-time error. A name constructed dynamically cannot be checked until execution; if it does not identify a known signal, the runtime raises `INVALID_SIGNAL_CODE`.

32.3 Raising a Signal

A program can raise a named signal directly:

```
signal failure
```

An accompanying message or payload may follow the name:

```
signal failure "The input file could not be processed"
```

The compact form treats the bare identifier as a signal name. It is conceptually equivalent to constructing a `.signal` object:

```
signal .signal("failure")
```

and:

```
signal .signal("failure",
               "The input file could not be processed")
```

The object form is also used when the signal name is held in a variable:

```
condition_name = "failure"
signal .signal(condition_name)
```

It can carry an object payload rather than only a string:

```
signal .signal("error", diagnostic_object)
```

The qualified factory form is available when the library namespace must be made explicit:

```
signal rxfnsb..signal("conversion_error", payload)
```

A `.signal` object received by a handler can be raised again. This is commonly used after local logging or cleanup:

```
signal problem
```

where `problem` is a local variable of type `.signal`.

32.4 The Signal Object

The public `.signal` interface separates the language-visible condition from the runtime's internal interrupt representation. User programs should work with `.signal`; the raw runtime signal classes are implementation details.

A signal object provides the following methods:

```
name:    method = .string
code:    method = .int
module:  method = .int
address: method = .int
message: method = .string
payload: method = .object
file:    method = .string
line:    method = .int
column:  method = .int
source:  method = .string
```

The principal descriptive properties are:

- `name()` – the symbolic signal name;
- `code()` – the corresponding runtime signal code;
- `message()` – a printable description of the condition; and
- `payload()` – the object supplied when the signal was raised.

For a string payload, `message()` returns that text. When the payload is another kind of object, the runtime derives the message from its string representation.

Runtime-originated signals may also identify where the condition occurred:

- `module()` gives the runtime module number;
- `address()` gives the instruction address in that module;
- `file()` gives the source-file name when metadata is available;
- `line()` and `column()` identify the source location; and
- `source()` gives the associated source fragment.

For a fault raised by an instruction, the address and source information refer to the faulting instruction. Source metadata is resolved from the closest preceding source location, because one authored clause may generate several runtime instructions.

Not every signal necessarily has meaningful source metadata. Native or asynchronous signals may occur between authored clauses, and a signal created explicitly by a program may contain only the information supplied by that program. Handlers should therefore treat source details as diagnostic information rather than assume that every field is populated.

For example:

```
report_signal: procedure
```

```
    arg condition = .signal

    say condition.name()":" condition.message()
    if condition.file() <> "" then
        say condition.file() condition.line() condition.source()
    return
```

32.5 Procedure-Scoped Handlers

A procedure-scoped handler is installed with:

```
signal on signal-name call handler
```

Several signals can share one handler:

```
signal on error, syntax call handle_problem
```

A typical installation is:

```
signal on conversion_error call handle_conversion

value = perform_conversion(input)
```

The named handler must be a procedure that accepts one `.signal` argument and returns a `.signalaction`:

```
handle_conversion: procedure = .signalaction
    arg condition = .signal

    say "Conversion failed:"
    say condition.message()
    say condition.source()

    return .signalaction.skip()
```

The compiler validates the handler interface. If the procedure cannot be found, it reports `SIGNAL_HANDLER_NOT_FOUND`. If its argument or return signature is unsuitable, it reports `SIGNAL_HANDLER_BAD_SIGNATURE` at the handler name in the `SIGNAL ON` statement.

32.5.1 Handler Actions

A procedure handler returns one of two explicit actions:

```
.signalaction.skip()
.signalaction.fail()
```

`skip` resumes after the signal point:

```
return .signalaction.skip()
```

The handler accepts responsibility for the condition and execution continues. This action must be used carefully for signals raised while calculating a value. Skipping a failed operation may leave its intended result unavailable.

`fail` allows the signal to continue to the default failure handling:

```
return .signalaction.fail()
```

Instruction-level `.signalaction.retry()` was retired before Release 1. It is not a standard REXX condition-trap continuation and can repeat partial writes or external side effects. Put retryable work in an explicit loop or wrapper procedure instead.

The runtime then produces its normal panic report, including source metadata when available.

A handler that falls off its end or returns `.void` is treated as `fail`. Continuation must be requested explicitly; an accidentally incomplete handler does not silently consume a runtime error.

32.5.2 Removing a Handler

A procedure-scoped handler is removed with:

```
signal off conversion_error
```

Several handlers may be removed in one statement:

```
signal off error, syntax
```

`SIGNAL OFF` restores the runtime's root behaviour for each named signal. That default is either to halt or, for signals that the runtime ignores by default, to ignore the signal.

32.5.3 Scope and Called Procedures

Handlers belong to a procedure invocation. A procedure called after a handler has been installed inherits the current handler settings:

```
main: procedure
    signal on error call handle_error
    call worker
    return
```

An `ERROR` raised in `worker` can therefore be handled by `handle_error`.

The called procedure receives its own copy of the handler settings. If it installs or removes a handler, that change is local to its invocation and does not alter the caller's settings. When the called procedure returns, execution continues with the caller's original handler configuration.

This gives procedure handlers a useful call-tree scope: they apply to later calls, but child procedures cannot permanently rewrite their caller's signal policy.

32.6 Block-Scoped Handlers

A block-scoped handler is attached to a simple `DO ... END` group:

```
do
    risky_work()
    more_risky_work()
on signal conversion_error as condition
    say condition.source()
end
```

The statements before the first `ON SIGNAL` clause form the protected body. If they complete normally, all handler clauses are skipped. If a matching signal occurs, execution transfers to the corresponding handler.

`ON SIGNAL` clauses are handlers, not labels. When a handler completes normally, execution leaves the complete `DO` block; it does not return to the statement following the fault.

32.6.1 Placement

Signal clauses may be attached only to a simple `DO` group: a group without a loop repetitor, conditional phrase, or expression-block value.

To protect an operation inside a loop, nest a simple group:

```
do i = 1 to count
    do
        risky_work(i)
    on signal error as problem
        call log_problem(problem)
    end
end
```

The inner block establishes and restores its handlers for that iteration. The outer loop retains its ordinary loop semantics.

32.6.2 Handling Named Signals

One clause can handle one or more named signals:

```
do
    result = convert(input)
on signal conversion_error, out_of_range as problem
    call report_problem(problem)
end
```

The names are matched against the signal delivered by the runtime. A different signal continues to another applicable handler or propagates out of the block.

32.6.3 Catching All Maskable Signals

An `ON SIGNAL` clause without names is a catch-all:

```
do
    perform_operation()
on signal as problem
    call log_problem(problem)
end
```

It catches any maskable signal not handled by an earlier, more specific clause. Signals such as `KILL`, which cannot be masked, are not made catchable by this form.

A catch-all handler should normally follow the named handlers:

```
do
    perform_operation()
on signal conversion_error as problem
    call report_bad_input(problem)
on signal
    call perform_general_cleanup()
end
```

The final clause needs no bound signal object because its body performs fixed cleanup and does not inspect the condition.

32.6.4 Binding the Signal Object

The optional `AS` phrase binds the current `.signal` object to a local name:

```
on signal error as problem
```

The handler can then inspect or re-raise it:

```
do
    update_resource()
on signal error as problem
    call log_problem(problem)
    signal problem
end
```

If `AS` is omitted, no implicit variable such as `condition` is created:

```
on signal error
    call rollback()
```

This form is appropriate when the handler performs a fixed action and does not need information about the signal.

32.6.5 Completion and Propagation

A block handler does not return a `.signalaction`. Completing the handler normally consumes the signal and leaves the protected block:

```
do
    result = convert(input)
on signal conversion_error
    result = default_value
end
```

To propagate the condition, bind its object and raise it again:

```
do
    result = convert(input)
on signal conversion_error as problem
    call log_problem(problem)
    signal problem
end
```

The enclosing block or procedure handler can then process it. If no applicable handler exists, normal runtime failure handling applies.

There is no `FINALLY` clause in this signal model. A bare `ON SIGNAL` is a broad error trap, not an unconditional cleanup clause: it runs only when a signal is delivered.

32.7 Nested Handlers

Block handlers temporarily replace the applicable handlers in the current procedure. On every normal block exit, the previous settings are restored. They are also restored when control leaves through a signal, an early return, or procedure termination.

This permits blocks to specialize a broader procedure policy:

```
signal on error call procedure_error

do
    special_operation()
on signal error as problem
    call local_error(problem)
end

call another_operation()
```

An `ERROR` in `special_operation()` is handled by the local block. After that block has been left, the procedure-level `procedure_error` handler is again in effect for `another_operation()`.

Nested blocks follow the same rule. The innermost applicable handler receives the signal first. Re-raising the `.signal` object allows an enclosing handler to see the same condition.

32.8 Signals Raised While Handling a Signal

Signal delivery is suspended while a Rexx signal handler is executing. If the handler raises another signal, the new condition becomes pending rather than interrupting the handler immediately.

The pending signal is delivered after control returns to a frame that is not itself handling a signal. This prevents immediate recursive interruption of the handler, but it does not discard the newly raised condition.

Handler code should nevertheless remain conservative. A handler that repeatedly raises the same uncorrected signal can produce a cycle of deferred deliveries.

If several signals are pending, the runtime handles them in signal-code priority order. Programs should not depend on the order in which unrelated asynchronous conditions happen to arrive.

32.9 Unhandled Signals

When no installed handler accepts a runtime error, the runtime applies the signal's default action. For ordinary error signals this normally halts execution and produces a panic report.

For faults raised by the current instruction, that report identifies the faulting instruction and uses the closest available source metadata. This normally provides the source file, line, column, and source fragment associated with the failed operation.

Some runtime signals are ignored by default. `SIGNAL OFF` restores that default rather than necessarily changing every signal to a halting condition.

32.10 Signals and TRACE

Signal handling and `TRACE` share parts of the runtime interrupt mechanism, but they have different language meanings.

Ordinary error signals describe a fault and report the faulting instruction. Tracing uses the special `BREAKPOINT` signal as a step-and-resume mechanism, whose

address identifies the next instruction. A user signal handler should not treat `BREAKPOINT` as an application error.

The trace runtime filters its own support code and the formal CREXX runtime by default. Installing an ordinary signal handler does not itself enable tracing, and tracing should not cause trace-support code to interrupt itself.

32.11 Choosing a Handler Form

Use a procedure-scoped handler when:

- one policy should apply throughout a procedure and the routines it calls;
- recovery requires a separate, reusable handler procedure;
- the handler must explicitly choose between retry, skip, and failure; or
- several signals share the same procedure-level policy.

Use a block-scoped handler when:

- the protected operation is small and clearly delimited;
- recovery belongs beside the operation that may fail;
- different parts of one procedure need different policies; or
- the signal should be consumed by leaving a protected region.

In both forms, catch only the signals that the code can handle meaningfully. A broad catch-all handler is useful for logging or cleanup, but it should re-raise conditions that it cannot safely resolve.

32.12 Complete Example

The following program combines a local conversion handler with a broader procedure handler:

```
options levelb
import rxfnsb

main: procedure
    signal on error call handle_error

    do
```

```
        value = convert_input()
        say "Converted value:" value
    on signal conversion_error as problem
        say "Invalid input:" problem.message()
        value = 0
    end

    call continue_processing(value)

    signal off error
    return

handle_error: procedure = .signalaction
    arg problem = .signal

    say problem.name()":" problem.message()
    if problem.file() <> "" then
        say problem.file() problem.line() problem.source()

    return .signalaction.fail()
```

The block consumes a `CONVERSION_ERROR` and supplies a local default. Other `ERROR` conditions are passed to `handle_error`, which reports the condition and deliberately selects normal failure handling.

33

Built-in functions

These built-in functions work with CREXX strings and binary values. The `.string` type lies at the heart of REXX and its related languages, and CREXX provides the built-in functions traditionally associated with it.[15] Language level B adds byte-oriented functions for values of type `.binary`.

The built-in functions are described here because they have traditionally formed an important part of the REXX language reference. This reflects the language's origins as a string-oriented programming language.

The *Library Reference* provides more detailed documentation of the standard library, including the forms available at each CREXX language level.

Use of these functions needs import of the `rxfnsb` package:

```
import rxfnsb
```

TABLE 10: SAA REXX Built-In-Functions.

BIF	Signature
ABS	ABS(number)
FORMAT	FORMAT(number [,before [,after [,expp [,expt]]]])
MAX	MAX(number, …)
MIN	MIN(number, …)
SIGN	SIGN(number)
TRUNC	TRUNC(number [,digits])
B2X	B2X(b)

[15] also colloquially referred to with the jargon-like expression BIFs.

BIF	Signature
C2D	C2D(s)
C2X	C2X(s)
D2C	D2C(number [,length])
D2X	D2X(number [,length])
X2B	X2B(x)
X2BIN	X2BIN(x)
X2C	X2C(x)
X2D	X2D(hexadecimal [,length])
XRANGE	XRANGE([start [,end]])
CENTER	CENTER(string, length [,pad])
CENTRE	CENTRE(string, length [,pad])
CHANGESTR	CHANGESTR(needle, haystack, replacement)
COMPARE	COMPARE(left, right [,pad])
CHARIN	CHARIN(name, count)
CHAROUT	CHAROUT(name, string)
COPIES	COPIES(string, count)
COUNTSTR	COUNTSTR(needle, haystack)
DELSTR	DELSTR(string, start [,length])
DELWORD	DELWORD(s, start, n)
INSERT	INSERT(new, target [,before [,length [,pad]]])
JUSTIFY	JUSTIFY(s, width, pad)
LEFT	LEFT(string, length [,pad])
LENGTH	LENGTH(string)
LINEIN	LINEIN(name)
LINEOUT	LINEOUT(name, string)
LINES	LINES(name)
LOWER	LOWER(string)
OVERLAY	OVERLAY(new, target [,start [,length [,pad]]])
POS	POS(needle, haystack [,start])
LASTPOS	LASTPOS(needle, haystack [,start])
RIGHT	RIGHT(string, length [,pad])
REVERSE	REVERSE(string)
SPACE	SPACE(string [,count [,pad]])
STRIP	STRIP(string [,option [,char]])
SUBSTR	SUBSTR(string, start [,length [,pad]])
SUBSTRO	SUBSTRO(string, start [,length [,pad]])
SUBWORD	SUBWORD(s, start, n)
TRANSLATE	TRANSLATE(s, new, old)
UPPER	UPPER(string)
VERIFY	VERIFY(string, reference [,option [,start]])
WORD	WORD(s, n)
WORDINDEX	WORDINDEX(s, n)
WORDLENGTH	WORDLENGTH(s, n)

BIF	Signature
WORDS	WORDS(s)
DATATYPE	DATATYPE(s, type)
RANDOM	RANDOM(min, max, seed)
TIME	TIME(option)
DATE	DATE(oformat, date, iformat, osep, isep)
SOURCELINE	SOURCELINE(n)
ARG	ARG(n)
STORAGE	STORAGE(address, length, newvalue)
TRACE	TRACE(option)
VALUE	VALUE(symbol, newvalue, selector)

TABLE 11: Non-SAA Functions.

Function	Signature
ABBREV	ABBREV(info, word, length)
ADDRESS	ADDRESS()
CONDITION	CONDITION([info])
DIGITS	DIGITS()
FORM	FORM()
FUZZ	FUZZ()
FNV	FNV(string)
QUEUED	QUEUED()

TABLE 12: cRexx additional functions.

Function	Signature
ARRAYAPPEND	ARRAYAPPEND(array, value [,count])
ARRAYCONTAINS	ARRAYCONTAINS(array, value [,case])
ARRAYCOPY	ARRAYCOPY(array [, from [, count]])
ARRAYDELETE	ARRAYDELETE(array, from, count)
ARRAYDROP	ARRAYDROP(array)
ARRAYFIND	ARRAYFIND(needle, array [,from [,case_sensitive]])
ARRAYGET	ARRAYGET(array, index [, default])
ARRAYHI	ARRAYHI(array[, `GET'
ARRAYINDEXOF	ARRAYINDEXOF(array, value [, from [, case]])
ARRAYINSERT	ARRAYINSERT(array, from, count[, default])
ARRAYJOIN	ARRAYJOIN(array, [, separator])
ARRAYMOVE	ARRAYMOVE(array, from, count, to)
ARRAYPOP	ARRAYPOP(array, [default])
ARRAYPREPEND	ARRAYPREPEND(array, value [,count])
ARRAYREVERSE	ARRAYREVERSE(array)

Function	Signature
ARRAYSET	ARRAYSET(array, index, value [, fill])
ARRAYSHIFT	ARRAYSHIFT(array [,default])
ARRAYSORT	ARRAYSORT(array [, offset] [, order] [,debug])
BIN2X	BIN2X(binary)
BINBYTE	BINBYTE(binary, position)
BINCOMPARE	BINCOMPARE(left, right)
BINCONCAT	BINCONCAT(left, right)
BINDELSTR	BINDELSTR(binary, start, length)
BININSERT	BININSERT(new, target, before)
BINLENGTH	BINLENGTH(binary)
BINOVERLAY	BINOVERLAY(new, target, start)
BINPOS	BINPOS(needle, haystack, start)
BINSETBYTE	BINSETBYTE(binary, position, byte)
BINSUBSTR	BINSUBSTR(binary, start, length)
BINAPPEND	BINAPPEND(dst, src)
BINCLEAR	BINCLEAR(binary)
BINCOPY	BINCOPY(dst, dst_offset, src, src_offset, length)
BINDROP	BINDROP(binary, offset, length)
BINFILL	BINFILL(binary, byte)
BINFILLAT	BINFILLAT(binary, offset, length, byte)
BINMAKEGAP	BINMAKEGAP(binary, offset, length)
BINMEMMOVE	BINMEMMOVE(binary, dst_offset, src_offset, length)
BINRESIZE	BINRESIZE(binary, length)
BINUPDATE	BINUPDATE(binary, offset, src)
QEXTRACTALL	QEXTRACTALL(open, close, text [, start [, mode]])
QEXTRACTPAIR	QEXTRACTPAIR(open, close, text [, start [, mode]])
QPOS	QPOS(needle, text [, start])
QREMOVEALL	QREMOVEALL(open, close, text [, mode])
QSPLIT	QSPLIT(text, sep)
QSPLITSAFE	QSPLITSAFE(text, sep [, start [, pairs]])
QSTRIPCOMMENT	QSTRIPCOMMENT(open [, close], text)
QSUBWORD	QSUBWORD(string, wordnum [, count])
QWORD	QWORD(line, wanted)
QWORDINDEX	QWORDINDEX(string,wordnum)
QWORDLENGTH	QWORDLENGTH(string,wordnum)
QWORDPOS	QWORDPOS(search, string [,start])
QWORDS	QWORDS(string)
RERADIX	RERADIX(subject, fromradix, toradix)
SEQUENCE	SEQUENCE(from, to)
SPLICE	SPLICE(replacement, source, at, remove_length)
VERSION	VERSION()

33.1 Level B text file I/O

The sequential file BIFs in `rxfnsb` are Level B UTF text functions. They operate on `.string` values and validate text read from files according to the normal Level B UTF-8 contract. They are not byte-oriented binary I/O BIFs.

Function	Result	Notes
`LINEIN(name)`	`.string`	Read one line from the named text stream, without the line terminator.
`LINEOUT(name [, string])`	`.int`	With `string`, write the text followed by a newline. Without `string`, close the named stream.
`CHARIN(name [, count])`	`.string`	Read up to `count` UTF codepoints from the named text stream; the default count is `1`.
`CHAROUT(name [, string])`	`.int`	With `string`, write text without appending a newline. Without `string`, close the named stream.
`LINES(name)`	`.int`	Return `1` when more text can be read from the stream, otherwise `0`.
`READBINARY(path)`	`.binary`	Read the complete file as exact bytes; I/O failure raises `NOTREADY`.
`WRITEBINARY(path, data)`	`.int`	Replace the file with `.binary` data and return its byte count; I/O failure raises `NOTREADY`.

`READBINARY` and `WRITEBINARY` use binary file modes and the VM byte I/O path. They preserve embedded NUL and invalid UTF-8 without newline translation. They are whole-file conveniences rather than incremental streams; callers own the memory cost. Do not use the text BIFs for arbitrary byte payloads.

33.2 Binary byte helpers

These Level B helpers operate on `.binary` byte buffers, not `.string` codepoints. Positions are 1-based at the Rexx surface; byte values are integers in the range `0..255`. Invalid text is never routed through these functions.

Function	Result	Notes
`BINLENGTH(data)`	`.int`	Byte length.
`BINBYTE(data, position)`	`.int`	Byte at position, or `-1` if out of range.
`BINSETBYTE(data, position, byte)`	`.binary`	Copy with one byte replaced; invalid byte/position raises `OUT_OF_RANGE`.
`BINSUBSTR(data, start, length)`	`.binary`	Byte slice; omitted/negative length means to the end.
`BINCONCAT(left, right)`	`.binary`	Byte concatenation. The `||` operator does the same when either operand is `.binary`.
`BINOVERLAY(new, target, start)`	`.binary`	Fixed-size byte overlay; writes past target raise `OUT_OF_RANGE`.
`BININSERT(new, target, before)`	`.binary`	Insert before the 1-based byte position; past the end appends.
`BINDELSTR(target, start, length)`	`.binary`	Delete a byte range; length `0` deletes to the end.
`BINPOS(needle, haystack, start)`	`.int`	1-based byte search, `0` when not found.
`BINCOMPARE(left, right)`	`.int`	`0` if equal, otherwise first differing byte position.
`BIN2X(data)`	`.string`	Uppercase hexadecimal text for the bytes.
`X2BIN(hex)`	`.binary`	Hex text to bytes; blanks are ignored and an odd nibble is left-padded with `0`.

The `||` operator also performs byte concatenation when either operand is `.binary`. In that case the result is `.binary`; a `.string` operand is copied as its exact UTF-8

bytes. Blank concatenation remains a text operation and is not for binary payload construction.

33.3 Packed binary memory helpers

The helpers in this section are the Release 1 packed-memory helper surface. They are distinct from the older `BIN*` byte helpers above:

- packed-memory helpers use zero-based byte offsets;
- mutating helpers update the first binary argument through `arg expose`;
- mutating helpers return the new logical byte length unless stated otherwise;
- the compiler or inliner may lower selected helpers directly to RXAS;
- use binary-memory intrinsics, not helpers, for direct reads from binary constants and for zero-copy compare.

Implementation note: Release 1 provides these helpers in `rxfnsb`. Some helpers currently use conservative library code and may be direct-lowered by the compiler or inliner later.

Function	Result	Notes
`BINRESIZE(data, length)`	`.int`	Resize `data` to `length` bytes. Existing bytes are preserved and growth is zero-filled.
`BINCLEAR(data)`	`.int`	Clear `data` to length `0`; returns `0`.
`BINFILL(data, byte)`	`.int`	Fill the whole current logical byte range with `byte`; returns the byte length.
`BINFILLAT(data, offset, length, byte)`	`.int`	Fill a zero-based byte span with `byte`; returns the byte length.
`BINCOPY(dst, dst_offset, src, src_offset, length)`	`.int`	Copy `length` bytes from `src` to `dst`. Use `BINMEMMOVE` for overlapping ranges in the same binary value.

Function	Result	Notes
`BINMEMMOVE(data, dst_offset, src_offset, length)`	`.int`	Move `length` bytes within one binary value. Overlapping ranges are safe.
`BINAPPEND(dst, src)`	`.int`	Append all bytes from `src` to `dst`; returns the new byte length.
`BINUPDATE(dst, offset, src)`	`.int`	Overlay all bytes from `src` into `dst` at zero-based `offset`; the write must fit.
`BINMAKEGAP(data, offset, length)`	`.int`	Open a zero-filled gap of `length` bytes at `offset`; returns the new byte length.
`BINDROP(data, offset, length)`	`.int`	Delete `length` bytes at `offset`; returns the new byte length.

Examples:

```
call binresize page, 4096
call binfill page, 0
call binmemmove page, dst_offset, src_offset, span_len
call bincopy target, 0, source, source_offset, span_len
```

Packed-memory helper offsets are intentionally zero-based so they match `<at..type>` and the RXAS binary-memory instructions. The older `BINBYTE`/`BINSUBSTR`/`BINOVERLAY` compatibility helpers remain 1-based and copy-returning.

Invalid packed-memory spans, negative lengths, and byte values outside `0..255` raise `OUT_OF_RANGE`. Zero-length spans accept offsets from zero through the current logical byte length, inclusive. The complete Level B contract and signal examples are in `binary.md`.

33.4 Array helpers

Level B arrays use `array[0]` as the high-water mark. User elements are stored in `array[1]` through `array[array[0]]`. The `array*` helpers below live in `rxfnsb`; mutating helpers take the array by `expose` and update it in place.

These helpers operate on `.string[]` arrays. They are not generic array helpers and are not the supported surface for typed numeric arrays such as `.int[]`; use direct indexing for those arrays until typed helpers are added.

Function	Result	Notes
`ARRAYHI(array, mode, newhi)`	`.int`	Get the high-water mark, or shrink it with mode `SET`.
`ARRAYDROP(array)`	`.int`	Clear the array in place and return `0`.
`ARRAYINSERT(array, from, count [,default])`	`.int`	Open a gap at `from`, fill it, and return the new high-water mark.
`OBJECTARRAYINSERT(array, from, count, value)`	`.int`	Open an object-array gap and fill it with object-value copies.
`OBJECTARRAYDELETE(array, from, count)`	`.int`	Delete an object-array range and return the new high-water mark.
`OBJECTARRAYAPPEND(array, value [,count])`	`.int`	Append object-value copies and return the new high-water mark.
`OBJECTARRAYPREPEND(array, value [,count])`	`.int`	Prepend object-value copies and return the new high-water mark.
`OBJECTARRAYDROP(array)`	`.int`	Clear an object array in place and return zero.
`OBJECTARRAYMOVE(array, from, count, to)`	`.int`	Move an object-array block and preserve its high-water mark.
`ARRAYDELETE(array, from, count)`	`.int`	Delete a range and return the new high-water mark.
`ARRAYAPPEND(array, value [,count])`	`.int`	Append `value` `count` times.
`ARRAYPREPEND(array, value [,count])`	`.int`	Prepend `value` `count` times.
`ARRAYPOP(array, default)`	`.string`	Remove and return the last element, or `default` when empty.
`ARRAYSHIFT(array, default)`	`.string`	Remove and return the first element, or `default` when empty.

Function	Result	Notes
`ARRAYSET(array, index, value, fill)`	`.int`	Set an element; growing gaps are initialised with `fill`.
`ARRAYGET(array, index, default)`	`.string`	Return an element, or `default` for an out-of-range index.
`ARRAYCOPY(array, from, count)`	`.string[]`	Return a copied slice; negative `from` counts from the end.
`ARRAYMOVE(array, from, count, to)`	`.int`	Move a range within the same array.
`ARRAYREVERSE(array)`	`.int`	Reverse the array in place.
`ARRAYSORT(array, offset, order, debug)`	`.int`	Sort strings by a substring key.
`ARRAYFIND(needle, array [,from [,case_sensitive]])`	`.int`	Find the first element containing a substring.
`ARRAYINDEXOF(array, value, from, case)`	`.int`	Find the first element equal to `value`.
`ARRAYCONTAINS(array, value, case)`	`.int`	Return `1` when an element equals `value`, else `0`.
`ARRAYJOIN(array, separator)`	`.string`	Join all elements with `separator`.

Insert, delete, append, prepend, pop, shift, shrink, and clear operations use the VM array attribute instructions, so the pointer array can be adjusted without a Rexx-level per-element copy loop. Element values are still ordinary Rexx strings and keep the usual copy and lifetime rules.

33.5 Quote-aware helpers

The `q*` helpers share one positional Unicode scanner. They are ordinary Level B `rxfnsb` functions, not Classic Level C BIFs. Single and double ASCII quotes can occur anywhere in a word, matching doubled quotes are escapes, quote delimiters remain in returned values, and malformed quote or pair grammar signals `INVALID_ARGUMENTS`. Positions and lengths are Unicode codepoints; word separation uses Unicode 17.0 `White_Space`.

`QPOS(needle, text [, start])` returns the 1-based position of `needle` outside single- or double-quoted regions, or `0` when it is not found. `start` is a positive codepoint position.

`QSPLIT(text, sep)` splits `text` on `sep` only when the separator is outside quoted regions. It returns exact `.string[]` fields, including empty and trailing fields, without stripping source whitespace.

`QSPLITSAFE(text, sep, start, pairs)` is the nested-safe splitter. In addition to quote tracking, it tracks nested one-codepoint delimiter pairs from `pairs`, such as the default `()`, and only splits at depth zero.

`QEXTRACTPAIR` and `QEXTRACTALL` return balanced top-level source spans. Modes `X/E` return the contents and `I/C` include delimiters. `QREMOVEALL` uses the same spans; its default inclusive mode removes the complete regions, while exclusive mode retains the delimiters. Equal text outside a selected region is never removed.

`QSTRIPCOMMENT` removes line comments when `close` is omitted or empty and preserves CRLF, LF, and CR endings exactly. With a closer it removes nested balanced block comments.

`QWORD`, `QWORDINDEX`, `QWORDLENGTH`, and `QWORDS` use the same word spans. `QWORDPOS` matches an exact word sequence. `QSUBWORD` preserves separators inside the selected source span; omitted `count` selects through the last word and explicit zero returns an empty string. The selector-local pages under `lib/rxfnsb/rexx` contain the complete signatures and examples.

33.6 ABS(number)

returns the absolute value of *string*, which must be a number. Any sign is removed from the number, and it is then formatted by adding zero with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). Scientific notation is used, if necessary.

The native Level B function accepts and returns `.decimal`. Its invalid dynamic conversion signal is currently tracked as a VM dependency in the programme worklist. The standalone Level C BIF accepts Classic numeric text through `rNUM`, including the blank-separated leading-sign forms below, and reports standard `RXC-LC-40.*` context errors. See the separate Level B ABS and Level C ABS pages

for their distinct contracts.

```
ABS('12.3')              == 12.3
ABS(' -0.307')           == 0.307
ABS('123.45E+16')        == 1.2345E+18
ABS('- 1234567.7654321') == 1234567.7654321
```

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

`FORMAT` lays out a numeric value under the numeric settings current at the invocation. With only `number`, it returns that normalized number.

`before` and `after` are optional non-negative whole numbers. `before` is the width of the integer part, including a minus sign, and left-pads with blanks. If the integer part cannot fit, error 40.38 results. `after` is the exact number of digits after the decimal point: missing digits are zero-filled and excess digits are rounded half up. A supplied zero removes the decimal point.

`expp` is the exponent digit width. If it is too small, error 40.38 results; if the exponent is zero, a supplied width produces `expp + 2` blanks. A supplied zero suppresses exponential notation. `expt` controls when exponent form is used; when `expp` is present and `expt` is omitted, the current `NUMERIC DIGITS` setting is the trigger. The current `NUMERIC FORM` setting selects scientific or engineering layout. `FORMAT` has five arguments; form is not a sixth argument.

```
FORMAT(' - 12.73')          == '-12.73'
FORMAT('1.73', 4, 0)        == '   2'
FORMAT('-.76', 4, 1)        == '  -0.8'
FORMAT('12345.73',,,2,2)    == '1.234573E+04'
FORMAT('1.2345',,3,2,0)     == '1.235    '
```

See the separate Level C BIF contract and native Level B typed API.

33.8 MAX(number, ...)

returns the larger of *string* and *number*, which must both be numbers. If they compare equal (that is, when subtracted, the result is 0), then *string* is selected for the result.

The comparison is effected using a numerical comparison with a digits setting

that is either nine or, if greater, the larger of the number of digits in the mantissas of the two numbers (excluding leading insignificant zeros).

The selected result is formatted by adding zero to the selected number with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). Scientific notation is used, if necessary.

The native Level B function accepts and returns `.decimal` and performs one linear scan. The standalone Level C BIF accepts Classic variadic `rNUM...` text, preserving the first selected normalized argument representation. See the separate Level B MAX and Level C MAX pages for their distinct contracts.

```
MAX(0, 1)          ==1
MAX('-1', 1)       ==1
MAX('+1', -1)      ==1
MAX('1.0', 1.00)   =='1.0'
MAX('1.00', 1.0)   =='1.00'
MAX('123456700000', 1234567E+5)   == '123456700000'
MAX('1234567E+5', '123456700000') == '1.234567E+11'
```

33.9 MIN(number, ...)

returns the smaller of *string* and *number*, which must both be numbers. If they compare equal (that is, when subtracted, the result is 0), then *string* is selected for the result.

The comparison is effected using a numerical comparison with a digits setting that is either nine or, if greater, the larger of the number of digits in the mantissas of the two numbers (excluding leading insignificant zeros).

The selected result is formatted by adding zero to the selected number with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). Scientific notation is used, if necessary.

The native Level B function accepts and returns `.decimal` and performs one linear scan. The standalone Level C BIF accepts Classic variadic `rNUM...` text, preserving the first selected normalized argument representation. See the separate Level B MIN and Level C MIN pages for their distinct contracts.

```
MIN(0, 1)          ==0
```

```
MIN('-1', 1)       =='-1'
MIN('+1', -1)      =='-1'
MIN('1.0', 1.00)   =='1.0'
MIN('1.00', 1.0)   =='1.00'
MIN('123456700000', 1234567E+5)   == '123456700000'
MIN('1234567E+5', '123456700000') == '1.234567E+11'
```

33.10 SIGN(number)

returns a number that indicates the sign of *string*, which must be a number. *string* is first formatted, just as though the operation "**string+0**" had been carried out with sufficient digits to avoid rounding. If the number then starts with **'-'** then **'-1'** is returned; if it is **'0'** then **'0'** is returned; and otherwise **'1'** is returned.

```
SIGN('12.3')    ==  1
SIGN('0.0')     ==  0
SIGN(' -0.307') == -1
```

The native Level B helper accepts a `.decimal` and returns `.int`, preserving sign outside binary floating-point range. The standalone Level C BIF accepts Classic `rNUM` text and returns a RexxValue integer with standard `RXC-LC-40.*` errors. See the separate Level B SIGN and Level C SIGN pages for their distinct contracts.

33.11 TRUNC(number [,digits])

returns the integer part of *string*, which must be a number, with *n* decimal places (digits after the decimal point). *n* must be a non-negative whole number, and defaults to zero.

The number *string* is formatted by adding zero with a digits setting that is either nine or, if greater, the number of digits in the mantissa of the number (excluding leading insignificant zeros). It is then truncated to *n* decimal places (or trailing zeros are added if needed to make up the specified length). If *n* is 0 (the default) then an integer with no decimal point is returned. The result will never be in exponential form.

```
TRUNC('12.3')         == 12
TRUNC('127.09782', 3) == 127.097
TRUNC('127.1', 3)     == 127.100
TRUNC('127', 2)       == 127.00
```

```
TRUNC('0', 2)          == 0.00
```

The native Level B helper accepts `.decimal` plus a non-negative `.int` and signals `INVALID_ARGUMENTS` for a negative digit count. The standalone Level C BIF accepts Classic `rNUM` and optional `oWHOLE>=0` RexxValue text with standard `RXC-LC-40.*` errors. Both avoid binary floating point. See the separate Level B TRUNC and Level C TRUNC pages for their distinct contracts and test scope.

33.12 B2X(b)

Binary to hexadecimal. Converts *string*, a string of zero or more binary (**0** and/or **1**) digits, to an equivalent string of hexadecimal characters. The returned string will use uppercase Roman letters for the values A-F, and will not include any blanks. If the number of binary digits in the string is not a multiple of four, then up to three **'0'** digits will be added on the left before conversion to make a total that is a multiple of four.

The empty string returns the empty string. Interior blanks may separate groups only when a multiple of four binary digits is to their right. Leading/trailing blanks, misplaced blanks, and other characters are invalid binary strings.

```
B2X('11000011')  == 'C3'
B2X('10111')     == '17'
B2X('0101')      == '5'
B2X('101')       == '5'
B2X('111110000') == '1F0'
```

See the separate Level C BIF contract and native Level B API.

33.13 BINLENGTH(data)

Binary byte length. Returns the number of bytes stored in `.binary` value *data*. This is a byte count, not a UTF-8 codepoint count.

```
BINLENGTH("ff0041"x as .binary) == 3
empty = .binary
BINLENGTH(empty)                == 0
BINLENGTH("□" as .binary)       == 2
```

33.14 BINBYTE(data, position)

Binary byte lookup. Returns the byte at 1-based byte *position* in `.binary` value *data* as an integer in the range `0..255`. If *position* is outside the byte buffer, `-1` is returned.

```
BINBYTE("ff0041"x as .binary, 1) == 255
BINBYTE("ff0041"x as .binary, 3) == 65
BINBYTE("ff0041"x as .binary, 4) == -1
```

33.15 BINSETBYTE(data, position, byte)

Binary byte replacement. Returns a copy of `.binary` value *data* with the byte at 1-based byte *position* replaced by *byte*. *byte* must be in the range `0..255`; invalid positions or byte values raise `OUT_OF_RANGE`.

```
BIN2X(BINSETBYTE("001122"x as .binary, 2, 255)) == "00FF22"
```

33.16 BINSUBSTR(data, start, length)

Binary substring. Returns a `.binary` byte slice from `.binary` value *data*, starting at 1-based byte position *start*. If *length* is omitted or negative, the slice continues to the end of the buffer. If the requested range extends past the end, the result is truncated at the end of *data*. A nonpositive *start* raises `INVALID_ARGUMENTS`.

```
BIN2X(BINSUBSTR("001122ff"x as .binary, 2, 2)) == "1122"
BIN2X(BINSUBSTR("001122ff"x as .binary, 3))    == "22FF"
BIN2X(BINSUBSTR("001122ff"x as .binary, 9))    == ""
```

33.17 BINCONCAT(left, right)

Binary concatenation. Returns the byte concatenation of `.binary` values *left* and *right*. The source-level `||` operator performs the same byte concatenation when either operand is `.binary`.

```
BIN2X(BINCONCAT("0011"x as .binary, "22ff"x as .binary)) == "001122FF"
BIN2X(("ff"x as .binary) || "A")                         == "FF41"
```

33.18 BINOVERLAY(new, target, start)

Binary overlay. Returns a copy of `.binary` value *target* with the bytes from `.binary` value *new* overlaid starting at 1-based byte position *start*. The overlay is fixed-size: it must fit inside *target*, or `OUT_OF_RANGE` is raised.

```
BIN2X(BINOVERLAY("abcd"x as .binary, "001122ff"x as .binary, 2)) == "00
    ABCDFF"
```

33.19 BININSERT(new, target, before)

Binary insert. Returns a copy of `.binary` value *target* with `.binary` value *new* inserted before 1-based byte position *before*. If *before* is less than or equal to 1, *new* is prepended. If *before* is beyond the end of *target*, *new* is appended.

```
BIN2X(BININSERT("abcd"x as .binary, "001122ff"x as .binary, 3))  == "
    0011ABCD22FF"
BIN2X(BININSERT("abcd"x as .binary, "001122ff"x as .binary, 99)) == "
    001122FFABCD"
```

33.20 BINDELSTR(target, start, length)

Binary delete substring. Returns a copy of `.binary` value *target* with a byte range removed. Deletion starts at 1-based byte position *start*. If *length* is omitted or `0`, bytes from *start* to the end are removed. A nonpositive *start* or negative *length* raises `INVALID_ARGUMENTS`.

```
BIN2X(BINDELSTR("001122ff"x as .binary, 2, 2)) == "00FF"
BIN2X(BINDELSTR("001122ff"x as .binary, 3))    == "0011"
```

33.21 BINPOS(needle, haystack, start)

Binary position. Searches `.binary` value *haystack* for `.binary` value *needle*, starting at 1-based byte position *start* (default `1`). Returns the 1-based byte position of the first match, or `0` if no match is found. A zero-length *needle* returns `0`. A nonpositive *start* raises `INVALID_ARGUMENTS`.

```
BINPOS("1122"x as .binary, "001122ff"x as .binary)    == 2
BINPOS("22"x as .binary, "001122ff"x as .binary, 3)   == 3
BINPOS("33"x as .binary, "001122ff"x as .binary)      == 0
```

33.22 BINCOMPARE(left, right)

Binary compare. Compares two `.binary` values byte by byte. Returns `0` when they are equal. Otherwise, returns the 1-based position of the first differing byte. If one value is a prefix of the other, the first differing position is one past the shorter value.

```
BINCOMPARE("001122ff"x as .binary, "001122ff"x as .binary) == 0
BINCOMPARE("001122ff"x as .binary, "001123ff"x as .binary) == 3
BINCOMPARE("001122ff"x as .binary, "001122"x as .binary)   == 4
```

33.23 BIN2X(data)

Binary bytes to hexadecimal. Converts `.binary` value *data* to uppercase hexadecimal text. Each byte becomes two hexadecimal characters, so the output length is always twice the input byte length.

```
BIN2X("ff0041"x as .binary) == "FF0041"
empty = .binary
BIN2X(empty)                == ""
BIN2X("□" as .binary)       == "CEB1"
```

33.24 X2BIN(hex)

Hexadecimal to binary bytes. Converts hexadecimal text *hex* to a `.binary` byte buffer. Blanks are ignored. If there is an odd number of hexadecimal digits, a leading `0` nibble is assumed. A non-blank character that is not hexadecimal raises `INVALID_ARGUMENTS`.

```
BIN2X(X2BIN("ff 00 aa")) == "FF00AA"
BIN2X(X2BIN("f"))        == "0F"
```

See the stable Level B binary module contract.

33.25 C2D(s)

Coded character to decimal. The native Level B helper converts the Unicode code point of *string*, which must contain exactly one character, to a non-negative `.int`. Empty or multi-character input raises `CONVERSION_ERROR`.

```
C2D('M')  == 77
C2D('□')  == 945
C2D('□') == 128293
C2D('00'x) == 0
```

c2x function %% (see page refc2x) can be used to convert the encoding of a character to a hexadecimal representation.

Classic Level C C2D is a different API with an optional signed-width argument. See the separate Level C BIF contract and native Level B API.

33.26 C2X(s)

Coded characters to hexadecimal. Converts every character in *string* to its established two-digit hexadecimal representation. The returned string uses uppercase Roman letters for A-F and contains no inserted blanks. The empty string returns the empty string, multi-character input is valid, and leading zero digits are retained.

The current Level B implementation preserves RXAS `hexchar` behavior for Unicode: a code point beyond the single-byte range contributes its low eight bits. Classic Level C C2X instead converts the exact coded bytes selected by the call context's BYTE or UTF8 profile.

```
C2X('M')     == '4D'
C2X('72s')   == '373273' -- ASCII/Unicode build
C2X('0123'x) == '0123'
C2X('')      == ''
```

c2d function %% (see page refc2d) can be used to convert the encoding of a character to a decimal number.

See the separate Level C BIF contract and native Level B API.

33.27 D2C(number [,length])

Classic Level C D2C converts a decimal whole number to configuration-coded characters. Without *length*, the number must be non-negative and the result uses the minimum encoded width. With a non-negative *length*, negative numbers use twos-complement and the result is padded or truncated to exactly that many coded characters. The configuration, not Unicode code-point numbering, defines those characters.

Level B deliberately provides a different typed helper: `d2c(codepoint=.int [,output_length=.int])`. It emits one Unicode scalar value; its explicit output length may only be zero or one.

See the separate Level C BIF contract and native Level B API. The direct Level C implementation returns exact bytes and records valid UTF-8 text when applicable.

33.28 D2X(number [,length])

Classic Level C decimal to hexadecimal. Returns a string of hexadecimal characters of length as needed or of length *n*, which is the hexadecimal (unpacked) representation of the decimal number. The returned string will use uppercase Roman letters for the values A-F, and will not include any blanks. *string* must be a whole number, and must be non-negative unless *n* is specified, or an error will result. If *n* is not specified, the length of the result returned is such that there are no leading 0 characters, unless *string* was equal to 0 (in which case **'0'** is returned).

If *n* is specified it is the length of the final result in characters; that is, after conversion the input string will be sign-extended to the required length (negative numbers are converted assuming twos-complement form). If the number is too big to fit into *n* characters, it will be truncated on the left. *n* must be a non-negative whole number.

```
D2X('9')       == '9'
D2X('129')     == '81'
D2X('129', 1)  == '1'
D2X('129', 2)  == '81'
D2X('127', 3)  == '07F'
D2X('129', 4)  == '0081'
D2X('257', 2)  == '01'
D2X('-127', 2) == '81'
```

```
D2X('-127', 4) == 'FF81'
D2X('12', 0)   == ''
```

The standalone Level C implementation accepts caller-context Rexx whole numbers without narrowing them to a native integer. Level B provides a separate signed-64-bit typed helper with the same result rules over its smaller numeric domain.

See the separate Level C BIF contract and native Level B API.

33.29 X2B(hexadecimal)

Hexadecimal to binary. Converts every hexadecimal digit to four binary digits; letters are case-insensitive and leading zero nibbles are retained. The empty string returns empty.

Interior blanks are ignored only when an even number of hexadecimal digits lies to their right. Leading/trailing blanks, mis-grouped blanks, or invalid characters are errors. The result contains no blanks and its length is four times the number of input digits.

```
X2B('C3')  == '11000011'
X2B('7')   == '0111'
X2B('1 C1') == '000111000001'
X2B('0001') == '0000000000000001'
X2B('') == ''
```

See the separate Level C BIF contract and native Level B API.

33.30 X2C(hexadecimal)

Classic Level C X2C validates hexadecimal text, removes valid interior grouping blanks, left-pads an odd leading nibble, and converts each full byte through the implementation's configured coded-character encoding. Empty input returns empty and encoded leading zero bytes are retained.

Hexadecimal letters are case-insensitive. Leading/trailing blanks, mis-grouped blanks, and non-hexadecimal characters are errors. Because the character encoding is configured, the character produced by a byte such as `4D` is not portable across ASCII/Unicode and EBCDIC configurations.

Level B deliberately provides a different typed helper that maps every parsed byte to Unicode U+0000 through U+00FF.

See the separate Level C BIF contract and native Level B API. Direct Level C X2C preserves exact configured bytes and records a text view only for valid UTF-8.

33.31 X2D(hexadecimal [,length])

Hexadecimal to decimal. Converts the *string* (a string of hexadecimal characters) to a decimal number, without rounding. If *string* is the null string, 0 is returned.

If *n* is not specified, *string* is taken to be an unsigned number.

```
X2D('0E')    == 14
X2D('81')    == 129
X2D('F81')   == 3969
X2D('FF81')  == 65409
X2D('c6f0')  == 50928
```

If *n* is specified, *string* is taken as a signed number expressed in *n* hexadecimal characters. If the most significant (left-most) bit is zero then the number is positive; otherwise it is a negative number in twos-complement form. In both cases it is converted to a CREXX number which may, therefore, be negative. If *n* is 0, 0 is always returned.

If necessary, *string* is padded on the left with **'0'** characters (note, not "sign-extended"), or truncated on the left, to length *n* characters; (that is, as though *string*.**right**(*n*, **'0'**) had been executed.)

```
X2D('81', 2)   == -127
X2D('81', 4)   == 129
X2D('F081', 4) == -3967
X2D('F081', 3) == 129
X2D('F081', 2) == -127
X2D('F081', 1) == 1
X2D('0031', 0) == 0
```

c2d function %% (see page refc2d) can be used to convert a character to a decimal representation of its encoding.

Classic Level C accepts standard grouped hexadecimal text and returns an exact Rexx whole number subject to the caller's `NUMERIC DIGITS`. Invalid grouping or digits report `40.25`; invalid lengths use the standard `40.12`/`40.13` errors, and a result too large for the current digits reports `40.35`.

The typed Level B helper instead returns a native signed `.int`. It signals `INVALID_ARGUMENTS` for invalid text or length and `OVERFLOW_UNDERFLOW` when the selected result is outside the signed-64-bit range. See the separate Level C contract and Level B API.

33.32 CENTER(string, length [,pad])

Returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up the required length. *length* must be a non-negative whole number. The default *pad* character is blank, and an explicit *pad* must contain exactly one character. If the string is longer than *length*, it will be truncated at both ends to fit. If an odd number of characters are truncated or added, the right hand end loses or gains one more character than the left hand end.

The Level B helper types *length* as `.int`; an invalid length or pad signals `INVALID_ARGUMENTS`. The standalone Level C BIF accepts Classic whole-number text and reports the standard `RXC-LC-40.*` context errors. Level B measures Unicode codepoints. Level C measures configured units: octets in the default BYTE profile and Unicode codepoints in the opt-in UTF8 profile. See the separate Level B CENTER and Level C CENTER pages for their distinct contracts.

```
CENTER('ABC', 7)          == '  ABC  '
CENTER('ABC', 8, '-')     == '--ABC---'
CENTER('The blue sky', 8) == 'e blue s'
CENTER('The blue sky', 7) == 'e blue '
```

Note: This function may be called either **centre** or **center**, which avoids difficulties due to the difference between the British and American spellings.

33.33 CENTRE(string, length [,pad])

Returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up the required length. *length* must be a non-negative whole number. The default *pad* character is blank, and an explicit *pad* must contain exactly one character. If the string is longer than *length*, it will be truncated at both ends to fit. If an odd number of characters are truncated or added, the right hand end loses or gains one more character than the left hand end.

The Level B helper types *length* as `.int`; an invalid length or pad signals `INVALID_ARGUMENTS`. The standalone Level C BIF accepts Classic whole-number text and reports the standard `RXC-LC-40.*` context errors. Level B measures Unicode codepoints. Level C measures configured units: octets in the default BYTE profile and Unicode codepoints in the opt-in UTF8 profile. See the separate Level B CENTRE and Level C CENTRE pages for their distinct contracts.

```
CENTRE('ABC', 7)          == '  ABC  '
CENTRE('ABC', 8, '-')     == '--ABC---'
CENTRE('The blue sky', 8) == 'e blue s'
CENTRE('The blue sky', 7) == 'e blue '
```

Note: This function may be called either **centre** or **center**, which avoids difficulties due to the difference between the British and American spellings.

33.34 CHANGESTR(needle, haystack, replacement)

Returns a copy of *haystack* in which every case-sensitive, non-overlapping occurrence of *needle* is replaced by *replacement*. Searching proceeds from left to right through the original haystack; inserted replacement text is not searched again. If *needle* is null or is not found, the haystack is returned unchanged. A null replacement deletes each match.

The Level B helper requires three `.string` arguments. The standalone Level C BIF accepts any three RexxValue texts and reports standard `RXC-LC-40.*` argument-presence errors. Level B finds codepoint-aligned text matches. Level C uses exact octet matches in BYTE and codepoint-aligned matches in UTF8. See the separate Level B CHANGESTR and Level C CHANGESTR pages for their distinct contracts and implementation notes.

```
CHANGESTR('the', 'the cat and the dog', 'a') == 'a cat and a dog'
CHANGESTR('aa', 'aaaaa', 'X')               == 'XXa'
CHANGESTR('a', 'banana', '')                == 'bnn'
CHANGESTR('', 'unchanged', '!')             == 'unchanged'
```

33.35 COMPARE(left, right [,pad])

Returns `0` when *left* and *right* compare equal. Otherwise it returns the first 1-based character position at which they differ. The shorter string is conceptually padded

on the right before comparison. *pad* defaults to blank and must contain exactly one character.

The Level B helper takes two `.string` arguments, returns `.int`, and signals `INVALID_ARGUMENTS` for an invalid pad. The standalone Level C BIF accepts RexxValue text and reports standard `RXC-LC-40.*` context errors. Its result position and one-unit pad use octets in BYTE and Unicode codepoints in UTF8; Level B always uses codepoints. See the separate Level B COMPARE and Level C COMPARE pages.

```
COMPARE('abc', 'abc')        == 0
COMPARE('abc', 'ak')         == 2
COMPARE('ab ', 'ab')         == 0
COMPARE('ab-- ', 'ab', '-')  == 5
```

33.36 COPIES(string, count)

Returns *count* directly concatenated copies of *string*. *count* must be a non-negative whole number; zero returns the null string.

The Level B helper types *count* as `.int` and signals `INVALID_ARGUMENTS` for a negative value. The standalone Level C BIF accepts Classic whole-number text and reports standard `RXC-LC-40.*` errors. Level C repeats the exact RexxValue bytes and preserves a BYTE result as binary-authoritative; Level B repeats valid UTF-8 `.string` text. See the separate Level B COPIES and Level C COPIES pages for their contracts and performance notes.

```
COPIES('abc', 3) == 'abcabcabc'
COPIES('abc', 0) == ''
COPIES('', 2)    == ''
```

33.37 COUNTSTR(needle, haystack)

Returns the number of case-sensitive, non-overlapping occurrences of *needle* in *haystack*, searching from left to right. A null, absent, or oversized needle returns `0`.

The Level B helper takes two `.string` arguments and returns `.int`. The standalone Level C BIF accepts any two RexxValue texts and reports standard `RXC-LC-40.*` argument-presence errors. Level B matches at codepoint boundaries; Level C

matches exact octets in BYTE and codepoint-aligned text in UTF8. See the separate Level B COUNTSTR and Level C COUNTSTR pages.

```
COUNTSTR('bc', 'abcabcabc') == 3
COUNTSTR('aa', 'aaaaa')     == 2
COUNTSTR('', 'anything')    == 0
```

33.38 DELSTR(string, start [,length])

Returns a copy of *string* with the substring that begins at the *start* character and is of length *length* characters, deleted. If *length* is omitted, or is greater than the number of characters from *start* to the end, the rest of the string is deleted. An explicitly supplied zero deletes nothing. *length* must be non-negative, and *start* must be a positive whole number. A start beyond the string returns it unchanged.

The Level B helper types *start* and *length* as `.int` and signals `INVALID_ARGUMENTS` for invalid values. The standalone Level C BIF accepts Classic whole-number text and reports standard `RXC-LC-40.*` errors. Level B positions and lengths are codepoints. Level C uses octets in BYTE and codepoints in UTF8. See the separate Level B DELSTR and Level C DELSTR pages.

```
DELSTR('abcd', 3)    == 'ab'
DELSTR('abcde', 3, 2) == 'abe'
DELSTR('abcde', 6)   == 'abcde'
DELSTR('abcde', 3, 0) == 'abcde'
```

33.39 DELWORD(s, start, n)

returns a copy of *string* with the sub-string of *string* that starts at the `n`th word, and is of length *length* blank-delimited words, deleted. If *length* is not specified, or is greater than number of remaining words in the string, it defaults to be the remaining words in the string (including the **n*th word)*. length* must be a non-negative whole number, and *n* must be a positive whole number. If *n* is greater than the number of words in *string*, the string is returned unchanged. The string deleted includes any blanks following the final word involved, but none of the blanks preceding the first word involved.

```
DELWORD('Now is the  time', 2, 2) == 'Now time'
DELWORD('Now is the time ', 3)   == 'Now is '
```

```
DELWORD('Now  time', 5)          == 'Now  time'
```

33.40 INSERT(new, target [,before [,length [,pad]]])

Returns a copy of *target* with *new* inserted after *before* characters. *before* defaults to zero, which inserts before the first target character. When *before* is beyond the target, padding extends the target to that position.

length controls the width of the inserted text. It defaults to the character length of *new*; a supplied zero inserts no new text. The insertion is truncated or padded to that width. *before* and *length* must be non-negative whole numbers. *pad* defaults to blank and a supplied pad must contain exactly one character.

The Level B helper uses `.string` text and `.int` position/length values and signals `INVALID_ARGUMENTS` for invalid values. The standalone Level C BIF accepts RexxValue text and reports standard `RXC-LC-40.*` context errors. Level B measures codepoints. Level C measures octets in BYTE and codepoints in UTF8, including the one-unit pad rule. See the separate Level B INSERT and Level C INSERT pages for their distinct contracts and implementation notes.

```
INSERT('123', 'abc')         == '123abc'
INSERT(' ', 'abcdef', 3)      == 'abc def'
INSERT('123', 'abc', 5, 6)     == 'abc  123   '
INSERT('123', 'abc', 5, 6, '+') == 'abc++123+++'
INSERT('123', 'abc', 0, 5, '-') == '123--abc'
INSERT('abc', 'def', 2, 1)      == 'deaf'
```

33.41 JUSTIFY(s, width, pad)

33.42 LEFT(string, length [,pad])

returns a string of length *length* containing the left-most *length* characters of *string*. The string is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank. *length* must be a non-negative whole number. A supplied pad must contain exactly one character.

The Level B helper requires `.int` *length* and signals `INVALID_ARGUMENTS` for an invalid length or pad. The standalone Level C BIF accepts Classic whole-number

text and reports standard `RXC-LC-40.*` context errors. Level B counts Unicode codepoints. Level C counts exact octets in BYTE and codepoints in UTF8. See the separate Level B LEFT and Level C LEFT pages.

```
LEFT('abc d', 8)     == 'abc d   '
LEFT('abc d', 8, '.') == 'abc d...'
LEFT('abc defg', 6)  == 'abc de'
```

33.43 LENGTH(string)

Returns the character-unit length of *string*. Level B returns its Unicode codepoint count, not the number of bytes in its UTF-8 representation; a combining codepoint is counted separately from the base character it follows. Level C returns the exact octet count in BYTE and the codepoint count in UTF8.

The Level B helper accepts `.string` and returns `.int`. The standalone Level C BIF accepts RexxValue text and returns the decimal count in a RexxValue, with standard `RXC-LC-40.*` argument errors. See the separate Level B LENGTH and Level C LENGTH pages. The specification's `23.1` invalid-character-data case is unreachable after normal CREXX text has entered the valid configured `.string` model.

```
LENGTH('abcdefgh') == 8
LENGTH('')         == 0
LENGTH('□□é')    == 3
```

33.44 LOWER(string)

Returns a copy of *string* after applying Level B's locale-independent, limited simple lowercase table. Covered letters are replaced by their lowercase equivalents; other codepoints are unchanged. This is deliberately not full Unicode case folding. The surface accepts only the string argument; it has no substring position or length options.

The helper accepts and returns `.string`, does not modify its argument, performs the runtime's limited simple mapping, and has no error branch for valid text. See Level B LOWER for its exact contract and implementation notes. LOWER is not a required Level C BIF in the repository catalog; the existing common-runtime helper is compatibility surface only.

```
LOWER('SumA') == 'suma'
LOWER('ÄÖÜÉ') == 'äöüé'
LOWER('')     == ''
```

33.45 OVERLAY(new, target [,start [,length [,pad]]])

Returns a copy of *target* with formatted *new* text written from the 1-based character position *start*. The default start is one. If the characters before *start* extend beyond the target, padding fills that gap.

When *length* is omitted it is the character length of *new*. A supplied zero writes no new text; otherwise *new* is truncated or padded to that width before replacing the corresponding target characters. *start* must be positive, *length* must be non-negative, and a supplied *pad* must contain exactly one character. The default pad is blank.

The Level B helper uses `.string` text and `.int` start/length values and signals `INVALID_ARGUMENTS` for invalid values. The standalone Level C BIF accepts RexxValue text and reports standard `RXC-LC-40.*` context errors. Level B measures codepoints. Level C measures octets in BYTE and codepoints in UTF8, including the one-unit pad rule. See the separate Level B OVERLAY and Level C OVERLAY pages.

```
OVERLAY(' ', 'abcdef', 3)      == 'ab def'
OVERLAY('.', 'abcdef', 3, 2)    == 'ab. ef'
OVERLAY('qq', 'abcd')         == 'qqcd'
OVERLAY('qq', 'abcd', 4)       == 'abcqq'
OVERLAY('123', 'abc', 5, 6, '+') == 'abc+123+++'
OVERLAY('foo', 'abcdef', 3, 0) == 'abcdef'
```

33.46 POS(needle, haystack [,start])

returns the position of the string *needle*, in *string* (the "haystack"), searching from left to right. If the string *needle* is not found, or is the null string, 0 is returned. By default the search starts at the first character of *string* (that is, *start* has the value 1). This may be overridden by specifying *start* (which must be a positive whole number), the point at which to start the search; if *start* is greater than the length of *string* then 0 is returned.

The Level B helper types *start* as `.int` and signals `INVALID_ARGUMENTS` for a non-positive value. The standalone Level C BIF accepts Classic whole-number text and reports standard `RXC-LC-40.*` context errors. Level B returns Unicode codepoint positions. Level C returns octet positions in BYTE and codepoint positions in UTF8. See the separate Level B POS and Level C POS pages for their contracts and direct search implementation.

```
POS('day', 'Saturday')    == 6
POS('x', 'abc def ghi')   == 0
POS(' ', 'abc def ghi')   == 4
POS(' ', 'abc def ghi', 5) == 8
```

33.47 LASTPOS(needle, haystack [,start])

Returns the position of the last occurrence of *needle* whose final character unit is at or before *start*. The search is case-sensitive. Level B positions are 1-based Unicode codepoints; Level C positions are octets in BYTE and codepoints in UTF8. When *start* is omitted, the complete haystack is considered. A value beyond the haystack has the same effect as omission. A null or absent needle returns `0`.

start must be a positive whole number when supplied. The Level B helper types it as `.int` and signals `INVALID_ARGUMENTS` for an invalid value. The standalone Level C BIF accepts Classic whole-number text and reports standard `RXC-LC-40.*` context errors. See the separate Level B LASTPOS and Level C LASTPOS pages.

```
LASTPOS(' ', 'abc def ghi')    == 8
LASTPOS(' ', 'abc def ghi', 7) == 4
LASTPOS('abc', 'abc abc', 6)   == 1
LASTPOS('aa', 'aaa')           == 2
LASTPOS('', 'anything')        == 0
```

33.48 RIGHT(string, length [,pad])

returns a string of length *length* containing the right-most *length* characters of *string* - that is, padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank. *length* must be a non-negative whole number. A supplied pad must contain exactly one character.

The Level B helper requires `.int` *length* and signals `INVALID_ARGUMENTS` for an invalid length or pad. The standalone Level C BIF accepts Classic whole-number text and reports standard `RXC-LC-40.*` context errors. Level B counts Unicode codepoints. Level C counts exact octets in BYTE and codepoints in UTF8. See the separate Level B RIGHT and Level C RIGHT pages.

```
RIGHT('abc  d', 8)  == '  abc  d'
RIGHT('abc def', 5) == 'c def'
RIGHT('12', 5, '0')  == '00012'
```

33.49 REVERSE(string)

Returns a copy of *string* with its character units in reverse order. Level B reverses Unicode codepoints, so combining marks are independent and grapheme clusters are not preserved as units. Level C reverses exact octets in BYTE and Unicode codepoints in UTF8.

The Level B helper accepts and returns `.string` and has no domain error for valid text. The standalone Level C BIF accepts RexxValue text and reports standard `RXC-LC-40.*` argument errors. Both use a single reverse pass over their active unit representation. See the separate Level B REVERSE and Level C REVERSE pages.

```
REVERSE('abc')    == 'cba'
REVERSE('□□aé') == '□□éa'
REVERSE('')       == ''
```

33.50 SPACE(string [,count [,pad]])

Returns a copy of *string* with its blank-delimited words joined by exactly *count* copies of *pad*. Level B uses Unicode 17.0.0 `White_Space`. Level C BYTE uses ASCII space plus configured blank octets; Level C UTF8 uses Unicode `White_Space` plus configured blank codepoints. Leading/trailing blanks are removed and each internal blank run becomes the requested separator. *count* must be non-negative; zero joins words directly. The default count is one and the default pad is blank. A supplied pad must contain one active character unit.

The Level B helper takes `.string`, an optional `.int` count, and `.string` pad, and signals `INVALID_ARGUMENTS` for an invalid count or pad. The standalone Level C

BIF accepts RexxValue text and reports standard `RXC-LC-40.*` context errors. Both scan the source once. See the separate Level B SPACE and Level C SPACE pages.

```
SPACE('abc  def  ')        == 'abc def'
SPACE('  abc def ', 3)     == 'abc   def'
SPACE('abc  def  ', 1)     == 'abc def'
SPACE('abc  def  ', 0)     == 'abcdef'
SPACE('abc  def  ', 2, '+') == 'abc++def'
```

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

Returns a copy of *string* with a leading, trailing, or both leading and trailing runs removed. The first codepoint of *option* is `L`, `T`, or `B` respectively, case-insensitively; the default is `B`.

When *char* is omitted, Level B removes Unicode 17.0.0 `White_Space`; Level C BYTE removes ASCII space plus configured blank octets and Level C UTF8 removes Unicode `White_Space` plus configured blank codepoints. When supplied, *char* must contain exactly one active unit and only that unit is removed. Thus an explicit blank differs from omission when other configured whitespace occurs at an edge.

The Level B helper accepts strings and signals `INVALID_ARGUMENTS` for an invalid option or supplied char. The standalone Level C BIF reports standard `RXC-LC-40.*` context errors. Both compute one direct source slice. See the separate Level B STRIP and Level C STRIP pages.

```
STRIP('  ab c  ')        == 'ab c'
STRIP('  ab c  ', 'L')   == 'ab c  '
STRIP('  ab c  ', 't')   == '  ab c'
STRIP('12.70000', 't', '0') == '12.7'
STRIP('0012.700', 'b', '0') == '12.7'
```

33.52 SUBSTR(string, start [,length [,pad]])

Returns the substring of *string* beginning at the positive 1-based position *start*. Level B measures Unicode codepoints. Level C measures exact octets in BYTE and codepoints in UTF8. When *length* is omitted, the result continues through the end, or is empty when *start* is beyond the source. A supplied *length* must be

non-negative and fixes the result width; missing source units are replaced with *pad*. The default pad is blank and a supplied pad must contain exactly one active unit.

The Level B helper requires `.int` start/length values and signals `INVALID_ARGUMENTS` for invalid values. The standalone Level C BIF accepts Classic whole-number text and reports standard `RXC-LC-40.*` context errors. Both leave the source unchanged; their active units differ as described above. See the separate Level B SUBSTR and Level C SUBSTR pages.

```
SUBSTR('abc', 2)       == 'bc'
SUBSTR('abc', 2, 4)     == 'bc  '
SUBSTR('abc', 5, 4)     == '    '
SUBSTR('abc', 2, 6, '.') == 'bc....'
SUBSTR('abc', 5, 6, '.') == '......'
```

Note: In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting sub-strings, especially if more than one sub-string is to be extracted from a string.

33.52.1 SUBSTRO

`SUBSTRO` is a cREXX-specific Level B alternate name with the same typed, signal-based slicing behavior as `SUBSTR`. It has a standalone direct VM implementation because it remains a bootstrap-library export, but it is not a Level C BIF. See Level B SUBSTRO.

33.53 SUBWORD(s, start, n)

returns the sub-string of *string* that starts at the **n*th word, and is up to* length* blank-delimited words long. *n* must be a positive whole number; if greater than the number of words in the string then the null string is returned. *length* must be a non-negative whole number. If *length* is omitted it defaults to be the remaining words in the string. The returned string will never have leading or trailing blanks, but will include all blanks between the selected words.

```
SUBWORD('Now is the  time', 2, 2) == 'is the'
SUBWORD('Now is the  time', 3)   == 'the  time'
SUBWORD('Now is the  time', 5)   == ''
```

33.54 TRANSLATE(string [,outputTable [,inputTable [,pad]]])

returns a copy of *string* with each character in *string* either unchanged or translated to another character.

The **translate** function acts by searching the input translate table, *tablei*, for each character in *string*. If the character is found in *tablei* (the first, leftmost, occurrence being used if there are duplicates) then the corresponding character in the same position in the output translate table, *tableo*, is used in the result string; otherwise the original character found in *string* is used. The result string is always the same length as *string*.

The translate tables may be of any length, including the null string. The output table, *tableo*, is padded with *pad* or truncated on the right as necessary to be the same length as *tablei*. The default *pad* is a blank.

When both tables are omitted, TRANSLATE applies the active profile's uppercase mapping. When the output table is supplied and the input table is omitted, Level B uses its fixed U+0000 through U+00FF codepoint domain and Level C BYTE uses its exact `00` through `FF` XRANGE. Level C UTF8 rejects that form because Classic XRANGE is not a Unicode range. Level B tables are codepoint based; Level C table units follow its BYTE or UTF8 profile. See the separate Level B API and Level C BIF contract.

```
TRANSLATE('abbc', '&', 'b')           == 'a&&c'
TRANSLATE('abcdef', '12', 'ec')       == 'ab2d1f'
TRANSLATE('abcdef', '12', 'abcd', '.') == '12..ef'
TRANSLATE('4123', 'abcd', '1234')     == 'dabc'
TRANSLATE('4123', 'hods', '1234')     == 'shod'
```

Note: The last two examples show how the **translate** function may be used to move around the characters in a string. In these examples, any 4-character string could be specified as the first argument and its last character would be moved to the beginning of the

```
TRANSLATE('gh.ef.abcd', 19970827, 'abcdefgh')
```

(which returns "**27.08.1997**") shows how a string (in this case perhaps a date) might be re-formatted and merged with other characters using the **translate** function.

33.55 UPPER(string)

Returns a copy of *string* after applying Level B's locale-independent, limited simple uppercase table. Covered letters are replaced by their uppercase equivalents; other codepoints are unchanged. This is deliberately not full Unicode case folding. The surface accepts only the string argument; it has no substring position or length options.

The helper accepts and returns `.string`, performs the runtime's limited simple mapping, and has no error branch for valid text. See Level B UPPER for its exact contract and implementation notes. UPPER is not a required Level C BIF in the repository catalog; the existing common-runtime helper is compatibility surface only.

```
UPPER('Fou-Baa') == 'FOU-BAA'
UPPER('äöüé')    == 'ÄÖÜÉ'
UPPER('')        == ''
```

33.56 VERIFY(string, reference [,option [,start]])

verifies that *string* is composed only of characters from *reference*, by returning the position of the first character in *string* that is not also in *reference*. If all the characters were found in *reference*, 0 is returned. The *option* may be either **'Nomatch'** (the default) or **'Match'**. Only the first character of *option* is significant and it may be in uppercase or in lowercase. If **'Match'** is specified, the position of the first character in *string* that **is** in *reference* is returned, or 0 is returned if none of the characters were found. The default for *start* is 1 (that is, the search starts at the first character of *string*). This can be overridden by giving a different *start* point, which must be positive. If *string* is the null string, the function returns 0, regardless of the value of the *option*. Similarly if *start* is greater than *string***.length**, 0 is returned. If *reference* is the null string, then the returned value is the same as the value used for *start*, unless **'Match'** is specified as the *option*, in which case 0 is returned.

The Level B helper types *start* as `.int`; an empty or invalid option and a non-positive start signal `INVALID_ARGUMENTS`. The standalone Level C BIF accepts Classic whole-number text and reports standard `RXC-LC-40.*` context errors. Level B

reports codepoint positions. Level C reports octet positions in BYTE and codepoint positions in UTF8. See the separate Level B VERIFY and Level C VERIFY pages for their distinct contracts.

```
VERIFY('123', '1234567890')          == 0
VERIFY('1Z3', '1234567890')          == 2
VERIFY('AB4T', '1234567890', 'M')     == 3
VERIFY('1P3Q4', '1234567890', 'N', 3)  == 4
VERIFY('ABCDE', '', 'n', 3)            == 3
VERIFY('AB3CD5', '1234567890', 'm', 4) == 6
```

33.57 WORD(s, n)

returns the n-th blank-delimited word in *string*. *n* must be positive. If there are fewer than *n* words in *string*, the null string is returned. This function is exactly equivalent to *string*.**subword**(*n*,**1**).

```
WORD('Now is the time', 3) == 'the'
WORD('Now is the time', 5) == ''
```

33.58 WORDINDEX(s, n)

returns the character position of the **n*th blank-delimited word in* string. n* must be positive. If there are fewer than *n* words in the string, 0 is returned.

```
WORDINDEX('Now is the time', 3) == 8
WORDINDEX('Now is the time', 6) == 0
```

33.59 WORDLENGTH(s, n)

returns the length of the **n*th blank-delimited word in* string. n* must be positive. If there are fewer than *n* words in the string, 0 is returned.

```
WORDLENGTH('Now is the time', 2)    == 2
WORDLENGTH('Now comes the time', 2) == 5
WORDLENGTH('Now is the time', 6)    == 0
```

33.60 WORDS(s)

returns the number of blank-delimited words in *string*.

```
WORDS('Now is the time') == 4
WORDS(' ')               == 0
WORDS('')                == 0
```

33.61 DATATYPE(s [,type])

With *type* omitted, returns `NUM` for a syntactically valid Level B number and `CHAR` otherwise. With *type* present, returns 1 if *string* matches the requested description, or 0 otherwise. The null string is valid for the `B` and `X` tests and false for the other explicit tests.

Only the first character of *option* is significant, and it may be in either uppercase or lowercase. The following *option* characters are recognized:

A (Alphanumeric); returns 1 if *string* only contains characters from the ranges "a-z", "A-Z", and "0-9".

B (Binary); returns 1 if *string* only contains the characters "0" and/or "1".

D (Digits, CREXX Level B extension); returns 1 if *string* only contains characters from the range "0-9".

L (Lowercase); returns 1 if *string* only contains characters from the range "a-z".

M (Mixed case); returns 1 if *string* only contains characters from the ranges "a-z" and "A-Z".

N (Number); returns 1 if *string* is a syntactically valid CREXX number that could be added to **'0'** without error,

S (Symbol); returns 1 if *string* only contains characters that are valid in non-numeric symbols (the alphanumeric characters and underscore), and does not start with a digit. Note that both uppercase and lowercase letters are permitted.

U (Uppercase); returns 1 if *string* only contains characters from the range "A-Z".

W (Whole Number); returns 1 if *string* is a syntactically valid CREXX number that can be added to **'0'** without error, and whose decimal part after that addition, with no rounding, is zero.

X (heXadecimal); returns 1 if *string* only contains characters from the ranges "a-f", "A-F", and "0-9".

```
DATATYPE('101', 'B')    == 1
DATATYPE('12.3', 'D')   == 0
DATATYPE('12.3', 'N')   == 1
DATATYPE('12.3', 'W')   == 0
DATATYPE('LaArca', 'M') == 1
DATATYPE('', 'M')       == 0
DATATYPE('Llanes', 'L') == 0
DATATYPE('3 d', 's')    == 0
DATATYPE('BCd3', 'X')   == 1
DATATYPE('BCgd3', 'X')  == 0
```

Note: Level B traverses codepoints safely but deliberately uses the listed ASCII character classes. Its `W` test is exact and has no floating-point tolerance. An explicit empty or unsupported option signals `INVALID_ARGUMENTS`. The separate Classic Level C BIF excludes `D` and obtains extra letter/digit mappings, B/X blanks, and exponent limits from its call configuration; see `lib/rxfnsc/datatype.md`.

33.62 RANDOM(min, max, seed)

Returns a pseudo-random integer in the selected inclusive range. With no arguments the range is `0..999`. One argument is the inclusive maximum, so `RANDOM(10)` selects `0..10`. In the positioned three-argument form an omitted minimum defaults to `0` and an omitted maximum defaults to `999`.

A supplied non-negative seed resets the module-scoped sequence before drawing the result; omitted seeding continues that sequence. Negative arguments, reversed bounds, and a range wider than `100000` signal `INVALID_ARGUMENTS`.

Level B uses deterministic Park-Miller state and unbiased rejection sampling, not the VM-global `irand` instruction. It is suitable for repeatable language- level sequences, not cryptography. See the separate Level B contract and Level C contract; Level C owns its state on the call configuration and reports the standard `40.31` through `40.33` errors.

33.63 FNV(string)

Returns the conventional 32-bit FNV-1a hash of the string's exact UTF-8 byte sequence as an integer in `0..4294967295`.

```
FNV("")    == 2166136261
FNV("a")   == 3826002220
FNV("abc") == 440920331
```

Embedded NUL bytes participate in the hash. FNV does not mutate its input and does not signal numeric overflow because modulo-2^32 arithmetic is part of the algorithm. It is a Level B library function, not a Level C Classic BIF. See the Level B contract.

33.64 TIME(option)

Returns time information. The option is case-insensitive and defaults to `N`.

Option	Result
`N`	Local time as `hh:mm:ss`.
`L`	Local time as `hh:mm:ss.ffffff`.
`H`	Hour since midnight.
`M`	Minutes since midnight.
`S`	Seconds since midnight.
`US`	Microseconds since midnight.
`E`	Elapsed seconds.
`R`	Reset/read the elapsed timer.
`C`	Civil-style time with `am` or `pm`.
`UTC`	UTC time using the normal `hh:mm:ss` format.
`ZD`	UTC offset in seconds.
`T`	CPU ticks since program start.
`TS`	Ticks per second.
`ZN`	Time zone name.

An unsupported option signals `INVALID_ARGUMENTS`. This typed Level B extension is documented separately from the Classic three-argument Level C BIF in the Level B API and Level C contract.

33.65 DATE(oformat, date, iformat, osep, isep)

Converts dates between supported date formats. With no `date` argument, it uses the current local date. Empty `oformat` or `iformat` values default to `NORMAL`; `osep` and `isep` can override the output or input separator.

Input formats are matched by abbreviation and currently include `NORMAL`, `STANDARD`, `ORDERED`, `EUROPEAN`, `GERMAN`, `USA`, `INTERNATIONAL`, `QUALIFIED`, `JULIAN`, `BASE`, `UNIX`, and `EPOCH`. `NORMAL` and `QUALIFIED` input dates accept full or abbreviated English month names; this month-prefix matching belongs to DATE and does not change the exact word matching performed by `WORDPOS`.

Output formats are also matched by abbreviation and currently include `NORMAL`, `XNORMAL`, `STANDARD`, `ORDERED`, `XORDERED`, `EUROPEAN`, `XEUROPEAN`, `GERMAN`, `XGERMAN`, `USA`, `XUSA`, `INTERNATIONAL`, `QUALIFIED`, `JULIAN`, `DAYS`, `WEEKDAY`, `MONTH`, `CENTURY`, `BASE`, `UNIX`, `JDN`, `EPOCH`, `DEC`, and `XDEC`.

33.66 SEQUENCE(from, to)

Returns the sequence of characters from `from` through `to`, using Unicode codepoint values. It is the Unicode-capable replacement for byte-oriented `XRANGE`; unlike `XRANGE`, it does not wrap around when `from` is greater than `to`. Both endpoints must be single characters. A descending range or an invalid endpoint signals `INVALID_ARGUMENTS`; surrogate code points are skipped because they are not Unicode scalar values. See the stable Level B API.

33.67 XRANGE(start, end)

The typed Level B helper returns the inclusive U+0000 through U+00FF byte-domain character range and wraps at U+00FF. Both endpoints are required, must contain exactly one character, and invalid endpoints signal `INVALID_ARGUMENTS`. It is retained for legacy byte-range use; `SEQUENCE` is the non-wrapping Unicode range API.

Classic Level C `XRANGE([start [,end]])` is different. In the default BYTE profile it returns the inclusive wrapping exact-byte range, defaulting to `00` through `FF`. It is

intentionally unavailable in UTF8 because it is not a Unicode scalar or grapheme range. See the separate Level B API and Level C contract.

33.68 TRACE(option)

The SAA `TRACE(option)` built-in function name is reserved for compatibility. In the current beta, use the `TRACE` statement to set tracing:

```
trace off
trace normal
trace results
trace value option
```

`TRACE VALUE option` evaluates `option` at runtime and applies the same option rules as a static `TRACE` statement. See the `TRACE` statement reference for the supported modes, output targets, and namespace suppression controls.

The Level C library now also contains a standalone direct `RexxValue` implementation of the Classic `TRACE([option])` query/update contract. It is covered through a direct library harness, but normal compiled calls are not yet lowered to that entry point; that wiring is intentionally deferred to the later bulk Level C lowering change. Its library contract is documented in `lib/rxfnsc/trace.md`.

33.69 VALUE(name, newvalue, pool)

There are two intentionally different library contracts.

The Level B helper is read-only:

```
import rxfnsb
count = 12
say value("count")    /* 12 */
say value("missing")  /* MISSING */
```

It accepts one `.string` name, searches only the immediate caller procedure's scalar/constant metadata, and returns a `.string`. It does not assign variables or implement Classic stems. See `lib/rxfnsb/rexx/value.md`.

The standalone Level C BIF implements `VALUE(name [,newvalue [,pool]])` over `RexxValue` and the caller `RexxVariablePool`. The internal form expands compound-

variable tails, returns the old value, and optionally assigns the new value. Invalid internal symbols use error `40.26`.

When a third argument is present, Level C resolves its trimmed, case-insensitive name through the external-pool registry on `RexxClassicConfig`. The subject name is passed to the selected adapter unchanged. External assignment first gets and returns the old value, then sets the new value; a missing subject is not implicitly created. Adapter rejection reports `40.36`, and a blank or unknown pool reports `40.37`. The exact direct- call contract and test scope are documented in the Level C VALUE page. Compiler lowering to the direct entry point remains part of the later bulk Level C lowering change.

33.70 VERSION()

Returns a string with implementation and build information supplied by the VM `rxvers` instruction. The current string layout is:

```
platform bits crexx-version build-date
```

`platform` is one of the VM's compiled platform names: `linux`, `windows`, `macOS`, `cms`, or `unknown`. `bits` is `32` or `64`; `crexx-version` starts with `crexx-` and may contain build metadata; and `build-date` is `yyyymmdd`. The function does not currently expose endianness as a separate field. Its selector-local Level B contract is in `lib/rxfnsb/rexx/version.md`.

33.71 ABBREV(info, word, length)

Returns 1 if *word* is equal to the leading characters of *info* and *word* is not less than the minimum length, *length*; 0 is returned if either of these conditions is not met. *length* must be a non-negative whole number; the default is the length of *word* in the Level C contract. The Level B helper uses an integer default of zero, which produces the same result for an omitted minimum because the entire candidate must still match.

```
ABBREV('Print', 'Pri')   == 1
ABBREV('PRINT', 'Pri')   == 0
ABBREV('PRINT', 'PRI', 4) == 0
ABBREV('PRINT', 'PRY')   == 0
```

```
ABBREV('PRINT', '')      == 1
ABBREV('PRINT', '', 1)    == 0
```

Note: A null string will always match if a length of 0 (or the default) is used. This allows a default keyword to be selected automatically if desired.

The typed Level B and direct `RexxValue` contracts are documented separately in `lib/rxfnsb/rexx/abbrev.md` and `lib/rxfnsc/abbrev.md`. Level B measures the prefix and minimum in codepoints. Level C measures octets in BYTE and codepoints in UTF8.

```
say 'Enter option:';  option=ask
select  /* keyword1 is to be the default */
  when ABBREV('keyword1', option) then ...
  when ABBREV('keyword2', option) then ...
     ...
  otherwise ...
  end
```

33.72 DIGITS()

Returns the positive number of significant decimal digits in the current procedure's numeric context. The value is controlled by `NUMERIC DIGITS`; a called BIF observes the setting inherited from its immediate caller.

```
numeric digits 12
say DIGITS()  /* 12 */
```

`DIGITS` accepts no arguments. The distinct typed Level B and direct RexxValue contracts are documented in Level B numeric accessors and Level C numeric BIFs.

33.73 FORM()

Returns `SCIENTIFIC` or `ENGINEERING`, identifying the exponential notation selected by the current procedure's `NUMERIC FORM` setting.

```
numeric form engineering
say FORM()  /* ENGINEERING */
```

`FORM` accepts no arguments. Level B's typed helper intentionally returns the lowercase CREXX name, while the Level C BIF returns the uppercase Classic BIF result. See the separate Level B and Level C contracts.

33.74 FUZZ()

Returns the non-negative number of least-significant digits ignored during numeric comparisons in the current procedure. The value is controlled by `NUMERIC FUZZ` and is always smaller than `DIGITS()`.

```
numeric digits 12
numeric fuzz 2
say FUZZ()  /* 2 */
```

`FUZZ` accepts no arguments. Its typed Level B and direct RexxValue contracts are documented separately in Level B numeric accessors and Level C numeric BIFs.

PART IV

Appendices

A

cREXX Level B Authoring Guide For Agents

Use this guide when you need to write or edit Level B/Level G `.crexx` in this repository. Generic training data about "REXX" is often too vague, too classic-Rexx oriented, or simply wrong for CREXX Level B as it exists in this tree.

This guide is intentionally grounded in code that already compiles here. When in doubt, copy the nearest repo pattern instead of inventing syntax.

A.1 What To Trust First

For Level B authoring, these sources are more reliable than model memory:

- `docs/ai-context/CREXX_ARCHITECTURE.md`
- `docs/ai-context/CREXX_LIBS.md`
- `docs/books/crexx_language_reference/classes_and_interfaces.md`
- `docs/books/crexx_language_reference/data_types.md`
- `docs/books/crexx_language_reference/statements.md`
- nearby working `.crexx` files in `lib/`, `bin/`, `compiler/exits/`, `debugger/`, and `tests/`

A.2 Repo-Native Level B Patterns

A.2.1 1. Small library function

Use explicit `options levelb`, a namespace, and an exposed symbol:

```
options levelb

namespace rxfnsb expose abs

abs: procedure = .string
arg number = .string
if left(number,1) = '-' then number = substr(number,2)
return number
```

Reference: - `lib/rxfnsb/rexx/abs.crexx`

A.2.2 2. Top-level script

Small scripts can use top-level `arg` directly instead of a `main:` routine:

```
options levelb
import rxfnsb

arg searches = .string[]

if searches.0 < 1 then searches.1 = ".crexx"
ordered_searches = .string[]
address crexx "echo .crexx" output ordered_searches
```

References: - `tests/demo/countlines.crexx` - `compiler/exits/address/test_-address.crexx`

A.2.3 3. Tool-style entry point with procedure-exposed module state

Longer tools often use `main:` plus procedure-level `expose` for module state:

```
options levelb
namespace rxdb expose stephandler
import rxfnsb
import globals

main: procedure = .int expose next_instruction last_instruction mode
    arg cmd_line = .string[]
```

The `expose` list belongs to that one procedure. Other local procedures see the same module-global storage only if they also list the variable:

```
proc1: procedure = .void expose var
    var = "Hello World"
    return

proc2: procedure = .void expose var
    say "var is" var
    return
```

Reference: - `debugger/rxdb.crexx`

A.2.4 4. Mutating an exposed argument

When you really need caller-owned state, use `arg expose ...`:

```
pushidentifier: procedure = .int
    arg expose tokens = .token[], text = .string, value_type = ".
        unknown"
    index = tokens[0] + 1
    tokens[index] = .token(...)
    return index
```

Reference: - `compiler/exits/ExitTestSupport.crexx`

A.3 Level B Differences That Matter In Practice

A.3.1 Types are normal, not exceptional

In this repo, typed procedures and typed `arg` declarations are the normal Level B style. Do not default to untyped classic-Rexx-looking code when you are editing standard libraries, tools, exits, or tests.

Common examples in-tree:

- `procedure = .int`
- `procedure = .string`
- `procedure = .token[]`
- `arg name = .string`
- `arg cmd_line = .string[]`
- `arg expose tokens = .token[]`

Use bare type expressions to declare a typed local before later assignment when that makes scope or intent clearer:

```
slot = .int
if map.containsKey(key) then slot = existingSlot(key)
else slot = createSlot(key)
```

Prefer this to a dummy literal such as `slot = 0` when the value is not semantically a sentinel. This is especially useful when a local is assigned in both sides of a branch and then read after the branch.

References:

- `lib/rxfnsb/rexx/abs.crexx`
- `debugger/rxdb.crexx`
- `compiler/exits/ExitTestSupport.crexx`
- `tests/levelbfunc.crexx`

A.4 Library Changes Are Whole-Toolchain Tests

Treat every Level B library change as an opportunity to exercise the complete consumer path: compile with `rxc`, assemble with `rxas`, link with `rxlink`, and execute with `rxvm` (and both concrete VM dispatches when the behavior is VM sensitive). A direct compiler or library-unit check is not sufficient for a public library surface because imported metadata, optimized RXAS shape, link-time symbol resolution, or runtime behavior can fail independently.

For optimizer-sensitive helpers, retain no-opt and optimized executions with the same expected result and inspect the optimized RXAS when the intended shape matters. Supported inlining shapes must be repaired when they miscompile; an implementation defect or awkward generated shape is not a reason to fail closed. Falling back to a real call is reserved for cases where ordinary-call equivalence is mathematically impossible or cannot be established from the language semantics and available facts.

A.4.1 Namespace/import syntax is part of the real language surface

Do not treat namespace use as documentation sugar. It is part of how source is validated, imported, and linked.

Important points:

- file-level `namespace` and `import` clauses are real inputs to compilation
- `namespace..symbol` is the canonical qualified-reference form
- `namespace::symbol` is accepted as a compatibility alias, but do not prefer it
- the token to the left of `..` must be an imported namespace

References:

- `docs/ai-context/CREXX_ARCHITECTURE.md`
- `docs/books/crexx_programming_guide/intralanguage.md`
- `docs/books/crexx_language_reference/classes_and_interfaces.md`

A.4.2 Keep the leading file header conventional

The compiler does a lightweight pre-scan of the leading `options`, `namespace`, and `import` clauses before full parsing. Preserve that structure when editing existing files.

Practical guidance:

- keep `options levelb` near the top
- keep `namespace` and `import` in the leading header block
- do not bury them later in the file

Reference: - `docs/ai-context/CREXX_ARCHITECTURE.md` - `docs/books/crexx_programming_-guide/rxc.md`

A.4.3 Procedure-level expose and namespace auto-bind

There are two ways a Level B procedure can intentionally bind a module-global variable:

- file-level `namespace name expose var` exposes the symbol from the module and automatically binds it into local `procedure` scopes in that source file
- procedure-level `name: procedure [= .type] expose var ...` binds the listed variables only for that procedure; every local procedure that needs the same storage must list the variable itself

Use procedure-level `expose` for private module state that several local procedures share, or when you are following an existing stateful tool/library pattern. Do not add `procedure expose ...` just out of habit when a namespace-exposed module global is what you actually want.

Use `procedure expose` or `arg expose` when:

- you are intentionally sharing private module state across selected local procedures
- you are mutating a passed-in array/object
- you are following an existing stateful tool pattern such as `rxdb`

References: `- docs/ai-context/CREXX_ARCHITECTURE.md - docs/books/crexx_-programming_guide/global_variables.md - debugger/rxdb.crexx`

A.4.4 Module initializers are private lifecycle procedures

Use a module initializer when module-global state must be constructed before `main`, an embedded public call, or a task-worker request can observe that module instance:

```
options levelb
namespace example expose value

boot: initialiser expose value
  value = 42

read: procedure = .int expose value
  return value
```

A module may declare zero or more initializers. They run once for each mutable module instance, in declaration order. Each initializer has an ordinary namespace-qualified name for metadata and diagnostics, but it is a private lifecycle entry point: source cannot call it, import it as a callable, or list it in the namespace `expose` clause. The `expose` after `initialiser` has the same module-variable binding purpose as procedure-level `expose`; it does not export the initializer.

Initializers accept no `arg` declarations and have an implicit `.void` return. A bare `return` is valid; returning a value is a compile error. If an initializer calls an ordinary procedure in another unready module, the VM initializes that module first. There is no source `requires` list, and a cycle through cross-module initializer calls

fails at runtime.

Use this for module-owned singleton-like objects, tables, caches, and resource managers. The lifetime is one mutable module overlay, not one process-global instance. In particular, persistent task workers each initialize their own overlay once before accepting work.

A.4.5 Named constants are compile-time values

Use `constant NAME = expression` inside an explicit procedure, method, or factory scope for Level B masks, flags, and shared literal payloads that must not allocate runtime storage:

```
flags: procedure = .int
  constant RV_FLAG_STRING = 0x00010000
  constant SAMPLE_BYTES = "41424344"x as .binary
  return RV_FLAG_STRING
```

Do not put `constant` declarations in the file body. That top-level form is a Release 1 design defect being removed because it can imply an implicit `main()` in a module that was meant to be a library.

If a script needs a separate declaration procedure for shared constants, add an explicit `main: procedure` before the executable body. Procedure bodies continue until the next top-level callable boundary, so statements after a declaration procedure belong to that procedure unless a new boundary is present.

The initializer must fold at compile time. Integer constants used as assembler immediates are emitted as literals. An exact `.binary` use of a named constant is emitted through one compiler-private, module-scoped RXAS `.const` alias, so large payload text appears once and every operand resolves to the same RXBIN constant-pool item. This is a read-only operand reference, not a hidden mutable register or runtime copy. Converted or completely folded uses follow their ordinary result lowering. String, decimal, and float constants retain their normal constant-pool machinery.

Generated code that shares a family of constants inside one source module must list each name explicitly in the declaration procedure's `procedure expose` list. Release 1 does not import constants across source, RXAS, or RXBIN module boundaries. Cross-module constants and wildcard expose forms such as `TOKEN_*` are Release 2 ergonomics/design candidates, not Release 1 syntax commitments.

A.4.6 Choose encoded fields or host-native packed items deliberately

Use the established `<at..type>(byte_offset)` surface for portable binary formats. Those integer and float fields have explicit widths and canonical little-endian encoding.

Use `<packed..int>(item_index)` or `<packed..float>(item_index)` only for host-local numeric storage where the exact VM representations are required:

```
values = .binary
call binresize values, 3 * 8

<packed..float>(0) values = 100.0
<packed..float>(1) values = 125.5
say <packed..float>(1) values
```

Packed indexes are zero-based item numbers. `.int` means the exact host `rxinteger` representation and `.float` means the exact host VM `double` representation. The buffer has no stored type tag: the same bytes can be read through either packed view. The buffer length and `binresize` remain byte based, and a packed store never resizes it. Packed access is therefore unsuitable for files, protocols, persistent cross-platform data, or data exchanged between hosts with different native representations.

Only `.int` and `.float` are valid packed suffixes in Release 1. Ordinary binary allocation provides native alignment, readable binary constants are materialized into aligned runtime storage, and packed constants remain read-only. Invalid or overflowing indexes and partial trailing items raise `OUT_OF_RANGE` before any write occurs.

A.4.7 Explicit register views are system-programmer syntax

Normal classes should use ordinary attributes. Runtime and VM-integration classes may map attributes to fixed register storage:

```
  _string = .string with register.0.string
  _flags = .int with register.0.flags.library
```

`register.0` is the containing value, not RXAS attribute zero. Duplicate typed views over the same physical `register.N` slot are complex attributes. The compiler copies only the selected typed payload view across the link boundary; library/user cache flags are explicit runtime code, not hidden compiler side effects. Flag views

are direct status-word views: `.flags.vm`, `.flags.compiler`, `.flags.language`, and `.flags.readable` are read-only; `.flags.library`, `.flags.user`, and `.flags.public` are writable. Flag views must be `.int`.

`.flags.language` contains protected, language-wide facts such as intrinsic `.string` normalization certificates. Ordinary Level B code may inspect this view but cannot assign it. A trusted low-level language implementation may use explicit RXAS `GETANDTP`/`SETORTP` operations to consume or publish a proven fact; doing so without proving the complete current string contents violates the runtime contract.

A.4.8 Arrays are first-class Level B objects

Do not reason about them as loose classic stem variables only.

Patterns used in-tree:

- `items = .string[]`
- `items[0]` for count/cardinality reads only
- `items[1]` for first element
- `items.1` is also used in older code
- both bracket and dot indexing appear in the repo, so preserve the local style
- BIFs or helpers that resize/mutate a caller-owned array must use `arg expose items = .type[]`; otherwise they work on a local value copy

Do not initialise or resize an array by assigning to index `0`. The cardinality slot is read-only source syntax, and writes such as `items[0] = "3"` are rejected with `OUT_OF_RANGE`. Use ordinary element assignment or the standard array helpers instead:

```
items = .string[]
call arrayappend items, "alpha"
call arrayappend items, "beta"
say items[0]
```

To clear an existing array object, use the standard in-place helpers:

```
call arraydrop items
call objectarraydrop objects
```

This is especially important for class attributes. Reassigning an attribute-like name to `.string[]` or `.object[]` inside a method is not the same documented operation as clearing the existing array object, and can run into current Level B scop-

ing edge cases. The collection classes use `arraydrop` and `objectarraydrop` in their `clear`/`free` methods.

When a test needs delete/insert semantics, prefer library helpers such as `arraydelete`, `arrayinsert`, and `arrayappend` over assembler unless the test is specifically about RXAS.

The `array*` helper family is currently for `.string[]` arrays. Do not use it for `.int[]`, `.decimal[]`, or other typed numeric arrays; use direct indexing and explicit loops until the project adds a typed/generic helper surface.

Use the `objectarray*` helper family for mutating `.object[]` arrays:

- `objectarrayinsert`,
- `objectarraydelete`,
- `objectarrayappend`,
- `objectarrayprepend`,
- `objectarraydrop`, and
- `objectarraymove`.

Store concrete class instances with an explicit `as .object` upcast.

References:

- `tests/demo/countlines.crexx`
- `compiler/exits/ExitTestSupport.crexx`
- `docs/books/crexx_language_reference/data_types.md`

A.4.9 Class and interface factories omit return types

Use `*: factory` or `name: factory`; do not write `= .type` on a factory. The factory result is inferred from the owning contract: an interface factory returns that interface and a class factory returns that concrete class. In a class factory, bare `return` returns the constructed object, so there is no source-level `this` value to mention.

When creating an object value, include the factory call brackets. `.SomeClass()` returns an initialized instance. Bare `.SomeClass` is a typed default object value and is intentionally uninitialized; it is useful for type defaults and can be probed with `initialized(value)`, but calling methods on it raises `OBJECT_NOT_INITIALIZED`.

```
vehicle: interface
```

```
  *: factory
  arg name = .string
  describe: method = .string

car: class implements .vehicle
  _name = .string

  *: factory
    arg name = .string
    _name = name
    return

  describe: method = .string
    return _name
```

Reference: - `docs/books/crexx_language_reference/classes_and_interfaces.md`

When consuming a class from an imported binary namespace, prefer the explicit qualified form if the object will be used for method calls in a tool or runner:

```
runner = .rexxscript..rexxscriptevaluator()
```

During the RexxScript runner work, the unqualified imported factory form was accepted but method typing on the result was not preserved in that context. If you see `#METHOD_NOT_FOUND` after an imported factory call that should be valid, try the qualified form and record the source-import/binary-import mismatch as a Level B follow-up rather than hiding it in application logic.

A.4.10 Call out suspected Level B gaps

Level B is new enough that RexxScript and runtime-library work may uncover compiler or runtime gaps. Prefer documenting and surfacing these for resolution decisions over burying workarounds without explanation.

Recent observations from the RexxScript evaluator refactor:

- Assigning a new array object to a class attribute from inside a method, such as `items = .string[]`, can shadow the attribute instead of resetting the instance array. Reuse the instance array with an explicit count until the intended attribute-reset pattern is settled.
- A method-local first assigned only inside a nested `do` block is not visible later in the outer method. Declare it before the block when the value is read later. A later implicit assignment or taken-constant read with the same name re-

mains legal, but receives `#NOT_IN_SAME_SCOPE`; an explicit declaration of the separate later variable suppresses that warning. The warning is lexical and deliberately path-independent, so an early `return` or mutually exclusive branch does not suppress it; use an explicit declaration when the separate binding is intentional.

- Passing an object to a helper after a mutating method call, or passing a mutated-object method call inline as an argument, may expose stale state in some Level B paths. Snapshot state inside the owning method when correctness depends on the just-mutated object, and raise a focused language/runtime issue.
- A mutating same-receiver helper can be called as a statement, such as `call clear()`, when the return value is not needed. It can also return a scalar for assignment into a receiver attribute, such as `root = rebalance(root)`. The optimizer must preserve normal call ordering: evaluate the callee return expression, copy the mutated receiver back, then assign the returned value into the caller target. The regression test `inline_test_mutating_method_scalar_attr_return` covers this order. If an older beta 3 WIP build shows stale receiver copyback behavior, split through a local as a temporary diagnostic workaround:

```
next_root = rebalance(root)
root = next_root
```

When a helper naturally needs to update a caller-owned slot, prefer an explicit reference output location such as `reference root` or `reference left[parent]`, then dereference it inside the helper and assign the linked local.

A.4.11 Keep hot Level B loops honest with RXAS

Inlining correctness and inlining performance are separate questions. For performance-sensitive classlib code, inspect the generated optimized `.rxas` and, when relevant, `rxdas` output from the assembled `.rxbin` before assuming a private helper is free.

The `StringTreeMap` AVL rewrite found that a private `_findNode()` helper was correctly inlined into `get()` and `containsKey()`, but still left block-expression scaffolding around the hot lookup loop. Rewriting those two methods as direct loops made Release lookup time for 2,500 entries drop from about 200 ms to about 2.3 ms in the local benchmark. Use helpers for clarity unless the code is a measured

hot path; for hot paths, prefer direct loops when the RXAS shows avoidable call or block-expression structure.

For array-backed data structures, expect ordinary indexed attribute access to lower through `linkattr1`, `minattrs`, typed copy, and `unlink` instructions. That is the current cost model. It is fine to use small inline assembler assists such as `SETATTRS array,0` for array clearing until the language has a typed array-clear surface, but record broader array-access needs as language/runtime backlog items instead of baking large hand-written RXAS into classlib code.

A.4.12 `address crexx` is the standard command-environment pattern

When Level B code needs command-environment behavior, copy the repo pattern instead of inventing a new API shape:

```
out = .string[]
err = .string[]
address crexx "echo #42" output out error err
if rc <> 0 then say "command failed"
```

`address crexx` uses the CREXX command environment. It is the default ADDRESS environment, owns cREXX-specific, OS-independent commands such as `echo`, `pwd`, `cd`, `pushd`, `popd`, `ls`, `mkdir`, `copy`, `cat`, `platform`, `pid`, `resolve`, `tcp`, and `batch`, and reports normal command failure through `rc`. It is not a shell: `;`, `&&`, `||`, pipes, shell redirects, and shell expansion are usage errors. Use repeated ADDRESS statements or `address crexx "batch"` with input lines for multiple commands.

For CREXX commands, host-variable anchors can pass data without shell parsing: `:name` exposes a scalar as one command argument, while `:name[]` and `:name.` expose a stem/array as zero or more command arguments. Prefer `[]` in new code because it is visually unambiguous. For direct executable dispatch, build a `.string[]` argument vector and use `address crexx "run :argv[]"`; the `run` command launches from that argv vector rather than flattening and reparsing a command string.

Use `address system`, `address command`, or `address cmd` only when the caller intentionally needs the platform command processor. On POSIX this is standard `sh -c` resolved from the system standard utility path, not the user's `SHELL` environment variable; on Windows it is `%COMSPEC% /D /S /C` with a `cmd.exe` fallback.

Shell built-ins, pipes, redirects, and command chaining belong on this route.

Use `address shell` only when the shell executable is deliberately configured through `CREXX_ADDRESS_SHELL` and optional `CREXX_ADDRESS_SHELL_ARGS`.

Use `address path` only when the caller needs direct executable dispatch. That provider uses the platform process API without shell semantics. On POSIX it argv-parses the command and resolves the executable through process `PATH`; on Windows it uses direct `CreateProcessW` command-line dispatch:

```
address path "rxas -h" output out error err
```

References:

- `compiler/exits/address/test_address.crexx`
- `bin/crexx.crexx`
- `tests/demo/countlines.crexx`

A.4.13 Signal handlers live on simple `do` groups

Block-scoped signal handling is written with `on signal` clauses on a simple `do ... end` group:

```
do
  risky_work()
on signal conversion_error as problem
  say problem.source()
on signal
  call cleanup()
end
```

Do not attach `on signal` directly to counted, conditional, forever, or expression-form `do` loops. To protect part of a loop, nest a simple signal-handling `do ... end` group inside the loop body.

If `as name` is omitted, the handler has no local signal object. That is fine for fixed cleanup/logging handlers.

References:

- `docs/books/crexx_language_reference/statements.md`
- `docs/ai-context/LEVELB_SIGNALS_TRACE_WORKING.md`
- `compiler/exits/signal/test_signal_block.crexx`

A.4.14 Headerless scripts are a driver convenience, not a style rule

The `crexx` driver can compile simple headerless top-level scripts with synthetic defaults (`--level levelb --import rxfnsb`). That is useful for end users, but repo code should usually stay explicit with `options levelb` and imports unless there is a strong reason not to.

Reference:

- `docs/books/crexx_language_reference/tools.md`
- `docs/books/crexx_programming_guide/crexx.md`

A.4.15 Command-line arguments are a first-class Level B feature

Level B programs receive command-line arguments through `arg`, and when the compiler synthesizes the file-level `main()` wrapper, the VM argv payload is available there as a `.string[]`.

Program-side guidance:

- use `arg fn = .string[]` (or a typed equivalent)
- use `fn.0` / `fn[0]` for the count
- iterate over `fn.1`..`fn.fn.0`
- for explicit `main`, model it on in-tree patterns such as `arg cmd_line = .string[]`

Launcher-side guidance:

- do not guess the exact command form from generic model memory
- check the current tool docs/source when documenting `crexx` or `rxvm` invocation syntax

References:

- `docs/books/crexx_language_reference/arguments.md`
- `docs/ai-context/CREXX_ARCHITECTURE.md`
- `debugger/rxdb.crexx`
- `tests/demo/countlines.crexx`

A.5 Wayfinding: Best Example Files By Task

A.5.1 Writing a tiny BIF or helper

- `lib/rxfnsb/rexx/abs.crexx`
- `lib/rxfnsb/rexx/fileio.crexx`
- `lib/rxfnsb/rexx/regex.crexx`

A.5.2 Writing a top-level command/script

- `tests/demo/countlines.crexx`
- `bin/crexx.crexx`
- `compiler/exits/address/test_address.crexx`

A.5.3 Writing a stateful tool

- `debugger/rxdb.crexx`

A.5.4 Writing compiler-exit or structured typed code

- `compiler/exits/ExitTestSupport.crexx`
- `compiler/exits/address/Address.crexx`
- `compiler/exits/parse/Parse.crexx`

A.5.5 Checking argument signature syntax

- `tests/levelbfunc.crexx`
- `compiler/exits/ExitTestSupport.crexx`

A.5.6 Checking namespace/global behavior

- `docs/books/crexx_programming_guide/global_variables.md`
- `docs/books/crexx_programming_guide/global_procedures.md`

A.5.7 Checking argument handling

- `docs/books/crexx_language_reference/arguments.md`
- `debugger/rxdb.crexx`
- `tests/demo/countlines.crexx`

A.6 Agent Guidance

When writing Level B code:

1. Read one doc source and two nearby working `.crexx` examples before editing.
2. Match the local style of the directory you are in.
3. Prefer explicit Level B headers in committed repo code.
4. Prefer canonical `namespace..symbol` qualification.
5. Do not "simplify" typed signatures or namespace structure just because a generic REXX example elsewhere looks looser.

If a proposed snippet does not resemble existing repo code in `lib/`, `debugger/`, `compiler/exits/`, `bin/`, or `tests/`, stop and verify it before committing it.

B

Glossary

BIF Built-in-function. The traditional name for Rexx string functions.

Capability A cohesive unit of business or technical functionality provided by a system. Unlike a service or module, a capability is defined by what it accomplishes rather than how it is implemented.

Cognitive Load The mental effort required to understand, modify, or operate a system. Modern architectural practice attempts to minimize unnecessary cognitive load by limiting dependencies, simplifying interfaces, and exposing only essential details.

Fitness Function An automated test or measurement that continuously verifies that an architectural property remains true. Examples include limits on coupling, response time, dependency counts, or code complexity.

Lowering The process of translating a high-level representation into a more concrete or implementation-oriented one while preserving its meaning. In compilers, lowering typically refers to successive transformations from an abstract syntax tree through intermediate representations to machine instructions. More recently, the term has been adopted in software architecture and AI systems to describe the translation of abstract intent into executable actions.

Ratchet A mechanism that permits progress in one direction while preventing regression. In software engineering, a ratchet is usually an automated process, such as a test, quality gate, or architectural rule which ensures improvements cannot silently be undone. Examples include increasing minimum test cov-

erage, reducing dependency counts, or preventing the introduction of new compiler warnings.

Scalar A scalar is a single, indivisible value that represents one quantity. Unlike compound values (such as arrays, lists, records, or objects), a scalar has no internal components that are directly accessible.

Shape The overall structural form of a system, data model, API, or solution, emphasizing relationships rather than implementation details. The term is intentionally informal and generally refers to architectural organization rather than specific software constructs.

Surface The externally visible portion of a software component through which other components interact with it. Depending on context, this may refer to an API surface, user interface surface, or attack surface. Architects generally seek to minimize exposed surface area in order to reduce complexity, coupling, and maintenance effort.

Bibliography

Blasco, J. M. (May 2023). "The Search Order for External Rexx Files". In: *Proceedings of the 34th International Rexx Language Symposium*. Ed. by R. V. Jansen. Presented 16 May 2023; proceedings published 21 May 2023. Rexx Language Association. Amsterdam, The Netherlands: RexxLA Press, pp. 191–266. isbn: 978-94-036-5010-4.

Cowlishaw, M. (Feb. 1987). "The design of the REXX language". In: *ACM SIGPLAN Notices* 22, pp. 26–35.

Cowlishaw, M. F. (1985). *The REXX language: a practical approach to programming*. Prentice-Hall, Inc.

List of Tables

Index

ISBN 978-94-039116-3-2