CREXX
Programming Guide

The CREXX team

September 14, 2026

THE REXX LANGUAGE ASSOCIATION
CREXX Programming Series
ISBN 978-94-039116-2-5

Publication Data

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 978-94-039116-2-5

ISBN 978-94-039116-2-5

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 programming guide explains how to use the CREXX Release 1 beta line toolchain in practice.

It focuses on the tools and workflows around Level B source:

- compiling `.crexx` with `rxc` or `crexx`
- learning practical Level B syntax, modules, classes, interfaces, and tested examples through the Level B tutorial
- using initial Level G tasks, structured scopes and concurrent HTTP with complete checked examples in Concurrent programming
- assembling `.rxas` with `rxas`
- running `.rxbin` with the crexx virtual machine executables `rxvm` and `rxvme`
- linking modules with `rxlink`
- packaging native executables with `crexx -native` and `rxcpack`
- exporting versioned typed operation contracts with `crexx-contract`
- using standard libraries, classes, interfaces, plugins, compiler exits, and integration APIs

Level B is the supported beta baseline. The concurrency chapter clearly marks its Level G syntax and libraries as initial and `OPTIONS LEVELG`-gated.

The language itself is defined in the CREXX *Language Reference*. The lower level bytecode, assembly, and runtime model are described in the CREXX *VM Specification*.

1.1 Audience

This guide is for developers who want to build and run CREXX programs, integrate CREXX into tools, or understand how the command-line pieces fit together. Plugin library and compiler-exit authors should read this guide together with the VM specification and the relevant public headers.

PART I

Guide

2

Level B Tutorial

This tutorial is for experienced Rexx programmers who want to write practical crexx Level B programs in the Release 1 beta 3 documentation line. It assumes you know Rexx ideas such as `say`, `parse`, `do`, `arg`, and built-in functions, but it does not assume you have used cRexx's typed compiler, modules, classes, or interfaces before.

Level B is Rexx-family code, but it is not Classic Rexx compatibility mode. It is the implemented typed crexx language used by the toolchain, libraries, tests, and examples in this repository. When this tutorial says "Level B", it means what the current compiler accepts now, not a planned Level C or Level G feature.

For exact syntax details, keep these reference pages close:

- Data types
- Statements
- Binary memory
- Classes and interfaces
- Namespaces
- Toolchain overview

For more implemented examples in a source download, start with these paths:

- `examples/hello.crexx`,
- `examples/ooBank.crexx`,
- `lib/rxfnsb/rexx/stem.crexx`,

- `compiler/tests/rexx_src/interface_showcase_same_module.crexx`, and
- `compiler/tests/rexx_src/reference_source_iterator.crexx`.

2.1 Why Level B Still Feels Like Rexx

The first pleasant surprise is that much of the surface still reads like Rexx:

- `say` prints a value.
- `arg` receives command-line or procedure arguments.
- `do`, `if`, `select`, `leave`, `iterate`, and `return` shape control flow.
- `parse`, `address`, `signal`, and `trace` are present in the current beta surface.
- The `rxfnsb` library provides many familiar built-in functions.

The biggest difference is that Level B gives the compiler real information: values have types, modules have namespaces, procedures declare arguments and return types, and object contracts are checked.

That trade is worth leaning into. Do not write vague Classic Rexx and hope the compiler guesses your intent. Write a small amount of explicit Level B so the compiler and VM can help you.

2.2 First Program and Workflow

The usual source extension for new crexx code is `.crexx`. For reusable code, start with `options levelb` and import the Level B standard library when you use it.

```
options levelb
import rxfnsb

say "Hello Level B"
say "length=" || length("Rexx")
```

From the repository root:

```
crexx hello.crexx
```

Expected output:

```
Hello Level B
length=4
```

The `crexx` driver wraps the normal compile, assemble, and run path. The underlying stages are still visible:

1. `rxc` compiles source to `.rxas`.
2. `rxas` assembles `.rxas` to `.rxbin`.
3. `rxvm` or `rxvme` runs `.rxbin`.
4. `rxlink` can combine modules into a linked image.

The commands in this tutorial assume CREXX has been installed and `crexx`, `rxc`, `rxas`, and `rxvm` are on your `PATH`. In a source download before installation, use the matching binaries from your build directory instead. For day-to-day scripts, use `crexx`. For multi-module class/interface examples where provider modules must be present at runtime, compile the modules and run or link the resulting `.rxbin` files explicitly. The mini-project below shows that shape.

2.3 Source File Conventions

Use this header shape for committed Level B code. This is a header fragment, not a complete program:

```
options levelb
namespace myapp expose public_symbol
import rxfnsb
```

Keep `options`, `namespace`, and `import` in the leading header block. The compiler pre-scans that header before full parsing, and the rest of the repo uses this layout consistently.

The pieces mean:

- `options levelb`: compile this file as Level B.
- `namespace myapp expose public_symbol`: publish selected symbols from this module.
- `import rxfnsb`: make another namespace visible to this source file.

Headerless top-level scripts run through the `crexx` driver get practical defaults, including Level B and `rxfnsb`, but tutorial and library code should be explicit.

Command-line arguments arrive through `arg`:

```
options levelb

arg words = .string[]

if words[0] = 0 then do
  say "no arguments"
  return
end

say "count=" || words[0]
do i = 1 to words[0]
  say i || ":" || words[i]
end
```

Run with arguments by putting `-args` last:

```
crexx args.crexx -args alpha beta
```

Expected output:

```
count=2
1:alpha
2:beta
```

2.4 Types You Will Use Immediately

Level B values have types. The most common built-in source spellings are:

Type	Use
`.string`	UTF-8 text
`.int`	integer values
`.float`	binary floating-point values
`.decimal`	decimal numeric values
`.boolean`	truth values
`.binary`	arbitrary bytes
`.object`	object-shaped values
`.void`	no return value

Write `.boolean` in current Level B source. Some Rexx or C habits may suggest

`.bool`, but `.boolean` is the documented Level B spelling.

A local variable can be inferred from its first assignment, or you can declare the type first and assign the value next. The latter is useful when the value will be filled later or when you want a specific target type.

```
options levelb
import rxfnsb

title = .string
count = .int
price = .float
ready = .boolean
payload = "4f4b"x as .binary
words = .string[]
people = .stem()

title = "Level B"
count = 2
price = 1.5
ready = 1

call arrayappend words, "alpha"
call arrayappend words, "beta"
words.3 = "gamma"

people.ada = "analytical"
people["grace.hopper"] = "compiler"
tail = "ada"

say title || ":" || count
say typeof(price)
say typeof(ready)
say binlength(payload)
say words[0] || ":" || words.1 || "," || words[2] || "," || words.3
say people.tail
say people["grace.hopper"]
```

Expected output:

```
```

Practical notes:

- Arrays are typed: `.string[]`, `.int[]`, `.object[]`, and so on.
- Default growable arrays are commonly used with one-based element indexes, but Level B also supports explicit bounds such as `.int[0 to 10]` and

`.int[-2 to *]`.

- `array[0]` or `array.0` reads the current high water mark.
- Elements can be read and written with bracket notation (`words[1]`) or Rexx stem-style dot notation (`words.1`).
- Do not assign to index `0`; use ordinary element assignment or helpers such as `arrayappend`, `arrayinsert`, and `arraydelete`.
- The `.stem` class is a string-to-string keyed container for classic Rexx compound-variable style data. It supports dotted tails such as `people.ada` and bracket keys such as `people["grace.hopper"]`.
- `.string` is valid UTF-8 text in normal builds. Use `.binary` for bytes.

2.5 Procedures, Arguments, Returns, and Varargs

A Level B procedure can declare its return type after `=`, and it binds call arguments with `arg`.

```
options levelb

main: procedure
  total = sum(2, 3, 5)
  x = 10
  call bump(x)

  say "total=" || total
  say "x=" || x
  return

sum: procedure = .int
  arg first = .int, ... = .int
  total = first
  do i = 1 to arg[]
    total = total + arg[i]
  end
  return total

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

Expected output:

```
total=10
x=11
```

The important habits are:

- `arg name = .type` is by value. Changes in the procedure do not update the caller's variable.
- `arg expose name = .type` is by reference. Use it only when the procedure is meant to update caller-owned state.
- A final `... = .type` accepts a variable-length tail.
- `arg[]` is the number of values in that tail, and `arg[i]` reads tail values.

`procedure expose` is different from `arg expose`: it binds module-global storage into a procedure. Prefer explicit arguments and return values until you really need shared module state.

2.6 Expressions, Assignment, Casts, and Comparisons

Level B keeps Rexx operators, but type checking happens before bytecode is emitted.

Use these idioms:

- Assign a variable once with the type you mean; later assignments must be compatible.
- Use `expr as .type` when you need a checked conversion or object cast.
- Use `expr is .type` for object/interface tests.
- Use `typeof(expr)` for diagnostics and runtime type introspection.
- Use = for Rexx-style equality and == when you mean exact string comparison after string promotion.

For binary data, be explicit. This is a fragment:

```
payload = "4f4b"x as .binary
```

For objects, cast at the boundary. This is a fragment:

```
generic = selected as .object
restored = generic as .asset
```

If a cast cannot succeed, Level B raises a conversion signal instead of silently changing the meaning of the value.

2.7 Control Flow

The usual Rexx control-flow tools are present. `select` has both Classic Rexx condition style and a switch-like expression style. `leave` and `iterate` work on loops. Expression-form `do ... end` can return a value with `leave with`.

```
options levelb

arg words = .string[]

if words[0] = 0 then words[1] = "red"

kept = 0
do i = 1 to words[0]
  select
    when words[i] = "skip" then iterate
    when words[i] = "stop" then leave
    otherwise say "word=" || words[i]
  end
  kept = kept + 1
end

summary = do
  if kept > 1 then leave with "many"
  leave with "few"
end

say "summary=" || summary
```

Run:

```
crexx control_flow.crexx -args alpha skip beta stop gamma
```

Expected output:

```
word=alpha
word=beta
summary=many
```

2.8 Rexx-Flavoured Features

Level B keeps several Rexx features that make the language feel direct:

- `say` prints strings.
- `parse` is implemented through the certified `PARSE` exit.
- `address` sends commands to an external environment and can capture output.
- `signal` raises and handles condition objects.
- `trace` enables VM-backed tracing for debugging.

This example keeps the output small:

```
options levelb
import rxfnsb

main: procedure
  parse value "Ada Lovelace,1815" with first last "," year
  say last || ":" || year

  out = .string[]
  err = .string[]
  address crexx "echo 42" output out error err
  say "address=" || out[1]

  handled = ""
  do
    signal other "from block"
  on signal other as problem
    handled = problem.name() || ":" || problem.message()
  end
  say handled
  return
```

Expected output:

```
Lovelace:1815
address=42
OTHER:from block
```

Trace is useful but intentionally noisy. These are reference-only forms, not a runnable tutorial example:

```
trace results
```

```
trace asm
trace llm to file "trace.jsonl"
trace unsuppress namespace rxfnsb
trace off
```

Do not add `TRACE`, `PARSE`, or `ADDRESS` directly to `lib/rxfnsb/rexx/` library sources while debugging the library build. Those files are built with compiler exits disabled. Write a small caller program instead.

2.9 Modules and Libraries

Each source file is a module. Public module identity comes from `namespace`, not from the filename alone. A module exposes selected symbols; another module imports the namespace and calls those symbols.

Library module:

```
options levelb
namespace greetings expose salutation count_items

salutation: procedure = .string
  arg name = .string
  return "Hello, " || name

count_items: procedure = .int
  arg items = .string[]
  return items[0]
```

Caller:

```
options levelb
import greetings

names = .string[]
names[1] = "Ada"
names[2] = "Grace"

say greetings..salutation(names[1])
say "names=" || greetings..count_items(names)
```

Run the caller with the directory containing `greetings.crexx` on the source import path:

```
crexx modules_main.crexx -s/path/to/examples
```

Expected output:

```
Hello, Ada
names=2
```

Use `namespace..symbol` for qualified calls. The older `namespace::symbol` spelling is accepted for compatibility, but new examples should prefer the double-dot form.

2.10 Classes

Classes hold attributes and methods. A class factory is written as `*: factory` or `name: factory`; do not write a return type on a factory. In a class factory, bare `return` returns the object being constructed. This fragment is from the complete interface example in the next section:

```
fileasset: class implements .asset
  _name = .string

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

  kind: method = .string
    return "file"

  name: method = .string
    return _name
```

Ordinary application classes should let the compiler allocate attribute storage. The `with register.N` form exists for low-level VM/native interop, not for day-to-day business objects.

2.11 Interfaces

Interfaces define contracts. Classes implement them. Interface methods can be abstract, or they can provide a default/final body. In current Level B, an interface method with a body is final: a class can rely on it, but cannot override it.

Here is a complete same-file interface and provider example:

```
options levelb
```

```
namespace tutorial_objects

main: procedure
  selected = .asset("log.txt")
  fallback = .asset("memo")

  say selected.describe()
  say fallback.describe()

  if selected is .fileasset then say "file selected"

  generic = .object
  generic = selected as .object
  restored = generic as .asset
  say typeof(restored)
  return

asset: interface
  *: factory
  arg name = .string

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

  kind: method = .string
  name: method = .string

fileasset: class implements .asset
  _name = .string

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

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

  kind: method = .string
    return "file"

  name: method = .string
    return _name

cacheasset: class implements .asset
  _name = .string
```

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

kind: method = .string
  return "cache"

name: method = .string
  return _name
```

Expected output:

```
file:log.txt
cache:memo
file selected
.tutorial_objects..fileasset
```

The `asset` interface owns the public factory. `fileasset` and `cacheasset` are providers. The `*: match` method is a class-side selector. Positive scores accept the provider, zero or negative scores reject it, and the highest score wins.

2.12 Object Use

Use object values through factories, methods, interfaces, type tests, and checked casts:

- `.asset("log.txt")` calls an interface factory.
- `.fileasset("log.txt")` calls a concrete class factory.
- `value.method()` invokes a method.
- `value is .asset` tests whether the concrete object implements an interface.
- `value as .asset` casts after a runtime check.
- `typeof(value)` reports the concrete runtime type.

Level B does not implicitly call a `toString()` method for arbitrary objects. If you want text, expose a method such as `format`, `describe`, or `name`, then call it explicitly.

Level B also does not currently implement interface inheritance, interface attributes, overloads, singleton declarations, or destructor/finalizer syntax.

2.13 Collections and Iteration

Current Level B collection patterns are explicit:

- Use typed arrays such as `.string[]`, `.int[]`, and `.object[]`.
- Default growable arrays are usually shown with one-based element indexes, but array bounds can be explicit.
- Read `array[0]` for the current count.
- Use bracket notation (`items[1]`) or stem-style dot notation (`items.1`) for array elements.
- Use `arrayappend`, `arrayinsert`, `arraydelete`, and related `rxfnsb` helpers for mutating array shape.
- Use `objectarrayappend`, `objectarrayinsert`, and related helpers for `.object[]` when you need object-array mutation helpers.
- Store concrete objects in `.object[]` with an explicit `as .object` upcast, then cast back to the expected interface or class at the read boundary.
- Use `.stem` when the natural model is a REXX compound variable or keyed string map rather than an ordered typed array.

The mini-project below uses this pattern. This is a fragment:

```
items = .object[]
items[1] = .ledger..entry("coffee", -3) as .object

item = items[1] as .ledger..entry
say item.format()
```

For iterator-like classes, current Level B uses explicit references when the iterator should see live container state. Use snapshots when the iterator should see a stable copy.

2.14 References

References are explicit weak aliases to storage. They do not keep a target alive. If the target storage goes away, `refvalid(ref)` returns false and using the reference raises `REFERENCE_INVALID`.

Use the four source forms deliberately:

- `reference target`: create a weak reference to aliasable storage.
- `local = dereference ref`: link a current-scope local to the target.
- `snapshot ref`: make a deep copy of the current target.
- `<refvalid>(ref)`: test whether the target is invalid.

This example contrasts a live reference with a snapshot:

```
options levelb
import rxfnsb
namespace tutorial_references

main: procedure
  list = .Bag()
  call list.add("red")
  call list.add("blue")

  live = list.liveCounter()
  snap = list.snapshotCounter()

  call list.add("green")

  say "live=" || live.count()
  say "snapshot=" || snap.count()
  return

Bag: class
  values = .string[]
  item_count = .int

  *: factory
    initial = .string[]
    values = initial
    item_count = 0
    return

  add: method = .void
    arg value = .string
    call arrayappend values, value
    item_count = item_count + 1
    return

  size: method = .int
    return item_count

  liveCounter: method = .LiveCounter
    return .LiveCounter(reference self)

  snapshotCounter: method = .SnapshotCounter
    return .SnapshotCounter(reference self)
```

```
LiveCounter: class
  bag_ref = reference .Bag

  *: factory
    arg source = reference .Bag
    bag_ref = source
    return

  count: method = .int
    if <refvalid>(bag_ref) = 0 then return -1
    bag = dereference bag_ref
    return bag.size()

SnapshotCounter: class
  bag_copy = .Bag

  *: factory
    arg source = reference .Bag
    bag_copy = snapshot source
    return

  count: method = .int
    return bag_copy.size()
```

Expected output:

```
live=3
snapshot=2
```

Keep reference boundaries visible. Level B does not let you write convenience forms such as `list_ref.add(...)` or `items_ref[i]` directly through the reference.

2.15 Mini-Project: A Typed Ledger Module

This mini-project uses two source files: a reusable module with an interface and class, and a small application that imports it.

`ledger.crexx`:

```
options levelb
namespace ledger expose entry ledger_total

entry: interface
  *: factory
```

```
  arg label = .string, amount = .int

  label: method = .string
  amount: method = .int

  format: method = .string
    return label() || "=" || amount()

ledgerentry: class implements .entry
  _label = .string
  _amount = .int

  *: factory
    arg label = .string, amount = .int
    _label = label
    _amount = amount
    return

  label: method = .string
    return _label

  amount: method = .int
    return _amount

ledger_total: procedure = .int
  arg entries = .object[]
  total = 0
  do i = 1 to entries[0]
    item = entries[i] as .entry
    total = total + item.amount()
  end
  return total
```

ledger_app.crexx:

```
options levelb
import ledger

items = .object[]
items[1] = .ledger..entry("coffee", -3) as .object
items[2] = .ledger..entry("book", -12) as .object
items[3] = .ledger..entry("gift", 20) as .object

do i = 1 to items[0]
  item = items[i] as .ledger..entry
  say item.format()
end

say "total=" || ledger..ledger_total(items)
```

Compile both modules, then run with the standard library, provider module, and application module loaded:

```
CREXX_BIN=$(dirname "$(command -v crexx)")
rxc -i "$CREXX_BIN" -o main ledger_app.crexx
rxas -o main.rxbin main
rxc -i "$CREXX_BIN" -o ledger ledger.crexx
rxas -o ledger.rxbin ledger
rxvm "$CREXX_BIN/library.rxbin" ledger.rxbin main.rxbin
```

Expected output:

```
coffee=-3
book=-12
gift=20
total=5
```

The explicit `rxvm` command matters here because interface factory providers are discovered from the modules present in the runtime image. For deployable programs, `rxlink` is the usual way to combine those modules into one image.

2.16 Migration Guide For Classic Rexx Habits

Classic Rexx habit: Rely on untyped variables.

Level B habit: Let obvious literals infer types, or declare variables with `.string`, `.int`, `.float`, `.boolean`, `.binary`, arrays, or object contracts.

Classic Rexx habit: Assume built-in functions are always visible.

Level B habit: `import rxfnsb` in reusable source, unless you are deliberately writing a headerless `crexx` driver script.

Classic Rexx habit: Use stems as flexible bags of values.

Level B habit: Use typed arrays and array helper functions for ordered typed data. Use `.stem` for classic compound-variable style keyed strings. Treat `array[0]` as a count, not as a writable stem slot.

Classic Rexx habit: Share state freely through globals.

Level B habit: Prefer arguments and returns. Use `namespace ... expose`, `procedure expose`, or `arg expose` only for intentional sharing.

Classic Rexx habit: Treat text and bytes as the same thing.

Level B habit: Use `.string` for valid UTF-8 text and `.binary` for arbitrary bytes. Convert explicitly.

Classic Rexx habit: Expect dynamic object behavior.

Level B habit: Define an interface, implement it with classes, and use checked casts and type tests at boundaries.

2.17 Known Beta 3 Boundaries

Level B is the main implemented Release 1 beta language, but this is still a beta line. Current boundaries that matter to tutorial code:

- Level B is not Classic Rexx compatibility mode. Level C is where Classic compatibility work belongs, and normal Level C compilation is not a beta 3 contract unless the beta 3 release notes explicitly say otherwise.
- Level G has real early library work, including `rxfnsg`, but it is not the baseline user language for this release line.
- Interface default methods are final.
- Interface inheritance, interface attributes/state, overloads, singleton declarations, external-object adoption syntax, and destructor/finalizer syntax are not current Level B features.
- Objects do not implicitly convert to strings.
- References are explicit and weak. They are powerful, but not a general replacement for ordinary arguments, returns, or object methods.
- Multi-module interface-provider programs need all provider modules loaded or linked at runtime.

3

Concurrent programming

CREXX concurrency is designed to feel like Rexx: declare work much like a procedure, call it with ordinary call syntax, and introduce explicit control only when you need it. The runtime supplies bounded worker threads or isolated worker processes without exposing thread IDs, locks or shared writable VM objects.

The current surface is initial on `develop` and requires `OPTIONS LEVELG` for task declarations, task targets and `DO PARALLEL`. The underlying pool, scope, channel and value classes remain available to Level B programs. Portable publication is tracked separately from implementation status.

3.1 Five ideas to learn first

1. A **task** is one independently schedulable invocation, not an OS thread.
2. A **pool** supplies a bounded amount of execution capacity. A pool is an object created by `.taskpool.local(...)` or `.taskpool.process(...)`.
3. A **scope** owns the lifetime of every task submitted through it. Leaving a scope accounts for every child; ordinary tasks cannot detach.
4. Values cross an execution boundary by value. Each receiver gets its own reconstructed data or object; ordinary mutable REXX objects are not shared.
5. `DO PARALLEL` does not send every clause to a worker. Ordinary clauses stay on the controlling execution. Only task calls and explicit submissions create task work.

That last rule makes a parallel block predictable. File I/O, logging, result combination and other ordinary operations keep their usual ordering unless you put them in a task deliberately.

3.2 Your first task calls

A task procedure replaces `PROCEDURE` with `TASK`:

```
options levelg comments_dash
namespace first_concurrency

first: task = .int
  return 20

second: task = .int
  return 22

main: procedure = .int
  answer = first() + second()
  say answer
  return 0
```

`first()` and `second()` still look like Rexx function calls. Because both are independent task calls in one expression, crexx submits them in source order, allows their bodies to overlap, waits for both results, and then performs the ordinary `+`. It has not made addition asynchronous and has not changed the meaning of a normal function call.

This is the simple surface for everyday use. A call such as `task1() + task2()` can exploit concurrency without requiring the programmer to create future objects or write an explicit join.

Argument and receiver expressions are still evaluated left to right. Short circuiting is also unchanged: a task in the unselected side of `&` or `|` is not submitted.

3.3 Grouping work with `DO PARALLEL`

Use `DO PARALLEL` when several statements should share one structured scope:

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

```
plusone: task = .int
  arg value = .int
  return value + 1

ordinary: procedure = .int
  return 1

main: procedure = .int
  do parallel
    left = twice(20)
    right = plusone(1)

    controller_value = ordinary()

    if left <> 40 then return 1
    if controller_value <> 1 then return 2
  end

  if right <> 2 then return 3
  return 0
```

The task assignments create pending typed bindings. The tasks may run while the controller calls `ordinary()`. Reading `left` materializes that result and waits if necessary. Even though `right` is not read inside the block, `END` still accounts for it before control continues.

A pending binding is deliberately not a public future object. Before it is materialized you cannot reassign it, take a reference to it, mutate through it or return it from the block. These restrictions keep scope cleanup reliable.

Void tasks work naturally with `CALL`:

```
announce: task
  arg text = .string
  say text
  return

do parallel
  call announce "worker one"
  call announce "worker two"
end
```

The two calls share the block scope and both complete by `END`. Their output order is not defined. Outside `DO PARALLEL`, `CALL announce ...` owns an implicit statement scope and completes before the next statement.

The complete runnable version is `examples/concurrency_basic.crexx`.

3.4 Returning a value from a parallel block

The existing expression-form `DO` also supports `PARALLEL`:

```
answer = do parallel
  left = twice(20)
  right = plusone(1)
  leave with left + right
end
```

`LEAVE WITH` supplies the block expression's result. The task results are materialized before addition and the scope closes before `answer` is assigned.

3.5 Pools are policy; scopes are lifetime

The simple forms use an execution-local default pool. Create a pool explicitly when worker count, admission capacity, failure policy, deadline or isolation is part of your program's design.

```
square: task = .int
  arg value = .int
  return value * value

increment: task = .int
  arg value = .int
  return value + 1

run_pair: procedure = .int
  arg pool = .taskpool

  scope = .taskscope.collectall(pool, 60000)
  answer = do parallel using scope
    squared = square(6)
    next = increment(5)
    leave with squared + next
  end
  return answer

main: procedure = .int
  local_pool = .taskpool.local(2, 8)
  local_answer = run_pair(local_pool)
  call local_pool.close()

  process_pool = .taskpool.process(1, 8)
  process_answer = run_pair(process_pool)
  call process_pool.close()
```

```
if local_answer <> 42 | process_answer <> 42 then return 1
return 0
```

The two numbers passed to a pool factory are worker capacity and bounded admission capacity. A local pool runs fresh Rexx executions in the same runtime. A process pool runs fresh executions in isolated worker processes. Changing provider does not change task syntax.

`DO PARALLEL USING scope` consumes and closes that scope at `END`, including abnormal exits, but it does not close the pool. A scope cannot be reused. Close the pool only after every scope created from it has closed.

Capacity controls possible overlap; it does not promise a schedule. The process pool above deliberately has capacity one, so the two task bodies run serially while retaining exactly the same structured result semantics.

The checked source is `examples/concurrency_providers.crexx`.

3.6 Choosing fail-fast or collect-all

`.taskscope.failfast(pool, milliseconds)` requests cancellation of siblings after the first unsuccessful child. `.taskscope.collectall(...)` continues to account for the remaining children. Both policies join every accepted child.

The timeout is a relative scope deadline in milliseconds:

- `-1` means no deadline or indefinite observation;
- `0` means immediate/nonblocking observation; and
- a positive value is a finite relative wait or deadline.

Cancellation is cooperative for a task already running in a local worker. An isolated process provider can terminate and replace its worker if cooperative cancellation does not finish in its grace period. Therefore task code should check its `.taskcontext` at sensible boundaries when using the advanced `.taskwork` interface.

3.7 Values do not become shared objects

Primitive arguments and results cross directly as canonical values. A concrete class can cross when it declares an exact pair of conversion members:

```
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
```

The same rule applies to the receiver of a task method:

```
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())
```

Now ordinary method syntax performs a task invocation:

```
calculator = .calculator(40)
pool = .taskpool.local(2, 8)
```

```
scope = .taskscope.collectall(pool, 60000)

answer = do parallel using scope
  leave with calculator.add(.numberbox(2))
end

call pool.close()
say answer.value()
```

The controller encodes the receiver and argument. The worker reconstructs its own objects, performs the method, encodes the result, and the controller reconstructs a new `.numberbox`. Neither side shares live object identity.

The full checked example is `examples/concurrency_transferable_objects.crexx`.

3.8 Advanced work classes and factories

Use `.taskwork` when you want an explicit target, a canonical application request and direct completion inspection. It is the low-level runnable interface beneath higher-level typed task calls.

```
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)
```

The factory expression becomes a sealed task target:

```
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)

if completion.succeeded() then answer = completion.value().as_integer()
else say completion.error_code() completion.message()
```

```
call scope.finish()
call pool.close()
```

The pool does not create the `.adderwork` object in the controller and then share it. The target records the statically known factory; its arguments cross as values and the receiver object is constructed in the worker.

`target.name()` is diagnostic information. Dispatch uses a linker-sealed callable descriptor, not a procedure-name string. Users should create targets with `task callable` or `task .class(...)`, not by constructing the underlying 80-byte binding.

The full local/process example is `examples/concurrency_taskwork.crexx`.

3.9 Observing tasks and completions

The Level B path exposes these useful operations:

- `child.wait(milliseconds)` observes that particular task;
- `child.completion()` is a nonblocking observation;
- `scope.next(milliseconds)` observes the next child in completion order; and
- `scope.join()` returns every child in stable submission order.

A timed or nonblocking miss is a `.completion` with `available() = 0`, not a failed child. A terminal completion can be `SUCCEEDED`, `FAILED`, `CANCELLED`, `DEADLINE_EXCEEDED`, `REJECTED`, `ENDPOINT_CLOSED`, `TRANSPORT_LOST` or `UNKNOWN_OUTCOME`.

`scope.finish()` joins, closes and raises `TASK_FAILURE` if a child did not succeed. Use explicit `wait()`/`join()` and inspect completion data when your application wants to handle each outcome itself. `scope.abort(reason)` is the safe cancellation, join and close path during controller-side cleanup.

3.10 Recursion and nested task calls

A direct self-call inside a task is ordinary synchronous recursion in that worker:

```
factorial: task = .int
  arg value = .int
  if value <= 1 then return 1
  return value * factorial(value - 1)
```

The recursive calls are not resubmitted. Calling a different task from a task body would need a blocking nested wait and is rejected in the current surface. Move the dependency to the controller, where CREXX can represent and join it without consuming a worker that waits for its own bounded pool.

3.11 HTTP uses the same task model

The Level G HTTP client declares `get`, `post`, `request` and `post_stream` as task methods, so ordinary calls and `DO PARALLEL` use exactly the rules in this chapter. The HTTP server uses the explicit form: its controller submits a sealed `task .my_handler()` factory target for each complete request. A handler implements `.httpservice .taskwork`; it is stateless task work, not the future durable service represented by `.serviceref` and `.taskscope.ask()`.

The HTTP client and server guide contains a complete handler class, parallel client calls, lifecycle rules and links to both-VM executable fixtures.

3.12 What not to design around yet

The following names or concepts are reserved rather than usable today:

- `.taskscope.ask()` and concrete single-owner services;
- `.taskpool.queued()` and `.taskpool.running()` telemetry;
- cross-host provider type `3` and a public provider-plugin ABI;
- detached tasks, worker affinity and public thread IDs; and
- shared mutable globals, locks or atomics between REXX executions.

Inside `.taskwork.run()`, `.taskcontext.endpoint(reference)` reconstructs a worker-local byte endpoint from a transferable type-4 provider reference. Controller-side code can use `.byteendpoint.from_reference()` directly.

3.13 Checked example catalogue

These examples are compiled with `rxc`, assembled with `rxas`, linked with `rxlink`, and executed on `rxbvm` and `rxtvm` in optimized and unoptimized modes:

Example	Demonstrates
`concurrency_basic.crexx`	ordinary task calls, `DO PARALLEL`, controller clauses and `CALL` tasks
`concurrency_providers.crexx`	explicit pool/scope policy and local/process selection
`concurrency_transferable_objects.crexx`	task method receiver, object argument and object result transfer
`concurrency_taskwork.crexx`	`.taskwork` factories, explicit submission and completion inspection

For formal syntax and compile-time restrictions, see the language reference's Concurrency chapter. For every Level B class and method, see the library reference's concurrency chapter.

4

Packed Binary Memory

Packed binary memory is for code that needs to treat a `.binary` value as a byte-addressed record space. Typical examples are sorted lookup tables, parser keyword tables, indexes, protocol records, and compact tree nodes.

Use this feature when a packed layout avoids many ordinary Rexx variables, temporary substrings, or repeated binary slice copies. Do not use it just to make simple code look lower-level; normal variables are clearer when the data is not layout-sensitive.

The exact language rules are in the Language Reference chapter Binary Memory.

4.1 Choosing Intrinsics Or Helpers

Use binary-memory intrinsics for hot direct access:

```
hash = <at..u32>(node + NODE_HASH) arena
if <compare..binary>(arena, key_offset, wanted_key) = 0 then say "hit"
```

Use ordinary helper functions for buffer management and mutation:

```
call binresize arena, NODE_SIZE * 128
call binmemmove arena, dst_offset, src_offset, count
call bincopy target, 0, source, source_offset, length
```

The compiler may direct-lower selected helpers, but the source still reads as an ordinary action. That is deliberate: moving, resizing, and opening gaps are operations, not field references.

4.2 Performance Shape

Direct binary-memory field access is designed for hot packed-data loops. The Release 1 VM keeps the strict bounds checks but uses direct fixed-width load/store helpers for common scalar widths. In microbenchmarks, repeated `<at..u8>`, `<at..u32>`, and `<at..int>` access is faster than indexed attribute-array access when the loop is mostly scalar field reads or writes.

That does not mean every data structure should be packed. A binary layout wins when it removes copying, stores dense records, or lets the program scan bytes without materializing substrings. A normal Rexx array or set of registers can still win when the algorithm naturally works with scalar values and the packed layout forces repeated manual offset arithmetic.

Good binary-memory candidates:

- byte scanners and generated lexers;
- fixed-width token or index tables;
- lookup records where key bytes can be compared without slicing;
- large binary constants used directly with `<at>` and `<compare>`;
- compact external record formats whose layout is already byte-based.

Weaker candidates:

- small scalar metadata that is read one field at a time;
- code that needs many separate offset calculations for every logical action;
- ordinary string processing where Unicode-aware `.string` operations dominate;
- structures where a register or `.int[]` representation already avoids copies.

Keep offsets in local variables inside hot loops. For repeated access to fields in one record, compute the row or node base once, then add field offsets:

```
row = HEADER_ROWS + i * ROW_SIZE
hash = <at..u32>(row + ROW_HASH) table
kind = <at..u8>(row + ROW_KIND) table
```

Use smaller fixed-width storage only when it matches the format. `.u8` and `.u32` reduce the binary footprint and can help dense structures, but they also add range semantics. Use `.int` when the field is genuinely a Rexx integer and the extra bytes do not matter.

4.3 Layout Constants

Write layout constants in bytes. This makes offset arithmetic visible and keeps the source independent of the storage type used for each field.

```
options levelb
namespace packedindex expose find_node

packedindex_layout: procedure expose NODE_LEFT NODE_RIGHT NODE_HASH
    NODE_KEY NODE_SIZE
  constant NODE_LEFT = 0
  constant NODE_RIGHT = NODE_LEFT + <sizeof..u32>
  constant NODE_HASH = NODE_RIGHT + <sizeof..u32>
  constant NODE_KEY_LEN = NODE_HASH + <sizeof..u32>
  constant NODE_KEY = NODE_KEY_LEN + <sizeof..u16>
  constant NODE_SIZE = 32
  return
```

For reusable modules, expose public procedures. Keep Release 1 layout constants inside the source module and declare them in an explicit procedure scope. Top-level executable code in a library-shaped file can synthesize an implicit `main()`, which is not usually what a packed-layout module wants.

For script-style examples that need a separate layout-constant declaration procedure, add an explicit `main: procedure`. Otherwise the statements after the declaration procedure are part of that procedure body until the next callable boundary.

4.4 Header Check

Binary constants and binary variables use the same direct-access syntax for reads and compares.

```
options levelb

check_header: procedure = .int
  arg page = .binary
  constant MAGIC = "4352584944583031"x as .binary  /* CRXIDX01 */

  if <compare..binary>(page, 0, MAGIC) = 0 then return 1
  return 0
```

Use = only when comparing already-materialized values:

```
magic_copy = <at..binary>(0, <blen>(MAGIC)) page
if magic_copy = MAGIC then say "copied compare"
```

For lookup loops, prefer `<compare..binary>` so the compiler can emit a direct binary compare.

4.5 Sorted Table Lookup

This example shows the intended pattern for a packed sorted table. The table header stores the row count. Each row has a hash and a variable-size key area. The example omits the full binary-search loop to focus on field access.

```
options levelb

find_row: procedure = .int
  arg table = .binary, wanted_hash = .int, wanted_key = .binary

  constant HEADER_COUNT = 0
  constant HEADER_ROWS = 8
  constant ROW_HASH = 0
  constant ROW_KEY_LEN = 4
  constant ROW_KEY = 6
  constant ROW_SIZE = 40

  count = <at..u32>(HEADER_COUNT) table
  do i = 0 to count - 1
    row = HEADER_ROWS + i * ROW_SIZE
    hash = <at..u32>(row + ROW_HASH) table
    if hash = wanted_hash then do
      key_len = <at..u16>(row + ROW_KEY_LEN) table
      if key_len = <blen>(wanted_key) then
        if <compare..binary>(table, row + ROW_KEY, wanted_key) = 0 then
          return row
    end
  end

  return -1
```

The hot loop reads fixed-width fields directly, checks the stored key length, and compares the key without making a binary slice. The compiler test fixture for this shape inspects emitted RXAS and fails if `bslice`, string extraction, or helper calls appear in the lookup path.

4.6 Insert And Delete Paths

Insertion and deletion usually need memory movement. Keep that as helper-style source. It may still be direct-lowered, but it does not need an angle-bracket intrinsic unless profiling proves the helper spelling is a problem.

```
options levelb

insert_gap: procedure = .void
  arg expose table = .binary, row = .int, count = .int
  constant ROW_SIZE = 40

  old_len = <blen>(table)
  new_len = old_len + ROW_SIZE
  call binresize table, new_len

  move_from = row
  move_to = row + ROW_SIZE
  move_len = old_len - row
  if move_len > 0 then call binmemmove table, move_to, move_from,
      move_len

  call binfillat table, row, ROW_SIZE, 0
```

`binfillat(table, offset, length, byte)` is the Release 1 span-fill helper. It currently uses a small checked helper loop; whole-buffer `binfill(table, byte)` is backed by the RXAS `bfill` instruction. `binmemmove` is safe for overlapping ranges and currently uses conservative library code; direct lowering remains a future optimization if profiling shows helper overhead in hot mutation paths.

4.7 Text And Decimal Fields

There are two text layouts, but new packed structures should prefer zero-terminated UTF-8 fields:

- zero-terminated UTF-8 fields, read with `<at..string>(offset)` and compared
- fixed-codepoint UTF-8 source spans, read with `<at..string>(offset, codepoints)` only when the program already knows the source span in codepoints.

For zero-terminated fields, the program must know that the binary format stores a zero byte after the text. Omit the length to read the field, or use `<compare..string>`

to compare it without allocating a string:

```
keyword = <at..string>(keyword_offset) record
if <compare..string>(record, keyword_offset, "select") = 0 then say "
    keyword"
```

Invalid UTF-8 signals `UNICODE_ERROR`.

Decimal fields use zero-terminated decimal text. This keeps the packed format simple for variable-width decimal values; the program only needs the starting byte offset:

```
amount = <at..decimal>(amount_offset) record
<at..decimal>(amount_offset) record = amount + 1d
```

The read form scans to the zero byte, validates the selected text, and parses it through the active decimal plugin. The write form converts the decimal value to decimal text, writes the bytes, and writes one zero byte. The binary buffer must already have room for the complete text plus terminator.

4.8 When Not To Pack

Prefer ordinary Rexx values when:

- the data is small and not in a lookup hot path;
- field layout is not externally visible;
- text processing is mostly Unicode-aware string work;
- the code would need many manual offsets but little measurable speedup;
- a class or array would express the model more safely.

Packed binary memory is a performance tool. Use it when it removes real copying or lets a data structure stay cache-friendly.

5

Overview of the Toolchain

CREXX is built as a small set of separate tools with visible boundaries. That is deliberate: users can inspect the generated assembly, assemble it directly, link modules, disassemble bytecode, or package a program for native deployment.

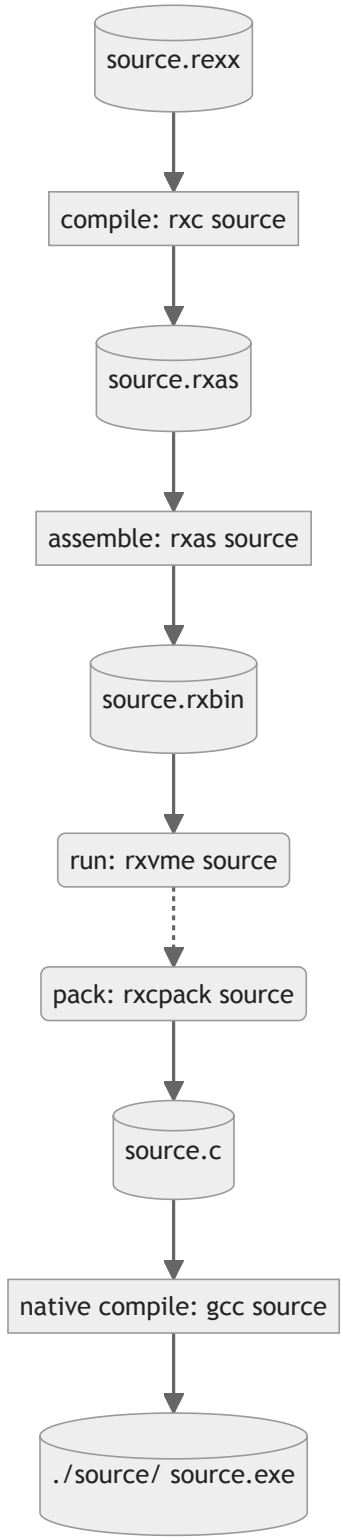
The common flow is:

1. `rxc` compiles CREXX source to `.rxas` assembly.
2. `rxas` assembles `.rxas` assembly to `.rxbin` bytecode.
3. `rxvm` or `rxvme` executes one or more `.rxbin` files using the compiler-selected product dispatch engine. `rxbvm` selects switch dispatch explicitly; `rxtvm` selects direct threading where supported.
4. `rxlink` optionally combines modules into a linked image with a shared constant pool.
5. `rxcpack` can package a linked image as C data for native executable builds.

The `crexx` driver (see the crexx tool on page 75) wraps the usual compile, assemble, run, link, and package steps for day-to-day use. For a program made from several sources, `crexx main.crexx support.crexx` compiles the listed files and runs them together. Imports do not add further source files to that build.

For larger projects and repeated development, `crexx --program output sources...` and `crexx --library output sources...` provide incremental builds and named linked outputs. A build system like Ninja or CMake[1] is appropriate when the project also owns native code, generators, installation or broader QA.

[1] The native executables in the CREXX distribution are built with CMake for all Operating System / Instruction Set Archi-



tecture combinations.

5.1 Source, Assembly, and Bytecode

Source files are UTF-8 text. CREXX source normally uses `.crexx`, with `.crx` as a short alias. `.rexx` remains the compatibility/classic extension. The compiler also accepts an arbitrary non-reserved initial source extension for that compile. Toolchain artifacts are less flexible: *rxas* assembly uses the fixed `.rxas` suffix and *rxbin* bytecode uses the fixed `.rxbin` suffix. The compiler output is RXAS assembly:

```
rxc hello.crexx
```

The assembler output is `.rxbin` bytecode:

```
rxas hello.rxas
```

`.rxbin` is the module format consumed by the VM, linker, disassembler, and packager.

For `rxc`, `rxas`, and `rxlink`, `-o` names an output stem/path unless it already ends in the tool's fixed artifact suffix. For example, `rxas -o app` writes `app.rxbin`, `rxas -o app.rxbin` writes `app.rxbin`, and `rxas -o app.debug` writes `app.debug.rxbin`.

5.2 Module Discovery

The compiler separates source discovery from binary discovery:

- source roots contain `.crexx`, `.crx`, `.rexx`, and any arbitrary extension used by the initial source file for this compile
- binary roots contain `.rxbin`, optional `.rxas`, and `.rxplugin` files

Use `rxc -s path` to add a source import root and `rxc -i path` to add a binary import root. The `crexx` driver exposes the same distinction for compilation. The directory of the source being compiled is searched for source imports, but it is not searched for `.rxbin` imports unless it is also passed with `-i`. That avoids stale generated bytecode shadowing the source you are editing.

When a source file has no `options level...` clause, `.crexx`, `.crx`, and arbitrary initial extensions default to Level G; `.rexx` defaults to Level C.

The runtime library path is separate. Use `crexx -l...` or the VM's own library loading rules when a compiled program must load bytecode or plugins at runtime.

5.3 Linking

`rxlink` combines one or more `.rxbin` modules into a linked image with one shared constant pool. This reduces duplicate constants and resolves module and interface-provider relationships up front while preserving the module records needed by the VM loader.

The use of `rxlink` is a good choice for release-style deployable images and native packaging.

5.4 Native Packaging

Native packaging uses `rxlink` before `rxcpack`. The linked image is serialized to C data and linked with the VM runtime library by the platform C toolchain. The generated executable contains the program image and the runtime pieces selected by the build.

Plugin libraries (which are native libraries) in the CREXX distributions have static and dynamic versions. Using a static or dynamic versions of a plugin is a build choice. Although the intention is to keep distributions equal over operating systems and hardware architecures, it is not realistic to expect that every distribution contains every optional native plugin. What is delivered, however, is documented in the *Library Reference* publication.

6 Global Procedures

6.1 Exposing Global Procedures

A procedure defined in a module file can be made available to other modules (provided it is in an imported namespace) by including the procedure in the namespace instruction.

```
namespace mynamespace expose myfunc1 myfunc2
```

6.2 Importing Global Procedures

When compiling a program, the compiler looks for procedures that are called but not defined in the current module. It searches in various locations in the following order:

1. The primary source root: source files in the directory of the source being compiled, or in the working location selected with `-l` when the source name has no directory.
2. Additional source roots specified with `rxc -s`: source files.
3. Binary roots specified with `rxc -i`: `.rxbin` files, optional `.rxas` files when `--import-rxas` is enabled, and CREXX native function libraries (plugins).
4. The directory where the compiler executable (`rxc`) is located: deployed `.rxbin` files, optional `.rxas` files when `--import-rxas` is enabled, and plugins.

The source root is not automatically searched for `.rxbin` files. Pass `-i .` or `-i build-dir` when a local binary module is an intended compile-time dependency. This separation avoids stale generated `.rxbin` files beside source files being imported by accident.

Source roots search `.crexx`, `.crx`, and `.rexx`. If the initial source file uses another extension, such as `.the`, that extension is searched for this compile too. Files without an explicit `options level...` default to Level G for `.crexx` and `.crx`, and arbitrary initial extensions, and Level C for `.rexx`.

The compiler only uses external procedures that are in the same namespace as the module being compiled or for namespaces listed in the `import` instruction.

```
import rxfnsb mynamespace anothernamespace
```

Metadata encoded with the procedures allows the compiler to determine the type and argument types of imported procedures.

The CREXX level b standard library namespace is `rxfnsb`, to access these procedures programs should import this namespace.

```
import rxfnsb
```

Without the `import` statement for the `rxfnsb` namespace, the level B built-in functions cannot be found. One could say that the built-in functions are not as built-in as they are in Classic Rexx. This is different in CREXX level C, the compatibility level.

7

Global Variables

Global variables are shared among modules and are linked to the namespace of the modules, similar to exposed procedures.

Namespace auto-exposing: Variables exposed through the module `namespace` instruction are automatically mapped into the local scope of all procedures within the module file. Use this when the variable is part of the module's published namespace surface.

```
namespace myproject expose global_var
```

Procedure-level exposing: The `expose` keyword in a `procedure` instruction binds the listed names to module-global storage for that procedure only. Use this for private module state shared by selected local procedures without publishing that variable through the namespace.

```
proc1: procedure = .void expose global_var
  global_var = "Hello World"
  return

proc2: procedure = .void expose global_var
  say "global_var is" global_var
  return
```

A procedure that does not list `global_var` does not see this shared storage unless `global_var` is also in the file-level `namespace ... expose ...` list.

Physical Placement and Hoisting: To ensure robust scoping, global variables physically reside in the **Namespace Scope** of the module. During the compilation process, when a variable is identified as global (either via the `namespace expose`

list or a `procedure expose` list), the compiler physically **hoists** the variable symbol from its local discovery scope up to the Namespace Scope. This ensures that standard tiered resolution (Local -> Namespace -> Universe) correctly identifies the variable across all procedures and blocks within the module. This hoisting is strictly upward; the compiler never demotes a namespace-level variable to a local scope.

Namespace Isolation of Global Variables: Unlike exposed functions, which can be called across imported modules, **exposed global variables are strictly isolated to their originating module's namespace**. The compiler explicitly prevents cross-namespace variable resolution. To read or write a global variable from a module in a different namespace, you must wrap the variable access in an exposed function (a getter/setter) and import that function.

Typing and Shadowing of Global Variables: Global variables are typed based on their first use or declaration. All modules exposing the variable must agree on this type.

- **Top-Level Declarations:** In the top-level scope of a procedure (i.e. directly inside `procedure`), a typed declaration like `x = .int` does *not* create a local shadow variable. Instead, it asserts and binds the type to the global variable `x`.
- **Sub-Scope Declarations (Shadowing):** In sub-scopes (e.g. inside a `do...end` block or an `if ... do` block), a typed declaration like `x = .int` **creates a shadow local variable** that hides the global variable for the duration of that block.
- **Untyped Assignments:** In single-instruction branches (e.g., `if cond then x = 10`), an untyped assignment binds directly to and updates the global variable.
- Inside a `DO` block (e.g., `if cond then do; x = 10; end`), if `x` does not already exist in an outer scope, it creates a block-local variable. If it does exist (including as a global), it updates that variable.

Note: The distinction between single-instruction branches and `DO` blocks is an intentional design choice in Level B.

7.1 Declaring and clearing an exposed array

The declaration `TOK = .string[]` binds the array type to exposed global storage. It does not clear that storage on each call. A routine that rebuilds tokens for each input must clear the array explicitly:

```
Tokenize: procedure = .void expose TOK
  arg line = .string
  TOK = .string[]
  call arraydrop TOK
  do i = 1 to words(line)
    call arrayappend TOK, word(line, i)
  end
  return
```

This example requires `import rxfnsb`. Calling it with `"a b c"` and then `"x y"` leaves only `x` and `y` in `TOK`. A factory call such as `VARSET = .stem()` is different: it creates and assigns an object each time.

7.2 Level B System Symbols

In Level B programming, namespaces, procedure names, class names, and variable names beginning with an underscore "_" are private. Level B is specifically designed to offer low-level system capabilities. These variables and procedures are intended for use in low-level capabilities that support other language levels.

8

Intralanguage calls

This chapter discusses calls from one CREXX procedure to another, including the built-in function package. Search order is an intrinsically related concept: how is the called component found. There are some differences with Classic REXX and ooRexx, which can call (and interpret) external procedures in source. In this respect, CREXX behaves like NetREXX, where a called component needs to be compiled, executable code; for NetREXX a `.class` file and in CREXX an `.rxbin` file. Level B introduces a package (module) system where a program can be part of a package and be imported into calling code.

8.1 At compile time

At compile time, a program uses the `CALL` statement, or the function notation with parentheses (also called round brackets). Going forward, and moving into object oriented notations for other REXX variants, the latter is going to gain importance, while the `CALL` statement will be fixed in its current functionality. For that reason, most examples will be in the function notation.

The compiler needs the callee's *signature*[2], which it can obtain from source or compiled metadata. Reading an imported source contract does not automatically produce or load that module's executable bytecode. At runtime, any call that remains in the compiled caller needs the library to be loaded.

[2]with signature we mean the combination of parameter types and return type.

This implies that there are inherent dependencies to be followed while building an application system that consists of multiple modules; this is not different than in other compiled languages. Building utilities like Make or Ninja can provide these services, and these can be orchestrated by meta-build tools like CMake. The CREXX toolchain itself is built using CMake and from its build specification in CMake most of these patterns can be gleaned.

The `import` statement tells the compiler we want to import functions from a certain package.

8.2 Imported interfaces and classes

Level B extends the same import model to classes and interfaces. An imported namespace may expose:

- procedures
- interfaces
- classes

An interface call is normally written against the contract name:

```
import garage

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

When two imported namespaces expose the same contract 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 an imported namespace. Qualification is a disambiguation mechanism; it does not create a second way to reach symbols that have not been imported. `::` remains accepted as a compatibility alias.

8.3 Casts and type inspection

Object contracts can be checked explicitly:

```
generic = .car("roadster") as .object
vehicle = generic as .garage..vehicle
current = vehicle as .garage..car

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

`as` performs a checked object cast, `is` performs a boolean type test, and `typeof` returns the canonical runtime type name.

8.4 Source and binary imports

At compile time, `rxc` distinguishes between:

- source roots, used for CREXX source files
- binary roots, used for `.rxbin`, optional `.rxas`, and plugins

The source file being compiled contributes its own directory as the primary source root. Additional source roots may be passed with `-s`. Binary roots are passed with `-i`. The primary source directory is not automatically a binary root, so a sibling `.rxbin` is visible only when that directory is also passed with `-i`.

Source-root discovery includes `.crexx`, `.crx`, and `.rexx`. If the initial source file has another extension, for example `.the`, that extension is added to source-root discovery for that compile. The default language level is Level G for `.crexx`, `.crx`, and arbitrary initial extensions, and Level C for `.rexx`; explicit `options level...` always wins.

`.rxpp` remains the extension for existing RXPP preprocessor workflows. A future idempotent preprocessor path may reduce the need for a separate preprocessor extension, but that is not part of the source import rule.

For ordinary project work this means:

- keep project source modules in source roots
- keep packaged binary libraries in binary roots

- use namespace qualification only when imported namespaces collide

This split is deliberate: it keeps active source preferred during development and stops stale generated `.rxbin` files beside source files from silently satisfying imports.

8.5 Supplying the library at runtime

Consider these two files. `libmod.crexx` exposes a function that calls a private helper:

```
options levelb
namespace libmod expose A

A: procedure = .string
  return B("fixed")

B: procedure = .string
  arg y = .string
  return "hello from B with " || y
```

`usemod.crexx` imports that namespace:

```
options levelb
import libmod
say libmod..A()
exit 0
```

Running `crexx usemod.crexx` can compile successfully by reading `libmod.crexx`, yet fail with `FUNCTION_NOT_FOUND` for `libmod.a` because the library has not been supplied to the VM. A very small function might appear to work because optimization inlines it into the caller. That does not establish that the library was loaded, and adding a helper or disabling optimization can expose the missing dependency.

For this two-source program, the ordinary solution is to list both files:

```
crexx usemod.crexx libmod.crexx
```

The command compiles both sources and runs them together, printing `hello from B with fixed`. No `-l` or `--program` option is needed. The shorter `crexx usemod libmod` form works with these `.crexx` filenames too.

8.5.1 Separately built libraries and larger projects

As the project grows, you may choose to build reusable code separately and supply its exact runtime path:

```
rxc libmod.crexx
rxas libmod.rxas
crexx –l ./libmod.rxbin usemod.crexx
```

For repeated development, build one named linked program incrementally:

```
crexx ––program combined usemod.crexx libmod.crexx
rxvme combined.rxbin
```

These routes also print `hello from B with fixed`. `--program` builds without running; an unchanged repeat skips compilation, assembly and linking. `--library output sources...` offers the same incremental build support for a reusable linked library. Both modes retain explicit source membership. The ordinary compile-and-run command recompiles its listed sources unless `--nocompile` is selected.

The `./` in `-l ./libmod.rxbin` selects the application-local library path; a bare `-l` library name is resolved relative to the tool installation.

Packaged binary imports provide another route. When the compiler selects a `.rxbin` library through a binary root such as `-i .`, it records the exact package stem for runtime autoload. The VM can load that package from its module roots when an unresolved callable needs it. This does not search for or compile source files, and compiler `-s`/`-i` roots do not themselves add VM search roots. Merely leaving `libmod.rxbin` beside the source does not make it a compiler input: the binary must be visible through a binary import root. Use `-i .` when the current directory is an intended binary-library location.

9 Interlanguage calls

A CREXX program is able to call programs written in the REXX language, but also programs native to the platform, using a number of calling conventions:

9.1 Current Mechanisms

Address the `address` statement can use the shell and I/O indirection to start native executables and provide input, and retrieve the output.

Plugins the plugin system that is available to the CREXX environment.

Crexxsaa the host integration facade for C applications that want to embed cRexx, register ADDRESS environments, and run `.rxbin` or source files through the CREXX toolchain and runtime.

10 crexxsaa host integration

`crexxsaa` is the initial CREXX compatibility API for C hosts that want a REXXSAA-shaped entry point without taking on the full historical REXXSAA ABI. It is deliberately small: the API is a stable facade over the current CREXX `rxvml`, `ADDRESS`, sandbox, and exposed-variable contracts.

This enables the use of CREXX as an integrated part of applications or operating environments; it enables CREXX to function as a universal macro language. Supplied examples show how CREXX functions as an edit macro language[3] or to enable the use of embedded SQL[4] in a CREXX program.

The first supported host model is command-environment execution:

- host C code creates a `crexxsaa_context`
- the host registers one or more `ADDRESS` callback environments
- the host configures the CREXX compiler, assembler, import directory, and optional cache directory
- the host runs either an existing `.rxbin` or a source file that `crexxsaa` compiles and caches

This is not a full `RexxStart()` or `RexxVariablePool()` clone. Future adapter entry points may be added where real integrations need them, but the internal runtime model remains the modern CREXX `ADDRESS` model.

[3] In the THE editor, a faithful reproduction of the VM system editor by Mark Hessling.
[4] For sqlite, the universally used in-process SQL engine.

10.1 Writing hosted source

`crexxsaa` compiles source exactly as supplied. It does not add `OPTIONS LEVELB`, an `ADDRESS` clause, imports, or host-specific boilerplate.

Hosted source should therefore declare the language level and command environment it needs:

```
options levelb
address the
'emsg CREXX_PROFILE_HOSTED'
```

The script owns its language mode and its command routing. `crexxsaa` owns only compile, cache, load, and run orchestration.

10.2 Minimal host flow

```
crexxsaa_context *ctx = NULL;

crexxsaa_create(location, library_rxbin, &ctx);
crexxsaa_set_compiler(ctx, rxc_path, rxas_path, import_dir);
crexxsaa_register_address_environment(ctx, "THE", the_callback,
    userdata);
crexxsaa_set_address_environment(ctx, "THE");
crexxsaa_run_source(ctx, profile_path, "THE", 0, argc, argv, &
    program_rc);
crexxsaa_destroy(ctx);
```

`crexxsaa_run_source()` and `crexxsaa_run_rxbin()` return zero when the host API successfully compiled, loaded, and ran the program. The REXX program status is returned separately through `program_rc`: an integer `main` return value and a top-level `exit n` both set `program_rc` to `n`.

The cache namespace argument, `"THE"` in this example, is part of the cache identity. Different hosts can compile the same source path without sharing a cache bucket accidentally.

10.3 Low-level rxvml calls

Hosts that bypass the `crexxsaa` facade and call `rxvml` directly must use the descriptor invocation APIs:

- `rxvml_call_procedure_descriptor()`

- `rxvml_call_factory_descriptor()`
- `rxvml_call_method_descriptor()`

Descriptors have the form `rxsig1|name|return_type|args`, for example `rxsig1|pkg.main|.int|pat`. The runtime checks the descriptor against the callable metadata before invoking, so a name match with the wrong return or argument signature is rejected. This descriptor requirement is part of the `RXVML_ABI_VERSION` 8 break. `CREXXSAA_ABI_VERSION` is 3 so source-cache entries produced by older host/runtime pairs are not reused.

10.4 ADDRESS callbacks

A host registers an environment with `crexxsaa_register_address_environment()`. When CREXX executes a command in that environment, the callback receives a `crexxsaa_address_request` containing the environment name, command text, and active context.

The callback fills a `crexxsaa_address_response`:

- `rc`: command return code
- `condition_name`: optional CREXX condition name
- `diagnostic`: optional diagnostic text

The callback should return zero for a successfully handled dispatch. Non-zero callback return values indicate host-side failure in the bridge itself.

10.5 Variable access during callbacks

`crexxsaa` exposes simple name-based helpers for active `ADDRESS` callbacks:

- `crexxsaa_address_variable_get_alloc()`
- `crexxsaa_address_variable_set()`
- `crexxsaa_free()`

These helpers are valid only while a native `ADDRESS` callback is active. They resolve variables in this order:

1. Direct scalar `ADDRESS ... EXPOSE name` bindings.

2. Direct stem/array `ADDRESS ... EXPOSE name[]` bindings.
3. The active `ADDRESS ... SANDBOX pool` object.
4. The request's standard sandbox fallback.

The host gets one API, while the CREXX script chooses whether a command uses direct exposure or sandbox storage.

Compound names such as `FILENAME.1` are mapped to exposed stems by splitting at the first dot and using the remaining text as the stem key. Names are matched case-insensitively at this facade layer to fit legacy host expectations. The standard CREXX sandbox already normalises keys internally.

For compatibility with command processors that treat a scalar result as a one-item stem, an exposed scalar also has a limited compound view:

- `name.0` reads as `"1"`
- `name.1` reads the scalar value
- writes to `name.0` are accepted and ignored
- writes to any other `name.tail` update the scalar value

This rule is applied only when no exposed stem with the same base name exists. Real `EXPOSE name[]` stems keep their normal stem semantics.

The variable facade is not a general variable-pool enumerator and does not provide arbitrary access to unexposed CREXX locals.

10.6 Source cache

`crexxsaa_run_source()` compiles source through `rxc` and `rxas`, then stores the resulting `.rxbin` in a disposable cache. Normal source edits, CREXX rebuilds, compiler path changes, and library rebuilds cause a recompile without manual cache clearing.

Cache location rules:

- `CREXXSAA_CACHE_DIR` overrides the platform default
- a host may call `crexxsaa_set_cache_dir()`
- if both `CREXXSAA_CACHE_DIR` and a host-provided cache directory are present, `CREXXSAA_CACHE_DIR` wins

- platform defaults:
  - macOS: `$HOME/Library/Caches/crexx/crexxsaa`
  - Windows: `%LOCALAPPDATA%\crexx\crexxsaa`, falling back under `%USERPROFILE%\AppData\`
  - other Unix-like systems: `$XDG_CACHE_HOME/crexx/crexxsaa`, falling back to `$HOME/.cache/crexx/crexxsaa`

Runtime cache controls:

- `CREXXSAA_CACHE_DISABLE=1`: compile source through temporary files only
- `CREXXSAA_CACHE_REFRESH=1`: ignore a valid cache hit and replace it
- `CREXXSAA_CACHE_TRACE=1`: write cache decisions to `stderr`

The trace currently emits events such as `miss`, `hit`, `stale`, `refresh`, and `disabled`.

Compiler path controls:

- hosts normally call `crexxsaa_set_compiler(ctx, rxc, rxas, import_dir)`
- `CREXXSAA_RXC`, `CREXXSAA_RXAS`, and `CREXXSAA_IMPORT_DIR` override the configured compiler values

10.7 Cache maintenance tool

The CREXX build/install includes a `crexxsaa` maintenance binary in the normal `bin` directory. It is a troubleshooting tool for the compiled-script cache, not a script runner.

Common commands:

```
crexxsaa --location
crexxsaa --list
crexxsaa --clear
crexxsaa --cache-dir /tmp/crexxsaa-cache --list
crexxsaa --cache-dir /tmp/crexxsaa-cache --clear --list
```

Default invocation with no arguments prints the cache location and lists cache entries. `--location` alone prints only the resolved cache directory.

Example list output:

```
cache: /Users/adrian/Library/Caches/crexx/crexxsaa
source: /path/to/profile.the
  bucket: 0379ad70148bf7ca
```

```
  rxbin: /Users/adrian/Library/Caches/crexx/crexxsaa/v1/0379
      ad70148bf7ca/b0ec3f1ffcc51e4c.rxbin
  rxbin_size: 1216
  source_hash: 9100aa6921615883
  config_hash: 5af0757a39c4eba1
```

10.8 Technical cache layout

The cache schema is versioned. Current entries live under `v1`:

```
<cache-root>/v1/<source-key>/
  manifest
  <object-hash>.rxbin
```

`source-key` is an FNV-1a 64-bit hash rendered as 16 hex characters. It includes the host namespace and canonical source path when available.

`object-hash` is also an FNV-1a 64-bit hash rendered as 16 hex characters. It includes the source content hash and compiler/library configuration hash.

Manifest fields:

```
version=1
source_path=/absolute/or/supplied/source/path
source_hash=<16-hex-content-hash>
source_size=<bytes>
source_mtime=<mtime>
config_hash=<16-hex-config-hash>
rxbin=<absolute/cache/path/to/object.rxbin>
```

The content hash, not only the timestamp, decides whether source changed. Size and mtime are recorded for diagnostics and human inspection.

The configuration hash includes:

- cache schema and `CREXXSAA_ABI_VERSION`
- configured or environment-overridden `rxc` path
- configured or environment-overridden `rxas` path
- configured or environment-overridden import directory
- loaded `library.rxbin` path
- file size and mtime for compiler, assembler, import directory, and library path where the platform can stat them

Compile and update sequence:

- compute source hash and configuration hash
- read the source bucket manifest if it exists
- on a valid hit, load the cached `.rxbin`
- on miss, stale, or refresh:
  - run `rxc -i <import-dir> -o <temp-base> <source>`
  - run `rxas -o <temp-base> <temp-base>.rxas`
  - rename the temporary `.rxbin` into the source bucket
  - write a new manifest through a temporary file and rename it into place
  - remove the previous cached `.rxbin` for the bucket when it has been superseded

Invalidation:

- `crexxsaa_invalidate_source(ctx, source_path, namespace)`
- `crexxsaa_invalidate_all(ctx)`
- `crexxsaa_clear_cache(cache_dir_override)`
- `crexxsaa --clear`
- `crexxsaa --cache-dir DIR --clear`

The cache is disposable. Deleting it should affect performance only, not program semantics. The current implementation uses atomic file replacement for compiled objects and manifests, but it is not a full cross-process locking protocol. For troubleshooting, clear the cache while the host application is idle.

11 Compiler exits

This chapter introduces CREXX compiler exits: what they are, how to enable them, and how to use them effectively. It also shows concrete examples, including how variable shadowing works inside exit-generated code.

11.1 What are compiler exits?

A compiler exit is a compile-time hook written in REXX that can inspect tokens and inject replacement code into the current compilation unit. Exits are packaged as a module that the compiler (`rxc`) loads on demand.

This chapter primarily describes the current general-exit model. The ongoing `ADDRESS` / REXXSAA work has also introduced a certified/system-exit model for core-team-curated exits that may own reserved keywords and use a richer compiler-facing planning contract. The working design record is address_rexxsaa_-working.md.

Typical uses:

- Lightweight source transforms for diagnostics and logging (e.g., a `dump` helper that expands into runtime `say` statements).
- Compile-time queries over the token stream (`query` exit).
- Project-specific code generation patterns.

Injected code is grafted into the AST and then compiled like normal source.

11.2 Enabling exits

The compiler looks for an exit bundle (a packed `.rxbin`) in the import path.

- Provide an import path that contains your bundle (e.g., the build `bin` directory):

```
rxc -i /path/to/bin -o out in.rexx
```

- Select the exit module to load using the `RXCP_EXIT_MODULE` environment variable. For the built-in bundle this is `rxcexits`:

```
RXCP_EXIT_MODULE=rxcexits rxc -i /path/to/bin -o out in.rexx
```

Notes:

- The exit module is discovered dynamically at compile time.
- All exits within the bundle are registered (e.g., `dump`, `query`).

11.3 Built-in demo exits

The project ships a small bundle of exits primarily for demonstration and testing:

- `dump` — emits `say` statements to print variables (including arrays) with type information.
- `query` — queries token properties.

You can find their sources under `compiler/exits/` in the repository.

The more elaborate `pprint` exit is a standalone packaged example under `examples/exits/pprint`. It lowers `pprint array,...` to the companion `arrayformatdemo` module under `examples/functions/array-formatting`. Neither the exit nor those formatter procedures are part of the default exit bundle or the Level B/Level G library contract. The example README gives the explicit bundle and import paths.

11.4 Certified/System Exits

Current user exits are intended for non-reserved keywords. The compiler now also has certified/system exits for core-team-policed language features.

Current certified exits:

- `ADDRESS`
- `PARSE`

Key distinctions in the current model:

- Certified/system exits may own reserved keywords.
- General user exits should not be able to override those reserved bindings.
- Certified/system exits may use a richer planning response than the legacy string-only `pre_process()` contract.

Treat the working note as the design authority for the evolving model: address_-rexxsaa_working.md.

11.5 Example: Dumping variables and arrays

Given

```
options levelb
import rxfnsb
main: procedure
  a = .int[5]
  a[1] = 42
  dump a
  return 0
```

bundle:

```
RXCP_EXIT_MODULE=rxcexits rxc -i ./cmake-build-debug/bin -o test ./path
    /to/file.rexx
```

At compile time, `dump a` is replaced with a small loop that prints every element:

```
a (.int[5])[1] = 42
a (.int[5])[2] = 0
a (.int[5])[3] = 0
a (.int[5])[4] = 0
a (.int[5])[5] = 0
```

11.6 Shadowing and scoping inside exits

Exit replacements are grafted back into the main AST and then normalized like ordinary source. Structured replacements are supported, including nested `DO ... END`, `IF ... THEN/ELSE`, and nested instruction blocks. Local scopes for those generated blocks are materialized during the normal symbol-building passes, so identifiers created by injected code do not collide with or overwrite variables in the host program.

Practical consequences:

- A typed declaration inside an exit fragment (e.g., `i = .int`) creates an exit-local variable that is not visible outside the replacement block.
- If the host program already has a variable named `i`, it is not affected by the exit's internal `i`.

Illustration (conceptual)

```
options levelb
main: procedure
  i = .int; i = 7
  dump i   -- exit may internally use a loop variable named i
  say i    -- still prints 7; host i is untouched
  return 0
```

Behind the scenes, any generated grouped blocks are compiled with their own local scopes so that temporary loop counters and helper variables remain confined to the exit-generated block structure.

11.7 Writing your own exit (overview)

An exit is a Rexx module that exports a `process` procedure taking a token descriptor and returning a replacement string or an empty string.

High-level shape

```
namespace myexits expose process

process: procedure
  arg ti = .token
  -- Inspect ti.get_type(), ti.get_text(), ti.get_value_type(), ...
  -- Optionally build and return source replacement text
  return ""
```

Package your exit module into a bundle (`.rxbin`) and place it in the compiler's import path, then set `RXCP_EXIT_MODULE` to the bundle name.

11.8 Planning Hooks

Current behaviour:

- `pre_process()` is used for early structural planning such as hoisting implicit variables.
- General exits may still communicate that planning back to the compiler as a space-separated list of 1-based token indices.
- Certified exits can now return a structured `rxcp.exitplan` object for binding hoists, contextual keyword claims, and planning diagnostics.

Planned direction:

- the Stage 1 `ADDRESS` work continues to extend the structured planning model with richer typed planning data and broader address-environment contracts.

See the working note for the proposal details: address_rexxsaa_working.md.

Tips

- Prefer uncommon temporary names and make typed declarations for any locals you introduce in the replacement.
- For detailed compiler diagnostics, use `rxc -d2` or `rxc -d3`, but redirect both stdout and stderr to a temp log and inspect the log with `tail`, `sed`, or `grep` rather than streaming the raw trace to the terminal.

11.9 Troubleshooting

- `EXIT_BRIDGE_DISPATCH_FAILED` — The compiler could not find a handler for the exit keyword. Ensure your bundle is on the import path and `RXCP_EXIT_MODULE` names it correctly.
- Replacement compiles but behaves oddly — verify your replacement string is valid Level B Rexx and, when using loops, that all loop variables are de-

clared locally inside the replacement (typed) so they do not depend on host context.

PART II

Tools Reference

12

The crexx tool

The `crexx` tool is the convenience driver for common CREXX workflows. It can compile, assemble, execute, link, and package programs without requiring the user to call each toolchain binary by hand.

It is also useful in larger builds because it keeps the release defaults in one place: source-level defaults are delegated to `rxc` by file type, native packaging runs through `rxlink` before `rxcpack`, and source/binary import paths are passed to the compiler phase consistently.

12.1 Compile and run from source

List the source files that make up your program:

```
crexx a.crexx b.crexx
```

If `a.crexx` is the main program and imports functions from `b.crexx`, this compiles both files, loads their bytecode together and runs the program. With `.crexx` files, the shorter `crexx a b` form works too. No project file, library option or separate link command is needed for this ordinary workflow.

An `import` makes another module's declarations available to the compiler; it does not add that module's source to the files being built. For a program whose sources are `a.crexx` and `b.crexx`, list both. `crexx a` alone does not automatically compile `b.crexx`.

The ordinary command compiles each listed source on every invocation and runs the result. Add `--noexec` to compile without running. `--nocompile` explicitly reuses existing bytecode without checking whether its sources have changed. Pass program arguments after the final driver option, `--args`:

```
crexx a.crexx b.crexx --args "first argument" second
```

As a project grows, or when you repeatedly build the same application, consider the existing `--program` and `--library` modes described under Choosing how to build a program or library. They add incremental builds and named linked outputs; they are optional for the simple compile-and-run workflow above.

12.2 Options

Options are used to differentiate between the choices that can be made while building a program. All options have defaults so that they can be left out in standard cases.

12.3 Options description

The following options are available (single and double dashes work for all options):

`-help` Display help on the usage of this tool.

`-version` Display the version of this tool. This is the same as the compiler and interpretr version.

`-exec` Execute the compiled `.rxbin` under `rxvme` (default).

`-args` Stop `crexx` option parsing; all remaining command-line arguments are passed to the executed program as separate argv entries. Use this as the final driver option when program arguments contain spaces or characters that a shell would normally interpret.

`-noexec` Compile only; do not execute the resulting `.rxbin`.

`-compile` Compile all CREXX program files on the command line to `.rxbin` files (default).

`-nocompile` Skip the `rxc` and `rxas` phases and reuse an existing `<stem>.rxbin`.

-native Compile to a native executable; default `--nonative`. This produces an executable file for the current operating system and instruction set architecture. The native route now links the compiled program with `rxlink` before `rxcpack` generates C source.

-nonative Disable native packaging.

-library output Build the explicit source files as one incrementally maintained library and atomically publish `output.rxbin`. Independent compile/assemble actions run in parallel; linking starts only after the complete source wave succeeds. This mode builds without executing the result.

-program output Build the explicit source files as one incrementally maintained linked program and atomically publish `output.rxbin`. Add `-native` to publish a native executable as well. This mode builds without executing the result.

-jobs auto|count Bound parallel compile/assemble work for `-library` and `-program`. `auto` is the default: 30 workers on macOS and five elsewhere, capped by the number of source actions.

-rebuild Bypass the project action keys and rebuild every explicit source plus the final link.

-verbose[0-4] Report on progress; default verbose0, which only issues error messages when the compile fails. Verbose 2 shows the rendered argv used for the toolchain utilities rxc, rxas and rxvme. Verbose 3 includes options and source listings, while verbose 4 includes the contents of the generated assembly code.

-[no]colo[u]r Enable or disable colourized progress output.

-[no]optimize Enable or disable optimization.

-keep Keep compile/link intermediates (default).

-nokeep Delete compile/link intermediates after the run.

-decimal Use decimal arithmetic where the driver has to choose arithmetic mode.

-l[library path] Load a binary/runtime library. A value containing a directory component is used as the exact file path; a bare packaged name remains relative to `CREXX_HOME/bin`. This permits applications to name an approved library file without searching runtime paths. Runtime/native library loading is separate from the compiler's `-s` and `-i` import-discovery paths.

For native packaging, `crexx` now separates `-l` inputs into two groups:

- packaged CREXX libraries (`.rxbin` inputs such as `classlib`) are passed into `rxlink`
- plugin/static-native libraries are linked natively when a matching platform static library is available

This keeps the direct interpreter path fast while still producing compact native executables.

`-s[path]` or `--source path` Add an extra source import root for the `rxc` phase. This is for off-directory `.rexx` modules that should be visible to source import discovery.

`-i[path]` Add an extra raw binary import root for the `rxc` phase. This is for `.rxbin` imports discovered during compilation.

`--import-rxas` Allow the `rxc` phase to auto-import `.rxas` files from binary roots. This is off by default.

`--linkmap path` When using `-native`, ask `rxlink` to write a link map.

`--link-keep-source` When using `-native`, keep source/TRACE debug metadata in the linked intermediate instead of using the default stripped output.

`--link-keep-inline` When using `-native`, keep inline-body metadata in the linked intermediate. The native link strips this metadata by default because it is only needed by later compiler imports and debugging/tooling checks.

`-s`, `-i`, and `--import-rxas` control compile-time discovery. They do not add imported source files to the build or configure VM search roots. When the compiler selects a packaged RXBIN, its autoload hint can identify that package for the VM to find through runtime roots. Use `-l` to supply an existing runtime library explicitly.

Source files without an `OPTIONS` clause use the `rxc` file-type defaults: `.rexx` defaults to Level C Classic REXX, while `.crexx` and `.crx` default to Level G. Explicit `OPTIONS LEVELC` scripts use the normal source header and the driver's standard runtime module set, including the Classic compatibility runtime `rxfnsc`. Level C support is being extended in stages: supported Classic REXX shapes lower and run, while constructs outside the implemented slice are rejected with an unsupported-shape diagnostic.

The driver invokes its toolchain phases through the CREXX ADDRESS command environment using direct argv dispatch. That avoids platform shell parsing for normal compile, assemble, link, pack, native-compile, and execute steps while keeping verbose output readable.

12.4 Choosing how to build a program or library

Start with `crexx a.crexx b.crexx` to compile and immediately run a program from its listed sources. This remains useful for quick experiments, inspecting individual module outputs, and deliberately repeating compilation during debugging.

For repeated development of a larger application, `--program` avoids recompiling unchanged work and produces one named linked program. `--library` provides the same incremental build support for reusable code built separately from its consumers. Both are available now; neither is required merely because a program has more than one source file.

Need	Command
Compile and run a program now	`crexx source.crexx`
Compile and run several source modules together	`crexx a.crexx b.crexx`
Compile sources without running the program	`crexx a.crexx b.crexx --noexec`
Build one maintained linked RXBIN incrementally	`crexx --program output sources...`
Build a native version of that linked program	`crexx --program output sources... --native`
Build a reusable linked library incrementally	`crexx --library output sources...`
Build a product with native components, generators, packaging or broad QA	CMake

An installed Release toolchain therefore supports ordinary REXX program and library development without requiring a CMake project:

```
crexx --library build/mylib source1.crexx source2.crexx --jobs auto
crexx --program build/myprogram main.crexx support.crexx --jobs auto
crexx --program build/myprogram main.crexx support.crexx --jobs auto --
    native
```

`--program` and `--library` are build-only modes: they do not run the output and do not accept `--args`. After a program build, run the result separately, for example

`rxvme build/myprogram.rxbin`. The ordinary command leaves separate module outputs; the project modes publish one linked output and keep their member artifacts and incremental state under `<output>.crexx-build`.

All these source-based modes require the program or library's source members to be listed explicitly. Automatic dependency tracking decides when to rebuild those members; it does not add further sources to the linked product. Projects that also own native code, plugins, generators, installation or extensive QA should use CMake as their outer build system.

These modes optimize REXX bytecode by default. Use `--nooptimize` when checking optimizer parity or diagnosing generated code. Existing `-s`, `-i`, `-l`, `--import-rxas`, diagnostics and locale options keep their normal meanings. Packaged RXBIN imports retain their autoload hints, so a linked program can load a separately published dependency from an `rxvm -l` location.

The builder records project and member content keys under `<output>.crexx-build`. A repeat with unchanged tools, options, sources and import candidates prints `SKIP: project current` and does no compilation, assembly or linking. Normally this fast path does not start `rxc`. With `--import-rxas`, the compiler's cheap checks still run because timestamp-only changes can select a different artifact. `--rebuild` forces every member and the final link.

When project inputs change, each member first checks its own source, compiler, assembler, wrapper and compile options. If those still match, the driver calls `rxc --check-project-dependencies` on the snapshot retained by that member's successful compile. Only members with changed action or dependency inputs enter `WAVE: project compile/assemble jobs=N`; current members print `SKIP: project member current`. Here `N` is the number selected, while `--jobs` bounds concurrent workers. The final link still includes every declared member.

An implementation edit in an independent source normally rebuilds that member. An edit in an imported source also rebuilds its consumers, including for private implementation changes, because cross-file optimization can inline its body. Contract changes invalidate those consumers too. Discovery and namespace-header changes are conservative: adding/removing a candidate, changing a previously excluded source's namespace, or changing root precedence can invalidate more members. Binary candidates are fingerprinted even if not ultimately loaded. This is safe reuse based on compiler inputs, not a promise of the smallest possible se-

mantic dependency graph.

The compiler owns dependency discovery; the wrapper does not parse Rexx import syntax. Dependency snapshots are created and maintained automatically. Users do not list implicit dependencies or edit a manifest. Missing or corrupt snapshots force compilation when member reuse is checked. Compiler options such as optimization, RXAS imports, diagnostics and locale participate in action keys. Compiler environment variables are included too: `RXCP_DISABLE_EXIT` (including an empty but present value), `RXCP_EXIT_MODULE`, `CREXX_DIAGNOSTICS`, `CREXX_DIAGNOSTIC_LOCALE`, `CREXX_MESSAGE_PATH`, `LC_ALL`, `LC_MESSAGES` and `LANG`. An explicit exit-module file is fingerprinted. If a custom compiler exit reads other external inputs, keep them stable and use `--rebuild` when they change; the builder cannot infer arbitrary file or environment reads by custom code. Changes to the compiler/assembler/wrapper rebuild members; linker-only changes require relinking. The builder fingerprints candidates from implicit installed binary roots as well as explicitly configured roots. CMake remains useful for generated sources, provider builds and native packaging; it invokes this same supported driver route.

Each member writes only in its private action directory. The link writes a private RXBIN and renames it over the public output only after success. A compiler, assembler or linker failure therefore leaves the last published library/program intact. A change in content inputs during the compile wave also aborts publication and invalidates its member stamps. RXAS-enabled waves recheck compiler dependency snapshots before publishing to catch timestamp-only selection changes too; retry once inputs are stable.

12.5 Examples

12.5.1 Just run it

The simplest way to run a CREXX program is to specify its source files as input to `crexx`. It compiles and assembles each listed file, then runs their bytecode together through `rxvme`, which includes the shipped bytecode libraries. Separately built application libraries must be supplied explicitly or be discoverable through packaged-library autoload.

```
options levelb
import rxfnsb
```

```
say 'hello crexx!'
say 'today''s date is:' date()
```

We will run that with:

```
crexx hello
```

```
hello crexx!
today's date is: 14 Sep 2026
```

12.5.2 Make an executable module

CREXX can make a native executable from a REXX program. For this, the `rxlink` utility will be called, to arrange the modules into an `.rxbin` library, while insuring that there is no duplication of resources. This `.rxbin` will be processed by the `rxcpack` program, after which the `gcc` or `clang` compilers and the os linkage editor will do their work.

The interesting part here is that all this is accomplished by the `--native` option on the CREXX utility. When we take the previous program and compile it with:

```
crexx hello --native
```

then we will have a `hello` program (`hello.exe` on windows) which can be executed by specifying its name in the shell, or even by double clicking it in the File Explorer, Finder or equivalent OS GUI interface. This program can be executed on machines without a CREXX installation.

12.5.3 Compile and run a program with commandline arguments

When using the CREXX compiler driver, commandline arguments as input to the program can be specified with the `--args` option; *this option needs to be the last option on the CREXX command*, because everything that follows will be sent to the program as a string array (`.string[]`) in the execution phase.

Let's say we have called this program *hello2.crexx*:

```
main: procedure
   arg sources=.string[]
   loop i=1 to sources.0
     say sources.i
   end
```

If we execute the above program with the commandline:

```
crexx hello2 --args one two three
```

We will see the following output:

```
one
two
three
```

12.6 Verbosity

With the default verbosity level the tools behaves in the standard unix way where a lack of messages indicates success. This level can be increased gradually to a full explanation of everything that is done. The default is `--verbose0`, which gives no inidcation of what happened unless something went wrong. This is the way to run known-good programs without any overhead.

All verbosity level examples run the following short script:

```
options levelb
import rxfnsb
say 'hello rexx!'
say 'today it''s' date('w')
'echo 42'
```

12.7 `--verbose1`

With `--verbose1`, the driver tells in a condensed way what it did and how it went. When the return codes from the `rxc` and `rxas` are 0, these are displayed with an 'OK' between square brackets.

```
[ OK ] rxc      - Compiled        hello.crexx
[ OK ] rxas     - Assembled       hello.rxas
hello rexx!
today it's Monday
42
```

When everything works out well, it issues some reassuring messages about the

compiler and the assembler running successfully and skips the starting of the runtime engine, because the output of the program follows these messages. When errors occur, these are signalled as in the `-verbose0` setting, with some extra indication in which phase of compilation, link and/or executing the problem took place.

12.8 `--verbose2`

With the `--verbose2` setting, there is more tourist information.

```
CREXX compiler driver crexx-1.0.0-beta.3+local.g4c4f4c48643e (macOS 64
20260914)
cREXX License (MIT)
Copyright (c) 2020-2026 Adrian Sutherland, Peter Jacob, René Jansen
See https://github.com/adesutherland/CREXX for project details
using relpath   : /Users/rvjansen/apps/crexx_release/
using lpath     : bin
using s roots   :
using i roots   :
rxc command        : "/Users/rvjansen/apps/crexx_release/bin/rxc" "-i"
"/Users/rvjansen/apps/crexx_release/bin" "-o" "hello" "hello.crexx"
[ OK ] rxc      - Compiled        hello.crexx
rxas command      : "/Users/rvjansen/apps/crexx_release/bin/rxas" "-o"
"hello" "hello"
[ OK ] rxas     - Assembled       hello.rxas
crexx executes: "/Users/rvjansen/apps/crexx_release/bin/rxvme" "hello"
"/Users/rvjansen/apps/crexx_release/bin/rx_treemap" "/Users/rvjansen/apps/crexx_-
release/bin/rx_llist"   "/Users/rvjansen/apps/crexx_release/bin/rx_keyac-
cess" "-a"
hello rexx!
today it's Monday
42
```

It starts by identifying the exact release of the crexx version. After this, a number of paths are shown:

Name	Meaning
relpath	The path where the CREXX system is installed
lpath	The path from which executables and libraries are found
s roots	additional sourcefile lookup locations
i roots	additional binary (.rxbin) lookup locations

Source-level defaults are owned by `rxc`, so the driver does not insert language or import flags for headerless files. The verbose output shows the exact `rxc` command and the source extension passed to the compiler.

With this verbosity level, the exact invocations of the tools in the toolchain are documented, including the complete paths and arguments. Also, the invocation of the runtime engine, with all libraries explicity named - and this includes the libraries that are not used. With native compiles, the `rxlink` tool will extract the used code from these libraries into the executable. It also shows the `-a` flag as a last option, even if this is unused.

12.9 --verbose3

In verbosity level 3, the output level is on par with traditional IBM compiler output, with a complete summary of used and unused compiler options.

```
CREXX compiler driver crexx-1.0.0-beta.3+local.g4c4f4c48643e (macOS 64
20260914)
cREXX License (MIT)
Copyright (c) 2020-2026 Adrian Sutherland, Peter Jacob, René Jansen
See https://github.com/adesutherland/CREXX for project details
======>  RexxLA  cREXX  compiler  crexx-1.0.0-beta.3+local.g4c4f4c48643e
14 Sep 2026 17:12:12
INVOCATION OPTIONS
Options in effect:
NOCOLOR
DECIMAL
EXEC
NONATIVE
KEEP
```

```
OPTIMIZE
NOIMPORT-RXAS
VERBOSE 3
SOURCE ROOTS
BINARY ROOTS
using relpath   : /Users/rvjansen/apps/crexx_release/
using lpath     : bin
using s roots   :
using i roots   :
rxc  command        :  "/Users/rvjansen/apps/crexx_release/bin/rxc"  "-i"
"/Users/rvjansen/apps/crexx_release/bin" "-o" "hello" "hello.crexx"
[ OK ] rxc      - Compiled        hello.crexx
rxas  command       :  "/Users/rvjansen/apps/crexx_release/bin/rxas"  "-o"
"hello" "hello"
[ OK ] rxas     - Assembled       hello.rxas
======>  RexxLA  cREXX  compiler  crexx-1.0.0-beta.3+local.g4c4f4c48643e
14 Sep 2026 17:12:12
crexx  executes:  "/Users/rvjansen/apps/crexx_release/bin/rxvme"  "hello"
"/Users/rvjansen/apps/crexx_release/bin/rx_treemap" "/Users/rvjansen/apps/crexx_-
release/bin/rx_llist"   "/Users/rvjansen/apps/crexx_release/bin/rx_keyac-
cess" "-a"
hello rexx!
today it's Monday
42
```

12.10 `--verbose4`

Verbosity level 4 is for the moment the most verbose level of tool output. In this level, the complete Rexx input source is expanded (when it is produced by the `rxpp` preprocessor), and all generated assembler output is visible and available for inspection and debugging, as well as the set of generated import statements for runtime linking. It is advisable to page this output through a `less`-like processor.

```
CREXX compiler driver crexx-1.0.0-beta.3+local.g4c4f4c48643e (macOS 64
20260914)
```

```
cREXX License (MIT)
Copyright (c) 2020-2026 Adrian Sutherland, Peter Jacob, René Jansen
See https://github.com/adesutherland/CREXX for project details
======>  RexxLA  cREXX  compiler  crexx-1.0.0-beta.3+local.g4c4f4c48643e
14 Sep 2026 17:12:12
INVOCATION OPTIONS
Options in effect:
NOCOLOR
DECIMAL
EXEC
NONATIVE
KEEP
OPTIMIZE
NOIMPORT-RXAS
VERBOSE 4
SOURCE ROOTS
BINARY ROOTS
using relpath   : /Users/rvjansen/apps/crexx_release/
using lpath     : bin
using s roots   :
using i roots   :
======>  RexxLA  cREXX  compiler  crexx-1.0.0-beta.3+local.g4c4f4c48643e
14 Sep 2026 17:12:12
LINENUM  ----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+
000001 options levelb
000002 import rxfnsb
000003 say 'hello rexx!'
000004 say 'today it''s' date('w')
000005 'echo 42'
rxc  command        :  "/Users/rvjansen/apps/crexx_release/bin/rxc"  "-i"
"/Users/rvjansen/apps/crexx_release/bin" "-o" "hello" "hello.crexx"
[ OK ] rxc      - Compiled        hello.crexx
======>  RexxLA  cREXX  compiler  crexx-1.0.0-beta.3+local.g4c4f4c48643e
14 Sep 2026 17:12:12
```

```
LINENUM  ----+----+----+----+----+----+----+----+----+----+----+----+----
+----+----+
000001 /*
000002  * SOURCE                 : hello.crexx
000003  */
000004
000005 .globals=0
000006
000007 main() .locals=8
000008    .meta "hello.main"="b" ".void" main() ""
000009    numsci 18,1,1
000010    .meta "hello.main.rc"="b" ".int" r0
000011    .srcstep 1 1 6 "hello.crexx" 5 1 10 "'echo 42'"
000012    .srcstep 2 2 17 "hello.crexx" 3 1 18 "say 'hello rexx!'"
000013    say "hello rexx!"
000014   .srcstep 3 3 17 "hello.crexx" 4 1 28 "say 'today it''s' date('w')"
000015    load r2,"w"
000016    .traceevent "L" 4 "R" "S" "r" 2 3 3 0 "" ""
000017    load r1,5
000018    settp r2,768
000019    settp r3,512
000020    settp r4,512
000021    settp r5,512
000022    settpcall r7,rxfnsb.date(),r1,r6,512
000023    .traceevent "F" 4 "R" "S" "r" 7 3 3 0 "date" ""
000024    sconcat r7,"today itś",r7
000025    .traceevent "O" 4 "R" "S" "r" 7 3 3 0 "" ""
000026    say r7
000027     .srcstep 4 4 33 "exit_fragment" 2 1 56 "rc=_address('','echo
42',_noredir(),_noredir(),_noredir"
000028    load r3,""
000029    .traceevent "L" 4 "R" "S" "r" 3 4 4 0 "" ""
000030    load r4,"echo 42"
000031    .traceevent "L" 4 "R" "S" "r" 4 4 4 0 "" ""
000032    load r1,0
```

```
000033    call r5,_rxsysb._noredir(),r1
000034    .traceevent "F" 4 "R" "O" "r" 5 4 4 0 "_noredir" ""
000035    load r1,0
000036    call r6,_rxsysb._noredir(),r1
000037    .traceevent "F" 4 "R" "O" "r" 6 4 4 0 "_noredir" ""
000038    load r1,0
000039    call r7,_rxsysb._noredir(),r1
000040    .traceevent "F" 4 "R" "O" "r" 7 4 4 0 "_noredir" ""
000041    load r2,5
000042    settp r3,768
000043    settp r4,768
000044    settp r5,256
000045    settp r6,256
000046    settpcall r0,_rxsysb._address(),r2,r7,256
000047    .traceevent "F" 4 "R" "I" "r" 0 4 4 0 "_address" ""
000048    .traceevent "A" 6 "R" "I" "r" 0 4 4 0 "rc" ""
000049    .srcstep 5 5 1 "hello.crexx" 5 10 10 "'echo 42'"
000050    ret
000051    .meta "hello.main.rc"
000052
000053 /* Imported Declaration from file: date@library.rxbin */
000054
000055 rxfnsb.date() .expose=rxfnsb.date
000056   .meta "rxfnsb.date"="b" ".string" rxfnsb.date() "?oformat=.string,?idate=.string,?i
000057    .meta "rxfnsb.date"=".autoload" "library"
000058
000059 /* Imported Declaration from file: _address@library.rxbin */
000060
000061 _rxsysb._noredir() .expose=_rxsysb._noredir
000062      .meta  "_rxsysb._noredir"="b"  "._rxsysb..addressredirect"  _-
rxsysb._noredir() ""
000063    .meta "_rxsysb._noredir"=".autoload" "library"
000064
000065 /* Imported Declaration from file: _address@library.rxbin */
000066
```

```
000067 _rxsysb._address() .expose=_rxsysb._address
000068   .meta "_rxsysb._address"="b" ".int" _rxsysb._address() "?env=.string,?command=.strin
...=.string"
000069    .meta "_rxsysb._address"=".autoload" "library"
rxas  command      :  "/Users/rvjansen/apps/crexx_release/bin/rxas"  "-o"
"hello" "hello"
[ OK ] rxas     - Assembled       hello.rxas
======>  RexxLA  cREXX  compiler  crexx-1.0.0-beta.3+local.g4c4f4c48643e
14 Sep 2026 17:12:12
crexx  executes:  "/Users/rvjansen/apps/crexx_release/bin/rxvme"  "hello"
"/Users/rvjansen/apps/crexx_release/bin/rx_treemap" "/Users/rvjansen/apps/crexx_-
release/bin/rx_llist"   "/Users/rvjansen/apps/crexx_release/bin/rx_keyac-
cess" "-a"
hello rexx!
today it's Monday
42
```

12.11 `--verbose[2-4]` with native compilation

When the program is compiled and linked to a native executable, no execution of this program follows. Instead the arguments to the `rxpack` object packager and the linkage editor `rxlink` and their results are included.

Also, the complete command line to the c-compiler and its linkage editor step is documented here, and can be copied into private building tools or scripts. Here the use of static import libraries for the native executables can be seen.

In the following chapters, these tools are documented in detail.

13 rxc - the CREXX Compiler

13.1 Command Line Options

```
cREXX Compiler
Version : crexx-1.0.0-beta.3+local.g4c4f4c48643e
Usage : rxc [options] source_file
Options :
-h Prints help message
-c Prints Copyright & License Details
-v Prints Version
-d[level] Debug Mode (Optional Level)
-dp Stop after parsing and validation
-l location Working Location (directory)
-s source Source import locations - ";" delimited list
-i import Binary import locations - ";" delimited list
--level level Default source level when OPTIONS omits one
--import ns Inject a file-level IMPORT namespace (repeatable)
--import-rxas Enable auto-import scanning of .rxas in binary roots
--autoload Emit packaged-RXBIN autoload hints (default)
--no-autoload Do not emit packaged-RXBIN autoload hints
--no-exe-import Do not add the executable directory to binary roots
```

```
--import-resolution-report path Write observe-only import selection
JSON
--project-dependencies path Write a private project dependency snap-
shot
--check-project-dependencies path Check a snapshot with the same com-
pile arguments
--diagnostics mode Diagnostic rendering: localized or raw
--diagnostic-locale locale Diagnostic locale such as en_GB or en_US
--no-localisation Use raw diagnostic code/parameter rendering
-o output_stem RXAS output stem or .rxas file
-n No Optimising
-x Disable compiler exits
--syntaxhighlight Run as a DSLSH syntax-highlighting server (stdio
mode)
--port port Run as a DSLSH syntax-highlighting server (socket mode on
port)
```

13.2 Import Search Roots

`rxc` separates automatic import discovery into two root classes:

- source roots, searched for CREXX source files
- binary roots, searched for `.rxbin`, optional `.rxas`, and `.rxplugin`

The recommended source extensions are:

- `.crexx`: canonical CREXX source, defaulting to Level G when no `options level...` clause is present
- `.crx`: short CREXX source alias, also defaulting to Level G
- `.rexx`: compatibility/classic source, defaulting to incremental Level C lowering

An initial source file may also use another extension, such as `.the` for an editor macro integration. That extension is added to source-root discovery for that compile and defaults to Level G unless the source states another level explicitly. This supports host-specific source names without making every possible extension a recommended CREXX extension.

`.rxpp` remains the preprocessor-oriented source extension for existing RXPP workflows. Longer term, preprocessing may become idempotent and better integrated so ordinary `.crexx` and `.crx` sources can pass through it safely, but that is separate from the source import extension rules.

The primary source root is the directory of the source file being compiled. If the source file name does not include a directory, the compiler uses the working location from `-l`. Additional source roots can be supplied with `-s`, using a semicolon-delimited list.

Binary roots are controlled separately with `-i`. The compiler always includes the executable directory in the binary-root set so deployed libraries remain visible without adding them explicitly. `-i` can be repeated, and repeated `-s` options are accumulated in the same way. Toolchain and library self-builds can pass `--no-exe-import` to suppress that implicit root and use only their explicit binary dependencies; this prevents a previous installed or build-tree library image from competing with current source interfaces. Normal application compilation retains the executable-root default. The source file's directory is not automatically treated as a binary root; pass `-i .` or `-i build-dir` when a sibling or build-output `.rxbin` is an intended compile-time dependency.

Search order is:

1. primary source root
2. extra `-s` source roots
3. binary roots for `.rxbin`
4. binary roots for `.rxas` when `--import-rxas` is enabled
5. binary roots for `.rxplugin`

This keeps same-project source imports preferred over deployed binary artifacts, while stopping the compiler from treating every binary directory as a source tree. It also prevents stale generated `.rxbin` files left beside edited source files from silently satisfying imports.

13.3 Import Discovery Notes

Automatic `.rxas` import scanning is disabled by default. It can be re-enabled with `--import-rxas` for workflows that intentionally use assembler modules as import

sources.

For source imports, `rxc` performs a cheap header scan before doing a full parse. That scan reads the leading `options`, `namespace`, and `import` clauses so the compiler can reject namespace-mismatched source files early and avoid unnecessary full validation work.

Within each source root, the compiler looks for `.crexx`, `.crx`, `.rexx`, and the initial file's extension when it is different from those standard names. Source files with the same module stem are de-duplicated within that source stage so one source module is imported for a stem.

Within a single binary root, when multiple artifacts share the same module stem, the compiler keeps only the freshest candidate. If timestamps tie, `.rxbin` wins over `.rxas`.

13.4 Project dependency checks

The `crexx --program` and `crexx --library` driver uses two compiler options to reuse individual compiled members safely. The driver manages these files automatically; ordinary project users do not need to invoke these options or maintain a dependency list:

- `--project-dependencies path` writes a private dependency snapshot after a successful compilation.
- `--check-project-dependencies path` checks a retained snapshot without parsing, validating or optimizing callable bodies, emitting assembly, or loading dynamic provider code. Exit status is 0 for current inputs and 2 for a stale, missing, unreadable or malformed snapshot. Other argument/input failures are ordinary compiler failures; callers must treat every nonzero status as a miss.

Place these options before the source argument. Use the same source, working location, ordered import roots and compiler options for creation and checking:

```
rxc --project-dependencies build/member.dependencies -s src -i lib -o
    build/member src/member.crexx
rxc --check-project-dependencies build/member.dependencies -s src -i
    lib -o build/member src/member.crexx
```

The snapshot includes the primary source, resolver options, compiler environment settings and ordered candidate selection. Loaded source files and binary candidates are checked by full content; excluded source files are checked by the namespace interpretation used by the compiler's existing header scanner. Candidate additions, removals, earlier-root shadowing and timestamp-based RXAS selection invalidate the snapshot. A private implementation change in an imported source can require rebuilding a consumer: normal optimization may have copied that implementation into its inline code.

The snapshot is an internal versioned binary format, not a Make dependency file or the observe-only `--import-resolution-report` JSON. The driver also checks the compiler, assembler, wrapper, diagnostics and action options; the snapshot alone does not establish that a different toolchain can reuse an earlier output. Keep inputs stable during a build. The driver rejects a wave whose content inputs change while its members compile. Deleting `<output>.crexx-build` or using `--rebuild` requests a fresh build. No optimization or callable-validation rules are relaxed by dependency checking. The driver also fingerprints an explicit `RXCP_EXIT_MODULE` file. Custom exits that read additional external inputs require caller-managed invalidation (`--rebuild` in the project driver).

Compiler declaration lookups inspect declaration-bearing file nodes instead of repeatedly walking expanded executable bodies. During binary metadata loading, classes/interfaces declared by that module are recognized as forward declarations before their full stubs are registered. This prevents an unrelated source file in the same namespace from being loaded merely to resolve that temporary gap. These are internal compiler repairs; normal optimization and callable validation stay enabled and no user-facing optimization switch is required.

13.5 Inline Assembler

On page 103 the inline assembler function of the CREXX compiler is discussed. This enables the incorporation of `rxas` assembler instructions into a REXX source file.

13.6 Optimizer

The compiler can do a number of optimizations that can make the execution of a program much faster; the next example shows how an operation can be done at compile time, to avoid instruction scheduling and execution at runtime:

```
options levelb
say 0.5**2 'should be 0.25'
```

```
 0.25 should be 0.25
```

```
/*
 * SOURCE                 : fpowtest.crexx
 */

.globals=0

main() .locals=0
   .meta "fpowtest.main"="b" ".void" main() ""
   numsci 18,1,1
   .srcstep 1 1 17 "fpowtest.crexx" 2 1 28 "say 0.5**2 'should be 0.25'
       "
   say "0.25 should be 0.25"
   .srcstep 2 2 1 "fpowtest.crexx" 2 28 28 "say 0.5**2 'should be 0.25'
       "
   ret
```

This works because, for a large number of operations, the `rxc` compiler can assume the result is never going to be different, and will determine that result during compile time. In the same vein, results from operations that are not displayed or handled further in the program, will lead to the operation being skipped entirely.

13.7 Debugging and Validation

For compiler developers and advanced users, `rxc` provides debug modes that enable internal validation of the AST and Symbol table and stress-test the fixpoint loop.

- `-d2` --- Structural Validator
    - Runs the built-in AST/Symbol validator between passes.

    - Catches parent/child inconsistencies, broken bi-directional symbol links, and illegal scope parentage.
- `-d3` --- Validator + Idempotency Stress Test
    - Everything in `-d2`, plus: the fixpoint loop is forced to run extra iterations and selected walkers are invoked multiple times per iteration.
    - Proves walker idempotency and overall convergence.

Example:

```
# Validate (structure only)
rxc -d2 -o out input.rexx

# Stress idempotency and convergence
rxc -d3 -o out input.rexx
```

Notes:

- These modes are for diagnostics and do not change user-visible semantics.
- All walkers inside the fixpoint loop are designed to be idempotent.
- See also: `compiler/docs/fixpoint_idempotency_and_scope.md` for design details.
- See also: `compiler/docs/testing.md` for information on regression testing and how to update golden files.

14

rxas - the CREXX Assembler

The purpose of the CREXX assembler `rxas` is to translate a text file with `rxvm` assembler instructions to a file with binary contents containing these instructions in their binary, executable form. Its main use is to translate an `rxas` file produced by the CREXX compiler `rxc` to a binary `.rxbin` object module.

14.1 Program Structure

The assembler processing goes through a number of steps in a single pass: first, the Lexer / Scanner tokenises the `.rxas` code. After that, the Parser parses the structure into a series of instructions. The binary writer generates the binary code and constant pool for the program at hand. The backpatcher runs last and handles forward references.

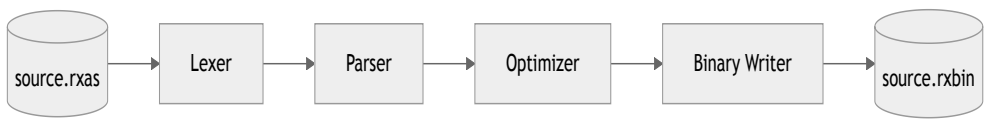


FIGURE 1: Program Structure

14.2 Input/Output

The `rxas` assembler has a `.rxas` file as input and produces an `rxbin` file as output, which can be considered an *object module*, as it has unresolved addresses, which can be resolved by the linkage editor component of the `rxvm` virtual machine interpreter. It also produces a report to stdout (in case of errors only) and can produce a trace file in Debug/verbose mode (option `-d`).

14.3 Character sets

The input file is assumed to be valid utf-8 and will fail invalid input. The assembler, like the compiler, operates using two character sets. The first is for symbols in the assembler language statements. These are all composed of the ASCII subset of Unicode. The second character set is used for data; the contents of variables. Here the whole of Unicode can be used.

14.4 Command Line Arguments

When the command line argument -h is specified the options are shown:

```
cREXX Assembler
Version : crexx-1.0.0-beta.3+local.g4c4f4c48643e
Usage : rxas [options] source_file
Options :
-h Help Message
-c Copyright & License Details
-v Version
-a Architecture Details
-i Print Instructions
-d Debug/Verbose Mode
-l location Working Location (directory)
-o output_stem RXBIN output stem or .rxbin file
-n No Optimising
```

```
Notes :
- source_file : The source file to be assembled; filetype (rxas) is
added to the name.
```

14.5 Optimizer

The assembler contains an optimizer; this is different from the optimizer which is part of the compiler. This phase of the assembler is running always, except when switched off by the `-n` option. When there is any doubt whether any encountered problem is caused by the optimizer, switching it off can help diagnosing the problem.

14.6 Assembler Directives

For machine instructions, see the *cRexx VM Specification* publication. This section discusses instructions to the assembler, which are called *directives* to clearly distinguish them from virtual machine instructions. These are necessary to pass information into the compiled `.rxbin` binary file, to enable execution by the `rxvm` virtual machine. In the following itemized list, *italic* descriptors are categories, while items in roman type are literal directives.

comments A block comment can be made by surrounding the text block with `/*` and `*/` indicators. The `*` (asterisk) can be used as a line comment. The remainder of the line after a line comment is ignored.

labels A label (a string ending with a colon, indicated in the machine instructions documentation as an `ID`, is a target for branching-type instructions.

Example:

```
loop:
    fndnblnk r3,r1,r3   /* find first/next non blank offset         */
    ilt r5,r3,0         /* if <0, nothing found, end search         */
    brt break,r5        /* branch to break                          */
    inc r6              /* else increase word count                 */
                        /* offset of word is in R3                  */
    copy r8,r3          /* save offset of word                      */
    fndblnk r3,r1,r3    /* from offset find next blank offset       */
    ieq r7,r6,r2        /* is this the word we are looking for?     */
    brt wordf,r7        /* go fetch it                              */
```

```
ilt r5,r3,0         /* if <0, nothing found, end search         */
brt break,r5        /* no word found                            */
bct loop,r4,r3      /* continue to look for next non blank char */
```

registers `r0 ...*n` are names of registers, indicated in the machine instructions documentation as `REG`.

.globals=*INT* Defines *int* global variable *g0* ... *g*n. These can be used within any procedure in the file.

.locals=*INT* The number of local registers (local to the source program). This number needs to be 1 greater than the highest used register number.

.expose=*ID* Any global register marked as exposed is available to any file which also has the corresponding exposed index/name.

.srcstep Used to document source-step metadata. This optional directive records a self-contained source anchor: module-local step id, clause id, provenance flags, source file, source line number, active column range, and the whole source line. It is used by TRACE, SOURCELINE, panic reporting, and debuggers.

.traceevent Used to document semantic TRACE events. This optional directive records compact event/value codes, mode visibility, value source, value type, source-step id, clause id, and optional symbol or resolved compound names. TRACE handlers use these records instead of guessing values from source text.

.proc A procedure is a scope delimiting mechanism for the access of registers. The registers of a procedure are independent of the caller's registers. The VM maps its registers to the registers in the caller. Each time a procedure/function is called a new *stack frame* is provided. This means that the called function has its own set of registers.

The function header defines how many registers (called locals) the function can access - for practical purposes one can consider that any number of registers can be assigned to a function. In addition, each file defines a number of global registers that can be shared between procedures.

14.7 Examples

Here are several examples of how to use `rxas` to assemble a program into an object module.

14.7.1 In-line assembly

The CREXX compiler `rxc` enables[5] inline assembly through the `assembler` statement. When used in this way, a lot of the complications of an assembly language program can be handled by the CREXX compiler, like assigning registers to variables, and the conversion of datatypes like `integer` for display as `string`.

```
/* crexx ipow test - inline assembler */
options levelb

number = 10
power = 2
result = 0

say 'test IPOW'
assembler do
   ipow result,number,power
end
say "result =" result /* The compiler converts integer to string */
```

```
test IPOW
result = 100
```

This is a simple and straightforward way to complement the low level assembler instructions with the power of the REXX language. The following example intends to explain how this is implemented; it can be skipped without consequences.

In this example, the compiler generates the following assembler source:

```
/*
 *SOURCE : pow.crexx
 */

.globals=0

main() .locals=3
```

[5] When used with `options level b`

```
.meta "pow.main"="b" ".void" main() ""
numsci 18,1,1
.srcstep 1 1 17 "pow.crexx" 5 1 10 "power = 2"
.meta "pow.main.number"="b" ".int" "10"
.meta "pow.main.power"="b" ".int" r0
load r0,2
.traceevent "L" 4 "R" "I" "r" 0 1 1 0 "" ""
.traceevent "A" 6 "R" "I" "r" 0 1 1 0 "power" ""
.srcstep 2 2 17 "pow.crexx" 6 1 11 "result = 0"
.meta "pow.main.result"="b" ".int" r1
load r1,0
.traceevent "L" 4 "R" "I" "r" 1 2 2 0 "" ""
.traceevent "A" 6 "R" "I" "r" 1 2 2 0 "result" ""
.srcstep 3 3 17 "pow.crexx" 8 1 16 "say 'test IPOW'"
say "test IPOW"
.srcstep 4 4 17 "pow.crexx" 10 4 28 " ipow result,number,power"
.traceevent "V" 6 "R" "I" "r" 1 0 0 0 "result" ""
.traceevent "V" 6 "R" "I" "r" 0 0 0 0 "power" ""
ipow r1,10,r0
.srcstep 5 5 17 "pow.crexx" 12 1 22 "say \"result =\" result /*The compiler converts
     integer to string */"
itos r1
.traceevent "V" 6 "R" "S" "r" 1 5 5 0 "result" ""
sconcat r2,"result =",r1
.traceevent "O" 4 "R" "S" "r" 2 5 5 0 "" ""
say r2
.srcstep 6 6 1 "pow.crexx" 12 22 22 "say \"result =\" result /*The compiler converts
     integer to string */"
ret
.meta "pow.main.number"
.meta "pow.main.power"
.meta "pow.main.result"
```

The `.srcstep` directives (intended for trace, sourceline, panic reports, and debuggers) indicate where the work is done. Related `.traceevent` directives identify values that are actually available for `trace` display. The variables are assigned, as integers, to the registers `r1` and `r2`. The line `ipow, number, power` becomes `ipow r3,r1,r2`, and the display on the terminal is handled by the `itos`,`sconcat` and `say` instructions.

This is an example, with the remark that in this case, the microcode for `ipow` is always executed, the example in `crexx` on page **??** shows that the CREXX optimizer of the compiler can eliminate this code entirely.

The use of assembler directives[6] is not allowed in inline assembly, so it is (for example) not possible to define procedures in an inline assembler block.

14.8 Troubleshooting

The assembler will give messages when there are problems in a source file. These are hopefully of enough clarity to resolve the immediate problem with syntactic issues or typos. When a program assembles correctly but its behaviour is unexpected, or its output is incorrect, a number of different strategies can be followed.

14.8.1 Adding say statements

It is easy to add `say` statements to your program. Unlike Rexx, there is no trace statement for assembler programs. However, it is easy to disassemble (see rxdas on page **??**) an `.rxbin` module, and reassemble it with added statements.

14.8.2 Using the debugger

The `rxdb` debugger has a mode for assembler. This can be used to set breakpoints and/or step through the code; here the registers can be traced so variables in your program can be followed and the comparisons and branches can be checked. For more information about the debugger, see rxdb on page 125).

14.8.3 Using the Trace statement

The `trace asm` statement in CREXX is designed to trace `.rxas` code as it is executed. This works only from a REXX program.

As an example, when we want to know which VM instructions are executed:

```
options levelb
# trace the world
trace asm
say 'hello trace'
say 5**2
```

[6] as opposed to assembler (rxvm) instructions

```
1C7 @ 001:003E SAY_STRING "hello trace",, * Say op1
hello trace
1C7 @ 001:0040 SAY_STRING "25",, * Say op1
25
030 @ 001:0042 RET ,, * Return VOID
```

and we see that the optimiser did its work by deciding the answer will never be anything other than 25. If we want to see how it is calculated, we need to compile without optimisation:

```
crexx hellotrace --nooptimise
```

and the resulting trace will show

```
1C7 @ 001:0041 SAY_STRING "hello trace",, * Say op1
hello trace
001 @ 001:0043 LOAD_REG_INT r4,2, * Load op1 with op2
155 @ 001:0046 IPOW_REG_INT_REG r4,5,r4 * op1=op2**op3
0E5 @ 001:004A ITOS_REG r4,, * Set register string value from its int value
1C2 @ 001:004C SAY_REG r4,, * Say op1
25
030 @ 001:004E RET ,, * Return VOID
```

15 rxlink - the cREXX Linker

`rxlink` combines one or more `.rxbin` modules into a linked image with a shared constant pool. The result is still a normal `006` format record stream and can be loaded by `rxvm`, `rxbvm`, or passed on to `rxcpack`.

15.1 Command Line Options

```
cREXX Linker
Usage: rxlink [options] input_file [input_file ...]
Options:
-o output_stem Linked output stem or .rxbin file
-c control_file Control file with INPUT/ROOT/INCLUDE/OMIT/OUTPUT/MAP/STRIP
-r root_member Root module selector (may be repeated)
-m map_file Write a simple link map
-p providers Write native-provider requirements for packaging
-l location Working location for input/output resolution
-s Strip source/TRACE debug metadata from linked output
-i Preserve inline-body metadata in linked output
-d Debug mode
-h Help
```

15.2 What It Produces

The linker writes:

1. one shared-pool record
2. one shared-backed module record for each selected module

This reduces duplication across modules while preserving the module boundaries that the virtual machine still uses for load and runtime link work.

15.3 When To Use It

The simplest workflows can stop at `rxas` and run one or more `.rxbin` files directly under `rxvm`. `rxlink` becomes useful when you want a single deployable linked image:

- to package an application root together with the modules it needs
- to reduce duplicated constant-pool data before `rxcpack`
- to produce a tidier deployable artifact than a loose set of `.rxbin` files
- to remove source/TRACE debug metadata from a deployment image

The linker does not try to replace all runtime loading. It links the modules you give it, resolves what it can within that set, and leaves other imports available for later runtime resolution if that is how the program is designed.

15.4 Module Selection

The linker can be driven from the command line or from a control file.

- `-r` selects one or more root modules directly.
- `INCLUDE` forces a module into the linked output even if it is not reached from the current roots.
- `OMIT` excludes a module from consideration.
- If no root is specified, `rxlink` selects modules containing `main`.
- If no selected module contains `main`, it falls back to the modules from the first input file.

After roots are chosen, `rxlink` follows exposed imports, `srcfprocsel` interface-factory references, and interface relationships to pull in the modules needed by those roots.

In normal application linking, explicit roots are usually enough. `INCLUDE` and `OMIT` are mainly for advanced scenarios:

- preparing a linked image that is still expected to do some dynamic or late loading
- carrying an optional provider that is not reached by the normal root graph
- excluding an alternative provider while testing or packaging a chosen variant

15.5 Selectors

Where a directive expects a module selector, `rxlink` accepts:

- the full module name
- the basename
- the source stem
- `input_path::member` for a specific member inside a multi-record input

This is especially useful when one input file contains multiple linked or concatenated modules.

15.6 Control Files

Control files are line-oriented and are useful when the link set is too large or too deliberate to keep on one command line. Blank lines are ignored, and lines beginning with * or # are treated as comments.

The supported directives are:

TABLE 1: Linker Directives.

Parm	Arg
INPUT	path
ROOT	selector
INCLUDE	selector

Parm	Arg
OMIT	selector
OUTPUT	path
MAP	path
STRIP	SOURCE

Selectors can use a module name, basename, stem, or `input_path::member` form.

The most common pattern is:

1. list one or more `INPUT` files
2. choose a `ROOT`
3. optionally ask for `STRIP SOURCE`
4. choose an `OUTPUT`

15.6.1 Minimal Application Control File

```
* app.ctl
INPUT build/app.rxbin
INPUT build/library.rxbin
ROOT app
STRIP SOURCE
OUTPUT build/app_linked
```

This tells `rxlink` to start from module `app`, pull in the providers it needs from the given inputs, strip source/TRACE debug metadata, and write `build/app_linked.rxbin`.

Run it with:

```
rxlink -c app.ctl
```

15.6.2 Advanced Include Example

```
# app_with_optional_provider.ctl
INPUT build/app.rxbin
INPUT build/providers.rxbin
ROOT app
INCLUDE providers::fallback_logger
OUTPUT build/app_with_optional_provider
```

Here `fallback_logger` is forced into the linked image even if the normal root/import walk would not select it. This is an advanced packaging tool: useful when you

know a module will be located indirectly or later than the normal static reachability analysis can see.

15.6.3 Omit One Provider Variant

```
* app_choose_provider.ctl
INPUT build/app.rxbin
INPUT build/provider_a.rxbin
INPUT build/provider_b.rxbin
ROOT app
OMIT provider_b
OUTPUT build/app_provider_a
```

This form is useful when you have alternative providers and want to make the choice explicitly at packaging time.

15.6.4 Add A Map File

```
INPUT build/app.rxbin
INPUT build/library.rxbin
ROOT app
MAP build/app_linked.map
OUTPUT build/app_linked
```

The map file records which modules were selected and which imports, if any, remain unresolved inside the current link set.

15.7 Stripping Source Metadata

`rxlink` strips inline-body metadata from linked output by default. Inline metadata is intended for library artifacts consumed by `rxc`; it is not needed after a final linked image has been built. Use `rxlink -i` to preserve it for diagnostic or tooling builds. The equivalent control-file form is:

```
PRESERVE INLINE
```

`rxlink -s` strips source/TRACE debug metadata from the linked output. The equivalent control-file form is:

```
STRIP SOURCE
```

This removes the serialized source-step records and semantic TRACE event records while preserving runtime metadata such as exposed symbols and interface/class contract data. Keep the linked image unstripped when classic TRACE, `TRACE LLM`, or RXDB source-level debugging is needed.

This option is intended for deployable `.rxbin` artifacts and for downstream `rxcpack` / wrapped images where source breadcrumbs are not needed.

15.8 Example

```
rxlink –o app linked_root.rxbin helpers.rxbin
rxvm app
```

`rxlink -o` names a linked-image stem unless the value already ends in `.rxbin`, so the first command writes `app.rxbin`.

To produce a smaller deployable linked image:

```
rxlink –s –o app_small linked_root.rxbin helpers.rxbin
```

The equivalent control-file driven form is:

```
INPUT linked_root.rxbin
INPUT helpers.rxbin
ROOT linked_root
STRIP SOURCE
OUTPUT app_small
```

16 Assembler Programming Guide

This chapter has general information about writing programs in assembly language using the `rxas` assembler programs. It discusses the assembler file format, linkage conventions, and other useful programming information, including the CREXX level B feature which allows including assembly language statements in REXX programs.

16.1 The `rxas` command

The assembler is started by invoking the `rxas` executable and takes options, which are specified before the name of the source file. For the options, see page 99. The input file has a fixed filename extension of `.rxas`, the output file replaces the extension with `rxbin`. The input is UTF-8[7] and the output is `rxbm` bytecode, which is platform independent.

16.2 The format of an assembler source file

The format of the source lines follows the general layout of assembly source, without a fixed format - there are no mandatory start columns. As most `.rxas` files

[7] except for older mainframe operating system versions, where it is EBDCIC.

will be generated by the `rxc` compiler from Rexx source, let's have a quick look at what it looks like.

```
options levelb
import rxfnsb

say "hello, rxas!"
say "Today is" date()
```

```
hello, rxas!
Today is 14 Sep 2026
```

This is the generated assembler source:

```
/*
 * SOURCE                 : hellox.crexx
 */

.globals=0

main() .locals=7
   .meta "hellox.main"="b" ".void" main() ""
   numsci 18,1,1
   .srcstep 1 1 17 "hellox.crexx" 4 1 19 "say \"hello, rxas!\""
   say "hello, rxas!"
   .srcstep 2 2 17 "hellox.crexx" 5 1 20 "say \"Today is\" date()"
   load r0,5
   settp r1,512
   settp r2,512
   settp r3,512
   settp r4,512
   settpcall r6,rxfnsb.date(),r0,r5,512
   .traceevent "F" 4 "R" "S" "r" 6 2 2 0 "date" ""
   sconcat r6,"Today is",r6
   .traceevent "O" 4 "R" "S" "r" 6 2 2 0 "" ""
   say r6
   .srcstep 3 3 1 "hellox.crexx" 5 20 20 "say \"Today is\" date()"
   ret

/* Imported Declaration from file: date@library.rxbin */

rxfnsb.date() .expose=rxfnsb.date
   .meta "rxfnsb.date"="b" ".string" rxfnsb.date() "?oformat=.string,?
      idate=.string,?iformat=.string,?osep=.string,?isep=.string"
   .meta "rxfnsb.date"=".autoload" "library"
```

Because this assembler source is generated by the compiler, it contains more than strictly necessary for a successful assembly; to be precise, support for the source-

line() function, tracing and numeric settings. We see a `main() .locals=7` in line 8. Lines 9-14 set defaults for numeric handling, while line 15 includes the sourceline from the CREXX program. Line 16 has the `say hello, rxas`; `say` is the assembler statement which assembles into an instruction for the runtime to put out this line.

When hand-written, this program would look more like:

```
main() .locals=7
  say "hello, rxas!"
  load r1,5
  call r6,date(),r1
  sconcat r6,"Today is",r6
  say r6
  ret

date() .expose=rxfnsb.date
```

We can see that the first component of the lines are the procedure names or the instruction mnemonics ; this program has no labels. After the instructions, their parameters are placed.

16.3 Register names

The CREXX Virtual Machine has a virtually unlimited number of registers. Registers are called r1..r*n. A procedure needs to specify the number of local registers it uses.

16.4 Labels

Labels are used, as in other assembly languages, as targets for branches and comparisons in the code: The label is a string that is ended by a colon (:'). Its purpose is to be a symbolic location to branch into when decisions are taken. It is recommended that a label starts in the first column of the line.[8] Branches need to have a target within a module; they cannot jump into other procedures, this is reserved for the `call` operation.

[8] it will not hurt, however, if it does not start in column one. It must be the first word of the line, though.

16.5 Procedure names

Procedure names are identifiers for routines in the program. Procedure `main()` carries a special role, as that is the one the VM runtime gives control to when a program is started. Procedures return either nothing (`.void`) or data of a specified type (like `.int` or `.string`). Procedures can be called and will return a value of the specified type.

16.6 Linkage conventions

16.7 Optimizing actions of the `rxas` assembler

`rxas` is an optimising assembler[9] which can rearrange instructions or eliminate them entirely. This optimisation can be switched off when in doubt. This will cause loss of performance, but will make debugging of the code more straightforward, because there is a more direct correspondence between the sourcelines and the generated assembly code.

TRACE output likewise describes the optimised executable: eliminated, fused, inlined or moved work may have fewer, combined or relocated trace events. Use an unoptimised compiler and assembler when source-correspondent tracing is more important than performance.

16.8 Embedded Rexx Assembler

The `assembler` command enables the inclusion of arbitrary assembler instructions within a Rexx program. The compiler validates the instruction mnemonic and the complete variable-length argument list against the opcode signature, ensuring that variables are converted into the appropriate register number. Inline assembly is therefore not limited to three operands.

This example uses the `linkarg` assembler instruction with two variables and an integer constant.

```
assembler linkarg e,i,5
```

[9] optimization is default but can be switched off, for example when debugging.

17 rxcpack - the cREXX C Packer

The C Packer program converts the `.rxbin` files into a C language structure which links together all needed modules, and a large part of the Virtual Machine infrastructure, which file then can be compiled and link edited by the C compiler. `gcc` and `clang` are the targeted compiler toolchains for Linux, macOS and Windows.

17.1 Command Line Options

```
cREXX rxcpack. Tool to convert rxbin files (or any binary file) into
a c file
that can be linked to a C exe
Version : crexx-1.0.0-beta.3+local.g4c4f4c48643e
Usage : rxcpack [options] input_file_1 input_file_2 ... input_file_n
(.rxbin is appended to input file names)
Options :
-h Help Message
-c Copyright & License Details
-v Version
-o output_file Binary Output File (.c is appended - default is input_-
file_1.c)
```

17.2 Source Code

Note that the files that are produced by `rxcpack` are valid C source code but only offer the minimal structures to link dumps of binary objects together. These are already platform dependent and can be combined with the statically linked plugin images for the target platform. The flags used for the compiler and linkage editor can be seen in the verbose output of the `crexx` compiler driver; this syntax is slightly different for `gcc` and `clang`.

17.3 The rxlink step

Though not strictly required, it is advisable to use the the `rxlink` tool in combination with the `rxpack`ed modules. It is automatically called by the `crexx` compiler driver to de-duplicate variable occurrences and select only referenced modules from libraries, shrinking the resulting native executable considerably.

18 rxdas - the disassembler

A disassembler reverses the actions of an assembler; where the assembler turns a text file containing instructions and directives into a binary executable, the disassembler returns this binary file into its text form[10] ;in this case it delivers a disassembly which in itself can be re-assembled - and still works.

18.1 Input/Output

The `rxdas` disassembler has a `.rxbin` file as input and produces a text file as output which goes to `stdout`. In this text file a disassembly has taken place; labels are synthetic and based on the combination of instructions around them. As clearly can be seen in the above, the labels generated by the CREXX compiler are not the same as the ones generated by the disassembler. With option `-p`, the constant pool of the `.rxbin` file is printed first, before the rest of the disassembly.

The current `006` `.rxbin` format is fully supported, including pooled float literals, packed code and constant sections, and the interface/class metadata directives `.class`, `.attr`, `.interface`,`.implements`, and `.member`.

Assembler-built jump tables are reconstructed as procedure-local synthetic `.jtable` and `.jcase` declarations. The disassembly records the packed algorithm actually selected (`linear`, `openhash`, or `acph`) and rewrites `jumps`, `jumpr`, `jumpn`, `jumpb`, `jumpbs`, and `jumpi` operands to the synthetic table name. Numeric tables omit the

[10] as much as possible, given the fact that some information on literals has disappeared

internal NaN compatibility entry, which is recreated by `rxas` when the disassembly is assembled again.

18.2 Command Line Arguments

When the command line argument -h is specified the options are shown:\

```
cREXX Disassembler
Version : crexx-1.0.0-beta.3+local.g4c4f4c48643e
Usage : rxdas [options] binary_file
Options :
-h Help message
-c Copyright & license details
-v Version
-p all constant Pool
-g semantic graph summary
-l location working Location (directory)
-o output_file Output file (default is stdout)
```

18.3 Example

```
/* compute sum of numbers 1 to 100 (5050) */
options levelb
/* compute sum of numbers 1 to 100000  */
sum = 0
do i=1 to 100000
   sum = i+sum
end
say "the sum of the numbers 1 to 100000 is:" sum
return
```

This program, when compiled by the `rxc' compiler, produces the following assembly source:

```
/*
 * SOURCE                 : sumLoop1000.crexx
 */
```

```
.globals=0

main() .locals=4
   .meta "sumloop1000.main"="b" ".void" main() ""
   setnumdgts 18
   setnumfuz 0
   setnumfrm 1
   setnumcas 1
   setnumstd 1
   .srcstep 1 1 17 "sumLoop1000.crexx" 4 1 8 "sum = 0"
   .meta "sumloop1000.main.sum"="b" ".int" r0
   load r0,0
   .traceevent "L" 4 "R" "I" "r" 0 1 1 0 "" ""
   .traceevent "A" 6 "R" "I" "r" 0 1 1 0 "sum" ""
   .srcstep 6 6 17 "sumLoop1000.crexx" 5 1 3 "do i=1 to 100000"
   .srcstep 2 2 17 "sumLoop1000.crexx" 5 4 7 "do i=1 to 100000"
   .meta "sumloop1000.main.i"="b" ".int" r1
   load r1,1
   .traceevent "L" 4 "R" "I" "r" 1 2 2 0 "" ""
   .traceevent "A" 6 "R" "I" "r" 1 2 2 0 "i" ""
   .srcstep 3 3 17 "sumLoop1000.crexx" 5 8 17 "do i=1 to 100000"
   load r2,100000
   .traceevent "L" 4 "R" "I" "r" 2 0 0 0 "" ""
l7dostart:
   .srcstep 3 3 17 "sumLoop1000.crexx" 5 8 17 "do i=1 to 100000"
   igt r3,r1,r2
   brt l7doend,r3
   .srcstep 5 5 17 "sumLoop1000.crexx" 6 4 15 "   sum = i+sum"
   .traceevent "V" 6 "R" "I" "r" 1 5 5 0 "i" ""
   .traceevent "V" 6 "R" "I" "r" 0 5 5 0 "sum" ""
   iadd r0,r1,r0
   .traceevent "O" 4 "R" "I" "r" 0 5 5 0 "" ""
   .traceevent "A" 6 "R" "I" "r" 0 5 5 0 "sum" ""
l7doinc:
   .srcstep 4 4 17 "sumLoop1000.crexx" 5 4 5 "do i=1 to 100000"
   inc r1
   .srcstep 7 7 17 "sumLoop1000.crexx" 7 1 4 "end"
   br l7dostart
l7doend:
   .srcstep 8 8 17 "sumLoop1000.crexx" 8 1 49 "say \"the sum of the
      numbers 1 to 100000 is:\" sum"
   itos r0
   .traceevent "V" 6 "R" "S" "r" 0 8 8 0 "sum" ""
   sconcat r3,"the sum of the numbers 1 to 100000 is:",r0
   .traceevent "O" 4 "R" "S" "r" 3 8 8 0 "" ""
   say r3
   .srcstep 9 9 17 "sumLoop1000.crexx" 9 1 7 "return"
   ret
   .meta "sumloop1000.main.sum"
```

The assembler produces an `rxbin' file from that. What follows is the disassembly from this `rxbin' file.

```
* MODULE - sumLoop1000
* DESCRIPTION - sumLoop1000

* CONSTANT POOL - Size 0x7c0.  Dump of EXPOSED entries only (option -p
    not used):


* CODE SEGMENT - Size 0x1d

.globals=0

main()          .locals=4
                .meta "sumloop1000.main"="b" ".void" main() ""
                setnumdgts 18                                   * 0x000000
                    :00f6 Set Numeric Digits digits=op1 (>4)
                setnumfuz 0                                     * 0x000002
                    :00f9 Set Numeric Fuzz digits=op1 (>=0)
                setnumfrm 1                                     * 0x000004
                    :00fc Set Numeric Form=op1 (1=sci,2=eng)
                setnumcas 1                                     * 0x000006
                    :00ff Set Numeric Case=op1 (1=lower,2=upper)
                setnumstd 1                                     * 0x000008
                    :0102 Set Numeric Standard=op1 (1=common,2=classic)
                .srcstep 1 1 17 "sumLoop1000.crexx" 4 1 8 "sum = 0"
                .meta "sumloop1000.main.sum"="b" ".int" r0
                load r0,0                                       * 0x00000a
                    :0001 Load op1 with op2
                .traceevent "L" 4 "R" "I" "r" 0 1 1 0 "" ""
                .traceevent "A" 6 "R" "I" "r" 0 1 1 0 "sum" ""
                .srcstep 6 6 17 "sumLoop1000.crexx" 5 1 3 "do i=1 to
                    100000"
                .srcstep 2 2 17 "sumLoop1000.crexx" 5 4 7 "do i=1 to
                    100000"
                .meta "sumloop1000.main.i"="b" ".int" r1
                load r1,1                                       * 0x00000d
                    :0001 Load op1 with op2
                .traceevent "L" 4 "R" "I" "r" 1 2 2 0 "" ""
                .traceevent "A" 6 "R" "I" "r" 1 2 2 0 "i" ""
                .srcstep 3 3 17 "sumLoop1000.crexx" 5 8 17 "do i=1 to
                    100000"
                load r2,100000                                  * 0x000010
                    :0001 Load op1 with op2
                .traceevent "L" 4 "R" "I" "r" 2 0 0 0 "" ""
                .srcstep 3 3 17 "sumLoop1000.crexx" 5 8 17 "do i=1 to
                    100000"
lb_13:          igtbr lb_1e,r1,r2                               * 0x000013
    :0157 Int Greater than if (op2>op3) goto op1
```

```
                .srcstep 5 5 17 "sumLoop1000.crexx" 6 4 15 "   sum = i+
                    sum"
                .traceevent "V" 6 "R" "I" "r" 1 5 5 0 "i" ""
                .traceevent "V" 6 "R" "I" "r" 0 5 5 0 "sum" ""
                iadd r0,r1,r0                                   * 0x000017
                    :000f Integer Add (op1=op2+op3)
                .traceevent "O" 4 "R" "I" "r" 0 5 5 0 "" ""
                .traceevent "A" 6 "R" "I" "r" 0 5 5 0 "sum" ""
                .srcstep 4 4 17 "sumLoop1000.crexx" 5 4 5 "do i=1 to
                    100000"
                inc1                                            * 0x00001b
                    :0020 Increment R1++ Int
                .srcstep 7 7 17 "sumLoop1000.crexx" 7 1 4 "end"
                br lb_13                                        * 0x00001c
                    :0024 Branch to op1
                .srcstep 8 8 17 "sumLoop1000.crexx" 8 1 49 "say \"the
                    sum of the numbers 1 to 100000 is:\" sum"
lb_1e:          itos r0                                         * 0x00001e
    :00e5 Set register string value from its int value
                .traceevent "V" 6 "R" "S" "r" 0 8 8 0 "sum" ""
                sconcat r3,"the sum of the numbers 1 to 100000 is:",r0
                    * 0x000020:008b String Concat with space (op1=op2||
                    op3)
                .traceevent "O" 4 "R" "S" "r" 3 8 8 0 "" ""
                say r3                                          * 0x000024
                    :01c2 Say op1
                .srcstep 9 9 17 "sumLoop1000.crexx" 9 1 7 "return"
                ret                                             * 0x000026
                    :0030 Return VOID
                .meta "sumloop1000.main.sum"
```

This file has the output of the disassembler; the procedure name, `main`, is identical, but the first label generated by the compiler, `l7dostart:` is called `lb_9` in the disassembler output. This is of no consequence for a subsequent re-assembly and execution of the program.

Do note that the lines are longer than expected, because every disassembled line has the binary representation and a standard explanation of the generated instruction. This makes it easier to find back instructions in an `.rxbin` load module.

The instructions, however, can be different than expected due to the optimizations the assembler performs. When the compiler has performed optimizations, this is already visible in the `.rxas` file. The assembler optimizations are visible in the disassembled object.

The standard instruction documentation the disassembler affixes is the same as

displayed by the `rxas -i` command, in a line comment after the instructions and their operands.

When stepping through a program using the CREXX Debugger (which is mentioned in the next chapter), the disassembly is the most representative record of what is in the `.rxbin` executable.

19 rxdb - the CREXX Debugger

The debugger is one of the programs in the toolchain delivered with CREXX as its source code; most other programs, at the moment, are compiled from C. It is easily adaptable and can be regarded a *debugger construction set*. By adapting and recompiling the user can implement their own wishes for a debugger. In this sense, it can be seen as an open-ended complement to the REXX `trace` statement. Because it has modes for REXX as well as `rxas` Assembler, it is a useful tool for debugging high-level as well as low-level problems.

19.1 Command Line Options

```
RXDB - REXX Debugger
Usage: rxdb [text|plain|llm]
```

19.2 Runtime Options

After the `rxdb` program is started, a few runtime options appear in the delivered version. This is an example session:

PART III

Plugins

20

cRexx /PA - Plugin Architecture

This facility allows for the compilation, linking, and execution of additional functionality (developed in C) alongside Rexx code. It enables the decoupling of native modules (plugins), which can be developed and packaged either as separate entities or linked statically to the main crexx core solution.

The architecture is designed to decouple dynamic plugins from the internal crexx libraries. The installed development surface starts at `<rxpa/crexxpa.h>` and includes its public generated version and integer-type headers through the `CREXX::RXPA` CMake target.

Plugin developers have the flexibility to structure their code in a way that allows for both dynamic or static linking. While the source code can remain the same, it needs to be built differently based on the desired linking approach, as it relies on macro expansion.

During the Rexx compilation process, the Rexx compiler inspects the plugin to determine the type and argument of any provided functions. This inspection helps ensure type safety. It is recommended for plugin developers to load or initialise dependencies only when an explicitly exposed initialisation Rexx function is called or when the first function is invoked (lazy initialisation) to avoid overhead during build (rather than run). In addition, for the static builds, there is macro definition **DECL_ONLY** which allows definition / implementation code to be excluded from the static library designed to be linked to the compiler only.

The compiler discovers dynamically packaged declarations (`*.rxplugin`) on its binary import roots. A used native declaration records its stable provider ID in the compiled module. At runtime, a matching static provider is preferred; otherwise `rxvm` opens the trusted `<provider-id>.rxplugin` and verifies the binary's manifest ID and function signature before linking it.

The Plugin Architecture offers a comprehensive set of resources for developers, including a header file, macros, and a cmake configuration. These tools aim to create a convenient and efficient development environment for plugin creation.

20.1 Dynamic Plugins Recommended

User-provided plugins are recommended to be provided as dynamic `.rxplugin` files. Dynamic plugins offer several advantages: easy site-wide distribution by placing them in the same directory as CREXX binaries, project-specific customization by locating them in the project REXX files directory, and flexible placement in any desired location using `rxc` and `rxvm` options.

In contrast, static plugin packaging is more complex in terms of linking and is intended for core CREXX components that are part of every CREXX release. Static plugins are shipped within the CREXX binaries, ensuring their availability and consistency across distributions.

A supported provider can publish dynamic and static forms as one package:

```
add_dynamic_plugin_target(_example example.c)
add_static_plugin_target(_example example.c)
add_rxpa_provider_package(_example)
```

The helper places the dynamic artifact, canonical static archive (`<provider-id>.a` or `.lib`), and compatibility `_static` archive under `bin/providers`. Ordinary `rxvm` execution then needs no `-p`, and `crexx -native` selects and retains the static archive automatically from the compiled dependency record. Application-local providers may instead be placed in a `providers` directory beside the main `.rxbin`; explicit trusted directories use `rxvm --provider-path` or `CREXX_PROVIDER_PATH`.

The choice between dynamic and static plugin packaging depends on the specific requirements and use cases: dynamic plugins provide flexibility and customization for users, while static plugins are designed to provide a robust solution for core CREXX components.

20.2 Example Plugin

The following code demonstrates a decryption plugin.

- The PROCEDURE macro starts the function, and argument access is via macros like ARG().
- Each argument is a CREXX register. This means it holds multiple types or values and these are accesses by macros like GETSTRING() and SETSTRING().
- Errors are handled by defining and returning a SIGNAL via the RETURNSIGNAL macro.

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/* Name: rxdes.C                                                      */
/*                                                                    */
/* Copyright René Jansen june 1st 1993                                */
/*                                                                    */
/* Function: crexx external functions RxDesEncrypt and RxDesDecrypt */
/*                                                                    */
/* dependencies/calls: desbase.c CREXX/rxpa                           */
/*                                                                    */
/* Ported to CREXX by Adrian Sutherland 2024                          */
/*--------------------------------------------------------------------*/

#include <stdio.h>
#include <string.h>
#include <rxpa/crexxpa.h> // cRexx/PA - installed Plugin Architecture
    header
#include "desbase.h"    // DES encryption/decryption functions

// Encrypt a string using DES - the first argument is the key, the
    second the data
PROCEDURE(RxDesEncrypt)
{
    char   des_out[8];
    char   des_in[8];
    char   key[8];
    char   result[17];

    if( NUM_ARGS != 2)
        RETURNSIGNAL(SIGNAL_INVALID_ARGUMENTS, "2 arguments expected")

    if (strlen(GETSTRING(ARG(0))) != 16 || strlen(GETSTRING(ARG(1))) !=
         16)
        RETURNSIGNAL(SIGNAL_INVALID_ARGUMENTS, "Both arguments must be
            16 hex digits")
```

```
    if( hex2bin(GETSTRING(ARG(0)), key) < 0)
        RETURNSIGNAL(SIGNAL_INVALID_ARGUMENTS, "Key is not valid hex")

    if( hex2bin(GETSTRING(ARG(1)), des_in) < 0)
        RETURNSIGNAL(SIGNAL_INVALID_ARGUMENTS, "Data is not valid hex")

    // Encrypt
    desinit(key);
    endes(des_in, des_out);

    // Set return and make sure the signal is reset/ok
    bin2hex(des_out, result);
    SETSTRING(RETURN, result);
    RESETSIGNAL
}

// Decrypt a string using DES - the first argument is the key, the
   second the data
PROCEDURE(RxDesDecrypt)
{
    char   des_out[8];
    char   des_in[8];
    char   key[8];
    char   result[17];

    if( NUM_ARGS != 2)
        RETURNSIGNAL(SIGNAL_INVALID_ARGUMENTS, "2 arguments expected")

    if (strlen(GETSTRING(ARG(0))) != 16 || strlen(GETSTRING(ARG(1))) !=
         16)
        RETURNSIGNAL(SIGNAL_INVALID_ARGUMENTS, "Both arguments must be
            16 hex digits")

    if( hex2bin(GETSTRING(ARG(0)), key) < 0)
        RETURNSIGNAL(SIGNAL_INVALID_ARGUMENTS, "Key is not valid hex")

    if( hex2bin(GETSTRING(ARG(1)), des_in) < 0)
        RETURNSIGNAL(SIGNAL_INVALID_ARGUMENTS, "Data is not valid hex")

    // Decrypt
    desinit(key);
    dedes(des_in, des_out);

    // Set return and make sure the signal is reset/ok
    bin2hex(des_out, result);
    SETSTRING(RETURN, result);
    RESETSIGNAL
}
```

```
// Functions to be provided to rexx - these are loaded either when the
    plugin is loaded (dynamic) or
// before main() is called (static)
LOADFUNCS
//      C Function__, REXX namespace & name, Option_, Return Type_,
    Arguments
ADDPROC(RxDesDecrypt, "rxdes.decrypt",       "b",     ".string",    "
    key=.string,data=.string");
ADDPROC(RxDesEncrypt, "rxdes.encrypt",       "b",     ".string",    "
    key=.string,data=.string");
ENDLOADFUNCS

// End of file
```

The following shows how the defined functions are published to cRexx.

```
// Functions to be provided to rexx \- these are loaded either when the
    *
// plugin is loaded (dynamic) or before main() is called (static)*
LOADFUNCS
//      C Function\_\_, REXX namespace & name, Option\_, Return Type\_,
    *
// Arguments*
ADDPROC (RxDesDecrypt, "rxdes.decrypt",       "b",     ".string",
   "key=.string,data=.string");
ADDPROC (RxDesEncrypt, "rxdes.encrypt",       "b",     ".string",
   "key=.string,data=.string");
ENDLOADFUNCS**
```

The ADDPROC Macro published the function to cRexx. This has the namespace and name, options/level (should be b), the function return type (in level b format) and the arguments (again in level b format). This allows the Compiler and VM to search and ``fix-up'' procedure calls using the same search order and syntax as for procedures developed in Rexx.

The return and argument strings are Level B declarations, not comma-separated type lists. Each argument therefore needs its source-level name and declaration, for example `left=.int,right=.int`; `.int,.int` is invalid. Empty components, malformed separators, and malformed return or argument type syntax are rejected at the importing CREXX call site with `RXPA_IMPORT_SIGNATURE_INVALID`. The diagnostic identifies the plugin, routine, failing field, and declaration.

20.3 Concurrency and per-VM sessions

An existing plugin needs no source change. A current host treats it as legacy: one VM uses the direct adapter, while concurrent legacy-capable VMs use a process-wide recursive compatibility lane. This is the safe default because the host cannot infer whether plugin statics and native dependencies are thread-safe.

After auditing every published procedure, writable static, dependency and cleanup path, a plugin that permits concurrent entry can add one file-scope declaration:

```
RXPA_PLUGIN_PROCESS_REENTRANT
```

This is an assertion about defined concurrent behavior, not an assertion that the functions have no side effects. The optional capability symbol leaves the existing initializer and call ABI unchanged, and older hosts ignore it.

For a mixed plugin, use `RXPA_PLUGIN_PROCEDURE_CAPABILITIES(query)` and return `RXPA_PROCEDURE_CAP_PROCESS_REENTRANT` only for audited procedures. Return `0` for procedures that must remain on the legacy lane.

A plugin that owns a connection, device, interpreter or similar mutable native resource per VM can use:

```
RXPA_PLUGIN_SESSION_AWARE(create_session, destroy_session,
                          enter_session, leave_session,
                          procedure_capabilities)
```

The query returns `RXPA_PROCEDURE_CAP_SESSION_AFFINE` for procedures that use the per-VM session. The host creates one session when that VM loads the plugin, preselects the call adapter, and destroys the session before unloading plugin code. `enter_session` receives the selected session and must return the previous thread-local session as a cookie; `leave_session` restores that cookie so nested plugin calls work correctly. A failed factory fails the plugin load and rolls back the DSO and any already-created sessions.

V2-aware hosts fail closed: malformed manifests, unknown or combined flags, and a session-affine procedure without all four session hooks become legacy. Older hosts call the unchanged procedures and never invoke the V2 hooks. If such hosts must remain supported, the plugin procedures must fall back to a plugin-owned default session when no per-VM session is active. Do not retain RXPA register handles or VM-owned values in either kind of session.

20.4 Macros

The following macros are provided for plugin developers (defined in `rxpa/crexxpa.h`).

Macro	Purpose
LOADFUNCS	Starts and finishes the block of ADDFUNC()'s
ADDFUNC()	Published a function to CREXX
NUM_ARGS	Is the number of arguments passed the the function
ARG()	Returns the nth argument (which is an opaque pointer to the CREXX register)
RETURN	Returns the register used to pass the function's returned value.
GETSTRING()	Gets the String value of a register
SETSTRING()	Sets the String value of a register
GETINT()	Gets the Integer value of a register
SETINT()	Sets the Integer value of a register
GETFLOAT()	Gets the float (double) value of a register
SETFLOAT()	Sets the float (double) value of a register
CALLMETHOD()	Synchronously invokes a descriptor-checked method on a live CREXX object and uses the current SIGNAL register.
CALLMETHODX()	The callback form of CALLMETHOD with an explicit signal register.
SIGNAL	Returns the registers used to pass and Signal Information.
RESETSIGNAL	Ensures that the signal register is set no ``no signal''
RETURNSIGNAL()	Sets the signal register and returns from the function. Used for error conditions.
ENDLOADFUNCS	Starts and finishes the block of ADDFUNC()'s
RXPA_PLUGIN_-PROCESS_-REENTRANT	Asserts that every published procedure permits concurrent process entry.
RXPA_PLUGIN_-PROCEDURE_-CAPABILITIES()	Supplies a load-time per-procedure reentrant/legacy policy query.
RXPA_PLUGIN_-SESSION_-AWARE()	Supplies the per-procedure query and per-VM session lifecycle/entry hooks.

`SETSTRING`, `RETURNSTR`, and the detail text passed to `RETURNSIGNAL` accept `const char *`. The VM copies the null-terminated bytes into register-owned storage during the call; it does not mutate or retain the caller's buffer. The caller therefore retains ownership and may reuse or release a mutable source buffer as soon as the macro returns. Pass a non-null, null-terminated string; use `""` for an empty value.

`CALLMETHOD(receiver, descriptor, argc, args, result)` enables a native pro-

cedure to call back into an object supplied by cREXX. The descriptor uses the canonical `rxsig1|name|return_type|arguments` form. `args` is an array of borrowed `rxpa_attribute_value` handles, and `result` is another borrowed handle that receives the method return value. The function returns zero after a successful invocation; on setup or resolution failure it returns non-zero and populates `SIGNAL`. An unhandled signal raised by the called method is likewise returned through that signal handle. `CALLMETHODX` takes the same arguments plus an explicit signal handle, which is useful from a C library callback outside a `PROCEDURE` body.

The invocation is synchronous on the current VM thread. Receiver and argument mutations are copied back before it returns, and the called method may make nested RXPA calls. Never store any of these handles in plugin or session state beyond the lifetime of the active outer RXPA call. Because `callmethod` is an appended pre-release initializer callback and `rxpa_initctx` has no negotiated size, plugins using it must be rebuilt together with the host.

The Signal values are:

Signal	Purpose
SIGNAL_NONE	
SIGNAL_ERROR	Syntax error
SIGNAL_- CONVERSION_- ERROR	conversion error between types
SIGNAL_- UNKNOWN_- INSTRUCTION	unknown RSAS instruction
SIGNAL_- FUNCTION_- NOT_FOUND	attempt to execute an unknown function
SIGNAL_OUT_- OF_RANGE	attempt to access an array element that is out of range
SIGNAL_FAILURE	error in an external function or subroutine
SIGNAL_HALT	an external request to halt execution
SIGNAL_- NOTREADY	IO error, such as a file not being ready
SIGNAL_- INVALID_- ARGUMENTS	invalid arguments are passed to a function
SIGNAL_OTHER	Other (or unknown) error condition

In addition

- **BUILD_DLL** is defined if the dynamic (rather than static) version of the plugin is being built. Developers may check for this macro.

- **DECL_ONLY** is designed for static builds linked to the compiler, enabling plugin developers to package a smaller library with declaration support only. The build process would create a separate static library with full functionality separately to link with rxvm. This facility is not provided for dynamic plugins to prevent confusion between declaration-only and full-function files.

20.5 Build Script

Install CREXX to a development prefix, then consume its installed CMake package. The project must not copy `crexxpa.h`, generated headers, or `RXPluginFunction.cmake` from a CREXX source or build tree.

```
cmake_minimum_required(VERSION 3.24)
project(rxdes C)

# The package version is the core semantic version.
    CREXX_VERSION_STRING below
# contains the exact release/prerelease/build identity.
find_package(CREXX 1.0.0 EXACT CONFIG REQUIRED)

if(NOT CREXX_VERSION_STRING STREQUAL "1.0.0-beta.3")
    message(FATAL_ERROR
            "This plugin requires cRexx 1.0.0-beta.3; found ${
                CREXX_VERSION_STRING}")
endif()

# Optional; the default for an external project is <build>/bin.
set(CREXX_RXPA_PLUGIN_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/plugin")
add_dynamic_plugin_target(des rxdes.c desbase.c desbase.h)
```

Test or replacement implementations that intentionally publish an existing stable provider identity under a different CMake target may pass `PROVIDER_ID`, for example `add_dynamic_plugin_target(des_mock PROVIDER_ID rxdes mock.c)`. The output artifact must still use the matching canonical provider stem; this option does not create a runtime alias.

Configure with the selected scratch or system prefix explicitly:

```
cmake -S . -B build -DCMAKE_PREFIX_PATH=/path/to/crexx-prefix
cmake --build build --target des --verbose
```

`find_package(CREXX CONFIG)` provides:

- the header-only `CREXX::RXPA` target and the `add_dynamic_plugin_target()` helper;
- imported executable targets such as `CREXX::rxc`, `CREXX::rxas`, `CREXX::crexx-contract`, product `CREXX::rxvm`, concrete `CREXX::rxbvm`, and optional concrete `CREXX::rxtvm`, with corresponding `CREXX_<tool>_EXECUTABLE` path variables (`CREXX_rxtvm_FOUND` is false when the installed compiler could not build direct threading);
- the `crexx_add_operation_contract()` helper for the build-time, JSON contract surface documented in Operation Contracts;
- `CREXX_IMPORT_DIR`, `CREXX_PLUGIN_DIR`, and `CREXX_RUNTIME_DIR`, which name the installed locations. A project-local plugin directory must still be supplied explicitly;
- `CREXX_VERSION_STRING`, `CREXX_VERSION_DISPLAY`, `CREXX_PACKAGE_PREFIX`, and `CREXX_BUILDINFO_FILE` for machine-readable identity checks.

The helper builds the platform-appropriate module with the `.rxplugin` suffix on Windows, macOS, and Linux. Dynamic plugins receive the RXPA callback table from the VM and do not link a private CREXX archive.

20.6 Static Builds

The CREXX Cmake build configuration should be referenced for static build support. Building the static library is simple, but proper linking is crucial to guarantee that the linker recognizes the need to link in the plugin and that the plugin's initialization function is called at program load across various platforms.

20.7 Execution from Rexx

The following is a REXX Level B example using the plugin.

There is no manual loading of a declaratively packaged dynamic or static provider. The REXX program does not need to distinguish the delivery form, and no REXX wrapper is required: RXPA publishes the typed

```
options levelb
import rxfnsb
import rxdes    /* Import the rxdes plugin functions */

/* Note that the input and output to the des functions are in hex
    strings */
Plaintext = "0000000000000000"
key =       "08192A3B4C5D6E7F"

Ciphertext = Encrypt(key,Plaintext)
```

In line 4, the import statement loads the namespace meaning that any Rexx modules or native plugins in the rxdes namespace will be loaded as needed; the function call, encrypt(), follows Rexx syntax, and the compiler can check parameters and return types as normal.

The compiler and VM paths are separate and explicit. For a plugin produced as `/path/to/plugin/rxdes.rxplugin` and a crexx installation at `/path/to/crexx-prefix`, a complete optimized build and run is:

```
/path/to/crexx-prefix/bin/rxc \
  -i /path/to/plugin -i /path/to/crexx-prefix/bin \
  -o execdes execdes.crexx
/path/to/crexx-prefix/bin/rxas -o execdes.rxbin execdes.rxas
/path/to/crexx-prefix/bin/rxvm \
  execdes.rxbin /path/to/plugin/rxdes \
  /path/to/crexx-prefix/bin/library
```

Use `-n` with both `rxc` and `rxas` for a non-optimized build. Omitting the plugin directory from the compiler import path makes its declarations unavailable; omitting the plugin module path from the VM invocation prevents the implementation from loading. The `.rxplugin` and `.rxbin` suffixes may be omitted from VM module arguments, but the directory must remain unambiguous.

PART IV

Appendices

A

Building the Toolchain

Most users will download and install a binary distribution for their platforms and will not need this information. In some cases when a binary distribution is unavailable, it will be beneficial to be able to build the CREXX system using a standard C toolchain.

This chapter aims to show all you need to know about how to build CREXX from scratch, and then run it. It will show what you need, what to do and when it is build, how to make working programs with it.

A.1 Requirements

Currently CREXX builds on Windows, macOS and Linux[11].

You need one of those and:

TABLE 2: Required tools.

Tool	Function
git	Source code versioning
CMake	Build Tool
gcc, clang or MSVC[12]	C compiler
Make	conventional build tool, or
Ninja	fast build tool

[11] There is a separate instruction for VM/370 (and later) mainframe operating systems.
[12] clang is a working equivalent and the default on macOS.

A.2 Platform specific info

On Linux and macOS, this instruction is identical. For macOS, Xcode batch tools need to be installed, which will provide you with git, make and the compiler. Brew will give easy access to Cmake and Ninja-build[13].

On Windows, MSYS2 or WSL remain useful when a GNU-compatible environment is preferred. A native x64 build is also supported with the Visual Studio 2022 C toolchain. Run CMake from an x64 Visual Studio Developer Command Prompt (or after calling `VsDevCmd.bat`) so `cl.exe`, the Windows SDK and the linker are available:

```
cmake -S CREXX -B crexx-build-msvc -G Ninja ^
  -DCMAKE_BUILD_TYPE=Release -DENABLE_PARSER_MODE=OFF
cmake --build crexx-build-msvc --parallel 8
```

MSVC builds the portable switch-dispatch VM and copies `rxbvm.exe` to the stable `rxvm.exe` product path. The optional parser-mode/syntax-highlighting integration currently depends on POSIX DSLSH sources and is therefore disabled in the native MSVC command above. This does not disable normal `rxc` compiler operation. Run `crexx -native` from the same Developer Command Prompt; the driver uses the compiler family and runtime-library mode recorded by CMake.

A.3 Process

Here it is assumed that all tools are installed and working, and available on your PATH environment variable.

Choose or make a suitably named directory on your system to contain the source code. Note that the CREXX source is kept in a different directory on you system than where it is built in, or will run from. Now run this command:

```
git clone https://github.com/adesutherland/CREXX.git
```

This will give you a CREXX subdirectory in the current directory, containing the source of CREXX and its dependencies. This is the `develop' branch, which is the one you would normally want to use. All The sources use C90 or C99 where suffi-

[13] For Linux, you will need to install git (which will be there on most distributions), cmake and gcc or clang. For installing the openssl development headers if needed: sudo apt update; sudo apt install libssl-dev pkg-config}

cient. The concurrency-enabled interpreter requires C11 atomics, which are supported by the documented GCC, Clang and Visual Studio 2022 toolchains.

Make a new subdirectory in the current directory (not in CREXX, but in the one that contains it), like `crexx-build'.

```
mkdir crexx-build
```

and cd into that directory. Now issue the following command (we assume that you installed ninja, otherwise substitute `make' for the two <!-- instances ofninja'): -->

```
cmake -S ../CREXX -B crexx-build -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build crexx-build --parallel 5
cmake --build crexx-build --target qa-comprehensive
```

This will do a lot of things. In fact, if all goes well, you will have a built and tested CREXX system.

A.4 Explaining the build process

Let's zoom in a little on what we did. The first step is to tell CMake to validate your system environment and generate a build script (and a test script) for the chosen build tool. Cmake will read the file CmakeLists.txt and validate that your system can do what it asks it to do. This can yield error messages, for example if the C compiler lacks certain functions or header files. (When that happens, open an issue and someone will have a look at it - or peruse stack overflow which is what we probably also will do).

After CMake has successfully validated the build environment, it will generate a build script (a Makefile in the case of Make and a `build.ninja` file in the case of Ninja). This is specified after the -G flag. The -DCMAKE_BUILD_TYPE=Release flag makes sure we do an optimized build, which means we specify an -O3 flag to the C compiler, which then will spend some time optimizing the executable modules, which makes them run faster (they do!). The alternative is a `debug' build which will yield slower executables, but with more debugging information in them.

The two ampersands (&&) mean we do the next part only if the previous step was successful. This is a `ninja` statement, which will build everything in the

`build.ninja` specification file. These are a lot of parts, and the good news is, when they are built once, only the changed source will be built, which will be fast.

Ninja builds also use named resource pools for work whose memory pressure is not represented by the global `--parallel` value. Configure with `-DCREXX_BUILD_RESOURCE_PROFILE=developer-fast` on a capable development machine, `portable` on slower or unknown hosts, or `memory-constrained` when RAM is tight. `auto` is the default: it selects `developer-fast` on Apple ARM64 and `portable` elsewhere. These profiles currently limit concurrent VM-core C compiles and native links while leaving independent graph work free to use the global job count. The limits can be overridden explicitly with `CREXX_VM_COMPILE_POOL_DEPTH` and `CREXX_NATIVE_LINK_POOL_DEPTH`; both must be positive integers.

For example, Adrian's macOS development profile still permits 30 runnable graph actions while limiting the high-memory VM compile family separately:

```
cmake -S . -B cmake-build-debug -G Ninja \
  -DCMAKE_BUILD_TYPE=Debug \
  -DCREXX_BUILD_RESOURCE_PROFILE=developer-fast
cmake --build cmake-build-debug --parallel 30
```

On an unknown or smaller host, select `portable` and use five global jobs. Non-Ninja generators retain their own scheduling because CMake job pools are a Ninja facility.

Production self-builds do not use the compiler's normal broad discovery defaults. CMake copies each action's declared RXBIN and RXPA metadata providers to a private import root and invokes `rxc --no-exe-import`. Where a source directory contains sibling modules or test fixtures, the primary source is also copied to an action-private source root. This is necessary because `rxc` normally considers sibling source before RXAS/RXBIN metadata and also appends its executable directory to the binary roots. Those defaults remain useful for interactive compilation, but would allow an older linked library or an unrelated test plugin to affect a clean or incremental self-build.

The build declarations, rather than filesystem timestamps, therefore select the metadata route. When adding a production `rxc` action, name its exact provider files in `DEPENDS`, stage only those files with `crexx_add_import_root`, isolate the source when it has siblings, and keep the resolution report as an opt-in diagnostic rather than a routine side effect.

After the product build, select a named QA target such as `qa-smoke` or `qa-comprehensive`. Each target first builds the generated artifacts for its own test tier and then starts CTest. CTest executes tests only; it does not start another build. This knows what to do, as the tests were defined in the Cmake recipes, and will show you successes and failures. If what you checked out if git is not a released version, there is a change that some test cases fail, but generally these should indicate success.

A.4.1 Running QA from CLion

Do not use CLion's automatically generated `All CTest` configuration for a normal CREXX check. It selects independently prepared correctness, qualification, stress and measurement tiers together.

The simplest CLion workflow is to select the `Release` CMake profile and build one of the named `qa-*` CMake targets. Use the CMake target selector or the CMake tool window and choose Build (the hammer), not Run:

- build `qa-smoke` for the normal quick development check;
- build `qa-comprehensive` for the broad correctness check before submitting a substantial change; or
- build `qa-qualification`, `qa-optimizer-parity`, `qa-stress` or `qa-measurement` only when that separate assurance lane is required.

Each `qa-*` target first builds its matching `qa-prep-*` dependency closure and then invokes CTest with the right labels and parallelism. The results appear in CLion's Build panel. Moving from smoke to comprehensive may build additional test harnesses and generated fixtures the first time; unchanged dependencies are incremental and should subsequently be no-ops.

To use CLion's structured Test Runner instead, keep preparation and execution as two explicit steps:

1. Select the `Release` CMake profile.
2. Build the matching `qa-prep-*` target from the CMake target selector or CMake tool window.
3. Open **Run | Edit Configurations**, add a local **CTest Application**, and give it the tier name.

4. Set its working directory by browsing to this project's `cmake-build-release` directory. Do not depend on a shared or copied machine-specific path.
5. Remove or disable its **Build** before-launch task because preparation was completed explicitly in step 2.
6. Enter the matching CTest arguments from the table below, save the local configuration, and choose Run.

Local CTest configuration	Preparation target	CTest arguments

Essential	`qa-prep-essential`	`--parallel 30`
		`--output-on-failure`
		`--label-regex ^essential$`

Smoke	`qa-prep-smoke`	`--parallel 30`
		`--output-on-failure`
		`--label-regex`
		`^(essential|smoke)$`

Comprehensive	`qa-prep-comprehensive`	`--parallel 30`
		`--output-on-failure`
		`--label-regex`
		`^(essential|smoke|comprehensive)$`

Qualification	`qa-prep-qualification`	`--parallel 30`
		`--output-on-failure`
		`--label-regex ^qualification$`

Optimizer parity	`qa-prep-optimizer-parity`	`--parallel 30`
		`--output-on-failure`
		`--label-regex`
		`^optimizer-parity$`
		`--label-exclude`
		`^performance-measurement$`

Stress	`qa-prep-stress`	`--parallel 30`
		`--output-on-failure`
		`--label-regex ^stress$`

Measurement	`qa-prep-measurement`	`--parallel 1`
		`--output-on-failure`
		`--label-regex`
		`^performance-measurement$`

Run measurement only on a quiescent host. After changing source, build inputs, or test tier, repeat the matching preparation step before starting CTest. The equivalent commands can always be run in CLion's Terminal; for example:

```
cmake --build cmake-build-release --target qa-prep-smoke --parallel 30
ctest --test-dir cmake-build-release --parallel 30 --output-on-failure
    \
```

```
--label-regex '^(essential|smoke)$'
```

The default product build is deliberately offline. Parser mode uses a local sibling `DSL-Syntax-Highlighter` checkout when it exists; otherwise parser mode defaults off. To permit CMake to fetch the pinned parser dependency, configure with `-DCREXX_ALLOW_NETWORK_DOWNLOADS=ON`. An explicit `-DENABLE_PARSER_MODE=ON` without either a local checkout or that network permission is a configuration error.

`rxsqlite` is a normal offline core component with a source-controlled SQLite amalgamation. Its ADDRESS demo is an ordinary Level G consumer of the installed façade; neither needs a SQLite build option or network access.

Standalone examples and demonstrations are not members of the default product build. Request the documented auxiliary groups explicitly:

```
cmake --build crexx-build --target crexx-examples --parallel 5
cmake --build crexx-build --target crexx-demos --parallel 5
```

Comprehensive QA still prepares the example and demonstration artifacts that its tests consume through `qa-prep-comprehensive`; some source examples are therefore also visible as QA fixture inputs. Measurement preparation is separate and does not execute a workload. Contributions and experiments are not currently configured as product targets. New ones should remain explicit opt-ins rather than joining the default product implicitly.

A.5 QA tiers and useful system test subsets

Use the workflow that matches the work being changed:

Developer	Normal build and close-out route
REXX user	Install an optimized Release archive/package and run `crexx program`.
REXX program/library developer	Use plain `crexx` for compile-and-run work, or installed Release `crexx --program` and `crexx --library` for maintained incremental products; prove clean, immediate no-op and changed-source behavior without requiring CMake.

Developer	Normal build and close-out route
plugin developer	Use the installed Release SDK from an external CMake project; close with the dynamic-plugin consumer, install/autoload checks and relevant sanitizer scope.
core developer	Configure a source Debug tree, build the affected target, run focused tests or `qa-smoke`, then run `qa-optimizer-parity` and `qa-comprehensive` when compiler/optimizer behavior is relevant.
release/QA maintainer	Build a clean Release product, install/package it, and run comprehensive, qualification, separate stress, sanitizer, CodeQL and supported-platform gates for the exact SHA.

`CMAKE_BUILD_TYPE` controls native C/C++ optimization. The `rxc` optimizer is a separate axis: Release is the normal installed/user product, while core Debug work still proves optimized and non-optimized REXX equivalence. Debug sanitizer builds and MinSizeRel size experiments are assurance products, not substitutes for the optimized Release binary supplied for user testing.

Every configured test has exactly one execution tier while retaining its topical labels. The named targets make the intended barriers visible:

Target	Purpose
`qa-essential`	Smallest correctness blockers
`qa-smoke`	Essential plus quick representative coverage
`qa-comprehensive`	Normal correctness sweep, excluding stress and measurement
`qa-qualification`	Install, packaging, reproducibility and external-consumer proof
`qa-stress`	Explicit high-load and race-oriented workloads
`qa-measurement`	Performance measurement only, serially on a quiescent host
`qa-optimizer-parity`	Focused optimized/non-optimized runtime matrices, excluding performance measurement

Each target depends on its matching `qa-prep-*` closure. Comprehensive prepara-

tion includes smoke and essential; qualification and stress remain independent. The compatibility `qa-prep` target combines all non-measurement closures for older scripts. `qa-measurement` uses `qa-prep-measurement` and always selects one CTest worker. Tests carrying the topical `performance` label are always normalized to this serial measurement tier, even if an older declaration also called them smoke. Do not combine performance measurement with a busy correctness or stress worker pool; timings from an active host are only indicative.

The named correctness targets use 30 CTest workers by default on Apple ARM64 and five elsewhere. Override that independently of build parallelism with `-DCREXX_QA_CTEST_JOBS=<positive-number>`.

Hosted pull-request and `develop` builds use the optimized Release product path, run `qa-smoke`, then upload one archive per supported platform. The archive name and its `BUILDINFO` contain the exact PR-head or `develop` SHA. It is a user-test candidate, not a qualified or signed release.

The other hosted lanes remain independent of artifact availability. Linux Debug optimizer parity runs for PRs and `develop`. After `develop` changes, the next scheduled deep QA run performs comprehensive and install/package qualification on every supported platform, builds the Release graph at jobs 1, 5 and 30, compares RXBIN manifests, checks an immediate no-op and Ninja generated-file dependencies, and runs stress separately. The scheduled sanitizer workflow similarly runs Linux ASan/LSan and macOS ASan only when the current `develop` SHA has not already passed. Failed scheduled runs retry the same SHA on the next night; explicit dispatches and version-tag sanitizer runs are unconditional. GitHub starts schedules from the repository's default branch, so the scheduled workflows explicitly resolve and check out `develop`; changes to the schedule itself become active when that workflow revision reaches the default branch. CodeQL retains its own workflow. Performance measurement remains an explicit `qa-measurement` run on a quiescent host, not a hosted parallel timing claim.

For routine system validation, run the product build and normal correctness suite:

```
cmake --build cmake-build-debug --parallel 5
cmake --build cmake-build-debug --target qa-comprehensive --parallel 5
```

For a direct CTest invocation, prepare the same tier first and use its exact labels:

```
cmake --build cmake-build-debug --target qa-prep-comprehensive --
    parallel 5
ctest --test-dir cmake-build-debug \
```

```
  --label-regex '^(essential|smoke|comprehensive)$' \
  --output-on-failure --parallel 5
```

For standard-library and BIF work, useful focused checks are:

```
cmake --build cmake-build-debug --target testbifs
ctest --test-dir cmake-build-debug -R '^ts_.*_(noopt|opt)$' --output-on
    -failure
```

If a BIF source change under `lib/rxfnsb/rexx/` causes compiler RXAS golden tests to fail but the corresponding runtime tests still pass, rebuild the linked standard-library image before judging the golden diff:

```
cmake --build cmake-build-debug --target library
cmake --build cmake-build-debug --target testbifs
```

Then rerun the focused compiler/runtime tests and review the generated RXAS:

```
ctest --test-dir cmake-build-debug/compiler/tests -R '13_stems' --
    output-on-failure
ctest --test-dir cmake-build-debug -R '^ts_stem_(noopt|opt)$' --output-
    on-failure
git diff -- compiler/tests/golden
```

Consumer RXAS import blocks are a snapshot of the callables needed for linking/runtime lookup, not a copy of the full provider API. If the diff only adds or removes unused imported declarations, update goldens only after confirming that the consumer still imports the callables it actually calls. The maintainer testing details are in `compiler/docs/testing.md`.

The `lib/rxfnsb/rexx` BIF build is a bootstrap build. It compiles most BIF source files with compiler exits disabled (`rxc -x`), so explicit certified exits such as `TRACE`, `PARSE`, and `ADDRESS` are not available inside those library source files during the BIF build. To debug a BIF, call it from a normal test fixture or scratch program compiled with exits enabled. Add `TRACE UNSUPPRESS NAMESPACE rxfnsb` when you need library frames in the trace.

The native `system` plugin has its own smoke test:

```
ctest --test-dir cmake-build-debug -R '^test_system$' --output-on-
    failure
```

For TRACE/debug metadata work, combine focused TRACE and linker checks before the full suite:

```
ctest --test-dir cmake-build-debug \
```

```
-R '^(trace_event_metadata|test_trace_|ts_trace_|rxlink_format_check|
    rxlink_rxdas_strip_smoke)' \
--output-on-failure
```

A.6 Build version and timestamp

The project version is read from the top-level `VERSION` file. To change the CREXX version number, edit that file and use a semantic version such as `1.0.0`, `1.0.0-beta.3`, or `1.0.0-beta.3+build.7`.

Local, non-release builds also include build metadata in the displayed version: the build channel, the short Git commit id when Git is available, and a `dirty` marker when the working tree has local changes. The commit id is the part of the local version string that identifies the source revision. After a `git pull`, the commit id is refreshed the next time CMake configures that build directory. If CMake does not reconfigure automatically, run the same `cmake -S ../CREXX -B .` configure command again before rebuilding.

`CREXX_BUILD_TIMESTAMP` is recorded in `BUILDINFO` for package provenance, but it is not part of the displayed tool version. When it is not supplied, CMake creates a UTC timestamp during the first configure of a build directory and stores it in `CMakeCache.txt`. Release and CI builds should pass an explicit timestamp:

```
cmake -DCREXX_BUILD_TIMESTAMP=20260527T120000Z -S ../CREXX -B .
```

A.7 Use of CREXX to build cRexx

CREXX is used to build the library of built-in functions that are written in REXX (and, for a very small part in REXX Assembler) and need to be compiled (carefully observing the dependencies on other REXX built-in functions) before they are added to the library and the CREXX executables in their binary form.

A.8 What do we have after a successful build

A.8.1 Native executables

When all went well, we have a set of native executables for the platform we built CREXX on. These are

TABLE 3: Delivered products.

Name	Function
CREXX	CREXX compiler driver
rxc	CREXX compiler
rxas	CREXX assembler
rxlink	CREXX linker
rxdas	CREXX disassembler
rxvm	CREXX VM, compiler-selected product entry point
rxpp	CREXX macro preprocessor
rxbvm	CREXX VM, portable switch-dispatch interpreter
rxtvm	CREXX VM, direct-threaded interpreter
rxvme	compiler-selected VM with linked-in REXX library
rxdb	CREXX debugger
rxcpack	CREXX C-generator for native executables

Clang and AppleClang make `rxvm` select `rxbvm`; GCC makes it select `rxtvm`. MSVC builds only the switch engine and supplies `rxvm.exe` as a copy of `rxbvm.exe`. The `rxtvm` executable is therefore not present in an MSVC build.

CREXX can compile the REXX script into an executable file, that can be run standalone, for example, on a computer that has no CREXX and/or C compiler installed.

The CREXX debugging tool `rxdb`, is written in REXX and compiled and packaged into an executable file.

The executables in the above table need to be on the PATH environment variable. These are to be found in the compiler, assembler, disassembler, debugger and cpacker directories of the cRexx-build directory we created earlier. It is up to you to add all these separately to the PATH environment, or to just collect them all into one directory that is already on the PATH.

A.8.2 Production and debug builds

Production builds are optimized, while debug builds are slower in execution time but deliver support for the analysis and debugging of problems in the code. The standard distribution is an optimized production build, while a debug build can be produced with the Debug CMake build option.

TABLE 4: Debug vs Release options.

Name	Function
-DCMAKE_BUILD_TYPE=Release	An Optimized build (default)
-DCMAKE_BUILD_TYPE=Debug	A Debug build

A.8.3 Libraries

We also have a set of libraries. Some are written in CREXX and others can be written in C and other programming languages. All executables and libraries are delivered in the `bin` directory of the distribution package.

A.8.4 Optional libraries - build options

TABLE 5: Optional plugin build options.

Name	Function
-DENABLE_ODBC=ON	Build the ODBC Plugin
-DENABLE_GTK=ON	Build the GTK (GUI) Plugin

Some libraries, with dependencies on installed software products, are only produced when they are opted-in with CMake build options. The defaults for these options are `OFF`.

B

cREXX Assembler Architecture (rxas)

The rxas assembler is responsible for translating human-readable Intermediate Representation (IR) assembly (.rxas) into packed executable bytecode (.rxbin) consumed by the rxvm interpreter.

B.1 1. Assembler Pipeline

The assembler processes source files through a pipelined, pseudo-two-pass architecture:

1. **Lexical Analysis (re2c)**:
   - Source defined in assembler/rxasscan.re.
   - Tokenizes input into REXX Assembly primitives: registers (RREG, GREG, AREG), literal types (STRING, INT, FLOAT, DECIMAL, HEX), symbols (ID, LABEL, FUNC), and assembler directives (e.g., .locals, .globals, .expose, .meta).
   - HEX is the RXAS binary-literal token. It accepts byte-paired 0x.../0X... text, including empty 0x, and is stored as OP_BINARY rather than as an integer literal.
   - The lexer also recognizes the interface metadata keywords used at the end of .meta records: .interface, .implements, and .member.
2. **Parsing (Lemon)**:

- Grammar defined in `assembler/rxasgrmr.y`.
- Enforces the structural integrity of the `.rxas` file (headers, function definitions, variable declarations, and instruction sequences).
- The parser actions invoke Builder API functions directly (e.g., `rxasgen*`, `rxaslabl`, `rxasproc`), translating syntax rules into buffered internal data structures.
- Instruction operands are parsed as a recursive comma-separated list. `rxasquev()` and `rxasgenv()` carry the resulting variable-length token vector; there is no three-operand grammar or queue limit.
- Instruction names are derived from `binutils/include/rxops.h`, with a public-source filter for bytecode/runtime-only entries. `RESERVED_*`, `INULL`, `INTERRUPT`, and `IUNKNOWN` remain opcode-table entries but are deliberately rejected as RXAS mnemonics. Recent core runtime additions include the object/interface instructions and the `sock*` TCP socket instructions.

3. **In-Memory Buffering & Constant Pooling**:
   - Handled in `assembler/rxasassm.c`.
   - Instructions and operand slots are appended sequentially into a dynamic array of `bin_code` elements.
   - Complex and pooled literal types (Strings, Binary literals, Floats, Decimals, Procedure Headers, Metadata) are deduplicated via AVL Trees and injected into a variable-length **Constant Pool**.
4. **Backpatching & Optimization (Second Pass)**:
   - Forward references (branches to undefined labels, calls to undefined procedures) are logged as a linked list of references.
   - At the end of the parse, `backptch()` traverses all trees, resolves symbol addresses, updates the binary stream, and applies peephole jump optimizations.

B.2 2. Core Internal Data Structures

The state of the assembler is held within the `Assembler_Context` struct, specifically within the `context->binary` object, which mirrors the layout of the final `.rxbin` file.

B.2.1 Instruction Stream

Instructions are flattened into an array of unions called `bin_code` (defined in `binutils/include/rxdefs.h`). An instruction is represented by an opcode element, immediately followed by elements representing its operands.

The canonical operand format in `rxops.h` is a NUL-terminated signature: one kind code (`R`, `I`, `S`, `L`, and so on) per operand. Consumers iterate that signature instead of switching over a closed `FMT_*` enum. The in-memory `no_ops` field remains a 32-bit `int`, preserving the eight-byte `bin_code` cell; the practical operand-count bound is therefore the bytecode stream's existing `INT_MAX`, not a small instruction-format ceiling.

```
// Example buffer size handling in gen_instr()
context->binary.binary[context->binary.inst_size].instruction.opcode =
    opcode;
context->binary.binary[context->binary.inst_size++].instruction.no_ops
    = operands;

// Operand elements
context->binary.binary[context->binary.inst_size++].index =
    get_reg_number(...);
context->binary.binary[context->binary.inst_size++].iconst = token->
    integer;
```

B.2.2 Constant Pool

Strings, binary literals, pooled float literals, procedure mappings, debug metadata, and exported symbols are packed into `const_pool`. This is a sequential buffer of dynamically sized records. Every record starts with a `chameleon_constant` header dictating its type and byte size. Types include: `STRING_CONST`, `BINARY_CONST`, `FLOAT_CONST`, `PROC_CONST`, `EXPOSE_REG_CONST`, `EXPOSE_PROC_CONST`, `META_FUNC`, `META_INLINE`, `META_REG`, etc.

Native-callable provenance uses `META_PROVIDER`, paired by callable symbol with the canonical `META_FUNC` signature:

```
.meta "namespace.callable"=".provider" "stable-provider-id"
.meta "namespace.optional"=".provider.optional" "stable-provider-id"
```

`.provider.required` is accepted as an explicit alias of `.provider`. Provider IDs are platform-independent and restricted to ASCII letters/digits followed by let-

ters, digits, ., _, or -. The assembler stores only symbol, provider ID and required/optional flags in this record; it does not duplicate the callable signature. RXBIN 007 writers set the native-provider feature bit.

Packaged bytecode provenance uses META_AUTOLOAD, also paired by callable symbol with the canonical META_FUNC signature:

```
.meta "namespace.callable"=".autoload" "packaged-library"
```

The payload is an exact platform-independent .rxbin filename stem, never a path. It must start with an ASCII letter or digit and may then contain ASCII letters, digits, ., _, or -. RXAS stores the symbol and artifact stem as typed string references. RXBIN 007 writers set the autoload-hints feature bit. The VM may use the record only if that callable remains unresolved after the already loaded module set has linked.

Module initialization uses a dedicated META_INITIALIZER record:

```
.meta "example.boot"=".initializer" ".void" boot() ""
```

The five-field shape deliberately resembles callable metadata so generated RXAS and rxdas round trips retain the namespace-qualified diagnostic symbol and the local procedure reference. The assembler requires .void, an empty argument signature, and a local bytecode procedure. It stores the symbol and procedure as typed constant-pool references rather than treating the label as an exported callable. Multiple records are valid and their metadata-chain order is execution order. RXBIN 007 writers set the initializer feature bit.

The serialized expose_head chain includes both EXPOSE_REG_CONST and EXPOSE_PROC_CONST records. Runtime linking and other module-local walkers now rely on that chain instead of scanning the whole constant pool.

The interface/callable-contract and initializer work extend that same metadata path rather than introducing a second binary header mechanism. In addition to META_CLASS and META_ATTR, the assembler now serializes:

- META_INTERFACE for one interface header
- META_IMPLEMENTS for one concrete-class-to-interface link
- META_MEMBER for one interface method or factory declaration

Cross-file compiler inlining also uses the metadata path. Callable signatures remain in META_FUNC; inline-body templates are carried separately in META_INLINE,

emitted in RXAS as:

```
.meta "fully.qualified.callable"=".inline" "I6;c,1,..."
```

The `I6` payload is the compiler-owned inline transport described in `compiler/docs/inlining_design.md`. `rxas` stores it as `META_INLINE`, and `rxdas` must emit it back to the same logical `.meta ... ".inline" "I6;..."` spelling so source, RXAS, and binary import paths do not drift. Linked final images normally strip `META_INLINE`; library artifacts preserve it for downstream `rxc` optimisation. The leading `c` record is a versioned callable proof summary; `rxc` reconstructs its facts from the transported body and requires exact agreement before enabling imported inlining.

Source/debug metadata now has two separate identities:

- `.srcstep step-id` is a module-local id for one concrete source anchor: pooled file name, whole source line, active range, and provenance flags. It is not an instruction address or a source line number.
- `.srcstep` and `.traceevent clause-id` is a module-local grouping key for all anchors and semantic events that belong to one logical authored clause. Simple clauses often use the same value for both ids; split clauses, generated helper code, loop pieces, or inlined fragments may use several step ids for one clause id.

`0` means there is no source/clause anchor. In a linked image, consumers should treat identity as module plus id because ids are only local to their original module.

The RXAS spellings are:

```
.srcstep <step-id> <clause-id> <flags> "<file>" <line> <start-col> <end
    -col> "<whole-source-line>"
.traceevent "<kind>" <mode-mask> "<value-source>" "<value-type>" "<
    register-type>" <value-ref> <source-step-id> <clause-id> <flags> "<
    symbol>" "<resolved-name>"
```

TRACE event records deliberately store compact codes, not presentation strings. Current event kinds include `A` assignment, `V` variable read, `C` resolved compound name, `L` literal, `O` binary operation, `P` prefix operation, `F` function result, and `M` message. The TRACE exit handler maps those to classic prefixes such as `>=>`, `>V>`, `>C>`, `>L>`, `>O>`, `>P>`, `>F>`, and `+++`. The mode mask controls visibility in modes such as Results and Intermediates. `value-source` is `R` for a register, `K` for a constant-pool value, or `N` when no value is attached. Register-backed events must name an actual available register at that address; handlers must not infer values from source text

or scope metadata.

Metadata-only modules are valid. For example, an interface contract file may compile to .rxas containing .meta records and no function bodies; rxas must still emit a .rxbin so import and runtime factory resolution can load the contract metadata.

For interface methods, the member-kind string now distinguishes: - method for an abstract interface method - method final for a Level B final/default interface method body

B.2.3 Jump Tables

Procedure-local .jtable declarations and same-line .jcase label decorations are resolved after label optimization. The assembler emits the resulting table as a host-layout-independent BINARY_CONST; jumps, jumpr, jumpn, jumpb, jumpbs, and jumpi operands are patched to that pool entry. jumps uses exact bytes, jumpr canonicalizes trailing-space-padded REXX text, and jumpn canonicalizes numeric strings through the shared parser in binutils/include/rxnumparse.h. Release 1 packed algorithms are linear, openhash, and acph. auto uses linear for one case. Larger tables use open hashing for average key lengths up to two bytes; tables of at least 256 cases also use it for averages up to four bytes. Remaining tables use ACPH. Shared format constants and hash functions live in binutils/include/rxjtable.h and must remain common to the assembler and VM readers.

The complete packed layout, canonicalization, corruption, disassembly, and ownership contract is in RXBIN_JUMP_TABLES.md.

B.2.4 Binary Literals

RXAS binary literals are written as byte-paired hex with a 0x or 0X prefix. 0x00ff stores two bytes (00 ff) in a BINARY_CONST, and 0x stores an empty binary constant. rxdas emits the canonical lowercase 0x... spelling. Operand rendering is dynamically sized: a binary constant may be much larger than the disassembler's ordinary stack line buffer and must still round-trip in full. For an oversized binary used directly by one or more instructions, ordinary rxdas output emits the value once as a private §rxdas.const.<pool-offset> .const alias and uses that alias at each operand; valid jump-table binaries retain their procedure-local .jtable form. The explicit -p constant-pool dump still prints each complete value. encode_binary_-

`to_hex()` is a bounded `snprintf`-style formatter; do not reintroduce pointer or remaining-length arithmetic that advances beyond the supplied buffer.

`load rDst,0x...` uses `LOAD_REG_BINARY` and loads the VM register's binary slot, not its string slot. Binary-memory VM instructions exposed at RXAS are:

- `blen rOut,rBin`
- `blen rOut,0x...`
- `bresize rBin,rLength`
- `bclear rBin`
- `bfill rBin,rByte`
- `getbyte rOut,rBin,rIndex`
- `setbyte rBin,rIndex,rByte`
- `bcopy rDst,rSrc`
- `bcopy rDst,rSrc,rOffset`
- `bcopy rDst,0x...,rOffset`
- `bgetu8 rOut,rBin,rOffset`
- `bgeti8 rOut,rBin,rOffset`
- `bgetu16 rOut,rBin,rOffset`
- `bgeti16 rOut,rBin,rOffset`
- `bgetu32 rOut,rBin,rOffset`
- `bgeti32 rOut,rBin,rOffset`
- `bgeti64 rOut,rBin,rOffset`
- `bgetf32 rOut,rBin,rOffset`
- `bgetf64 rOut,rBin,rOffset`
- `bgetu8 rOut,0x...,rOffset`
- `bgeti8 rOut,0x...,rOffset`
- `bgetu16 rOut,0x...,rOffset`
- `bgeti16 rOut,0x...,rOffset`
- `bgetu32 rOut,0x...,rOffset`
- `bgeti32 rOut,0x...,rOffset`
- `bgeti64 rOut,0x...,rOffset`
- `bgetf32 rOut,0x...,rOffset`
- `bgetf64 rOut,0x...,rOffset`
- `bsetu8 rBin,rOffset,rValue`

- bseti8 rBin,rOffset,rValue
- bsetu16 rBin,rOffset,rValue
- bseti16 rBin,rOffset,rValue
- bsetu32 rBin,rOffset,rValue
- bseti32 rBin,rOffset,rValue
- bseti64 rBin,rOffset,rValue
- bsetf32 rBin,rOffset,rFloat
- bsetf64 rBin,rOffset,rFloat
- pgeti rOut,rBin,rIndex
- pseti rBin,rIndex,rValue
- pgetf rOut,rBin,rIndex
- psetf rBin,rIndex,rFloat
- bcheckrange rBin,rOffset,rLen
- bgets rString,rBin,rOffset
- bgets rString,0x...,rOffset
- bsets rBin,rOffset,rString
- bsets rBin,rOffset,"literal"
- sget rString,"literal",rOffset
- bmove rDst,rSrc,rLen
- bmemmove rBin,rDstOffset,rLen
- bcmpb rCmp,rBin,rNeedle
- bcmpb rCmp,0x...,rNeedle
- bcmpb rCmp,rBin,0x...
- bcmpb rCmp,0x...,0x...
- bcmps rCmp,rBin,rString
- bcmps rCmp,rBin,"literal"
- bcmps rCmp,0x...,rString
- bcmps rCmp,0x...,"literal"
- bconcat rDst,rLeft,rRight
- bappend rDst,rRight
- bslice rDst,rSrc,rStart,rLen
- bupdate rDst,rOffset,rSrc
- stobin rReg

- `bintos rReg`

Indexes, offsets, and lengths are bytes and zero-based. The typed `bget*` and `bset*` instructions use canonical little-endian storage order; they do not use host-native struct layout, alignment, or padding. `bgetu8` is the strict counterpart to current `getbyte`: `getbyte` still returns `-1` for out-of-range reads, while typed reads raise `OUT_OF_RANGE`. Typed writes raise `OUT_OF_RANGE` when the field does not fit or when an integer value is outside the target storage type's range. `bgeti64`/`bseti64` are the Release 1 `.int` storage form. `bgetf32`/`bsetf32` store IEEE binary32 values and widen/narrow through the VM float register; `bgetf64`/`bsetf64` store IEEE binary64 values.

`pgeti`/`pseti` and `pgetf`/`psetf` are the distinct Release 1 host-native packed-item surface. Their index is a zero-based item number rather than a byte offset. Integer items have the exact host `rxinteger` representation and float items have the exact host VM `double` representation. They deliberately do not provide endian or width conversion, and the same binary bytes may be viewed as either item type on different accesses. A negative index, index multiplication overflow, or an item that does not fit wholly inside the binary's logical byte length raises `OUT_OF_RANGE` before a write. RXAS accepts register operands for these operations; the compiler materializes a readable binary constant into an aligned runtime binary register before emitting `pgeti` or `pgetf`.

`bresize` sets the logical byte length and zero-fills newly exposed bytes. The VM may keep a larger private physical capacity, grown in blocks and reused across later logical resizes; `blen` reports only the logical length. Negative lengths raise `OUT_OF_RANGE`, and allocation failure raises `FAILURE`. `bclear` sets the logical byte length to zero. `bfill` fills the current logical byte range and raises `OUT_OF_RANGE` for bytes outside `0..255`.

`bcopy rDst,rSrc,rOffset` copies exactly the current logical length of `rDst` from a binary register or constant source at a byte offset. `bcheckrange` is a strict side-effect-free range check for `rOffset..rOffset + rLen`; negative offsets, negative lengths, and ranges past the logical binary length raise `OUT_OF_RANGE`. `bgets`/`bsets` read and write zero-terminated UTF-8 fields in binary memory, including the stored NUL on writes but excluding it from the Rexx string value on reads. `sget` extracts a codepoint-counted slice from a `STRING_CONST`, starting at a byte offset that must be a UTF-8 boundary. `bcmpb` and `bcmps` compare without materializing a temporary

copy; the compare register supplies the input source offset and is overwritten with -1, 0, or 1.

bslice rDst,rSrc,rStart,rLen performs explicit zero-based extraction without mutating the source. The former binary cursor instructions are retired and their numeric opcode slots remain reserved; old RXBIN images using them are not compatible with this instruction set. Any build directory or auxiliary worktree first built before this cursorless change must therefore be cleaned and rebuilt before use; mixing retained generated RXBIN with current tools is unsupported. Release 1 source lowering also uses target-sized bcopy, typed reads/writes, and zero-copy compares where a materialized slice is unnecessary. setbyte and bupdate are strict and raise OUT_OF_RANGE for invalid indexes, bytes outside 0..255, or overlay writes past the destination length. stobin copies the register's current string bytes into its binary slot. bintos validates the register's current binary bytes as UTF-8 and copies them into its string slot; invalid bytes raise UNICODE_ERROR in UTF builds.

The explicit string form is substring rDst,rSrc,rStart,rLen, where positions and lengths count Unicode characters. A negative start clips to zero and a non-positive length produces the empty string. bslice likewise clips a negative start to zero, but a negative binary length raises OUT_OF_RANGE. Both operations are safe when destination and source name the same storage. A required allocation either completes before the destination changes or raises FAILURE; the opcode signal metadata records these failure-before-write contracts.

Level B exposes these byte-buffer instructions through the rxfnsb binary helpers (binlength, binbyte, binsetbyte, binsubstr, binconcat, binoverlay, bininsert, bindelstr, binpos, bincompare, bin2x, x2bin). The source-level || operator also lowers to bconcat when either operand is .binary; blank concat remains a text-only operation.

B.2.5 Attribute Array Helpers

RXAS exposes VM attribute-array operations for array/object backing storage:

- getattrs rOut,rArray
- setattrs rArray,rCount / setattrs rArray,count
- minattrs rArray,rCount / minattrs rArray,count

- `linkattr` / `linkattr1`, `linktoattr` / `linktoattr1`, and `unlinkattr` / `unlinkattr1`
- `insattrs rArray,index,count` and `delattrs rArray,index,count`
- `insattrs1 rArray,index,count` and `delattrs1 rArray,index,count`

The forms without a `1` suffix use zero-based indexes. The `*1` forms use one-based indexes and are the usual fit for Level B array BIFs. Bulk insert accepts an index in `0..num_attributes` (`1..num_attributes+1` for `*1`) and inserts before that position. Bulk delete is strict: the requested range must fit inside the current attributes. Passing a count of zero is a no-op, but the index still must be in range. Invalid ranges raise `OUT_OF_RANGE`.

B.2.6 Reference Helpers

RXAS exposes the VM-first reference surface before any Rexx source syntax is finalized:

- `mkref rRef,rSource` creates a reference value to the storage currently named by `rSource`, after existing local links have resolved to their target storage.
- `deref rDest,rRef` copies the current referenced value into `rDest`. This is a snapshot copy, including nested attributes, not a live alias.
- `linkref rLocal,rRef` links a local register to the referenced storage until the existing `unlink rLocal` restores its base storage.
- `setref rRef,rSource` copies `rSource` into the referenced storage.
- `refvalid rOut,rRef` stores `1` when the reference is still valid and `0` otherwise.
- `unref rRef` clears a reference value and releases its reference-cell retain.

Invalid reference use raises `REFERENCE_INVALID`; `refvalid` is the non-raising probe. A reference becomes invalid when its target storage lifetime ends, such as deleted/shrunk attribute storage or frame-owned local storage after return. Plain overwrite of still-live storage keeps the reference valid. These instructions are optimiser barriers until reference alias effects are modelled more deeply. Assigning a scalar value into the register that holds a reference value clears that register's reference payload; it does not invalidate the referenced target storage.

In UTF builds, RXAS string constants are text, not byte containers. Hand-written string operands are unescaped and validated before entering the constant pool; invalid UTF-8 is an assembly error with guidance to use `0x...` binary literals. Use `load rDst,0x...`, `freadb`, `fwriteb`, `sockrecvb`, or `socksendb` for arbitrary bytes.

B.2.7 Symbol Tracking (AVL Trees)

To deduplicate constants and resolve identifiers in `O(log N)` time, the assembler leverages a custom AVL tree implementation (`avl_tree.h`). Active trees include: `- string_constants_tree - decimal_constants_tree - binary_constants_tree - label_constants_tree - proc_constants_tree - extern_constants_tree`

B.3 3. Two-Pass Resolution (Backpatching)

Assembly requires a two-pass approach because a jump or call can reference a label or procedure defined further down in the file. `rxasassm.c` handles this elegantly by building a `struct backpatching` header for every symbol encountered:

```
struct backpatching {
    int defined;          // 1 if definition encountered, 0 if only
        referenced
    size_t index;         // Final resolved target index in the binary
        or const_pool
    struct backpatching_references *refs; // Linked list of unresolved
        usages
};

struct backpatching_references {
    size_t index;         // The instruction operand index in the
        bin_code array
    Assembler_Token *token;
    struct backpatching_references *link;
};
```

When the `EOF` is reached, the `backptch(Assembler_Context *context)` function orchestrates the final resolution:

1. **`optimise_labels()`**: Performs peephole optimization. If a label simply targets an unconditional branch (`br`), the target of the label is recursively collapsed to point directly at the final destination, saving execution cycles.
2. **`backpatch_procedures()`**: Walks the `proc_constants_tree`. Raises errors for any procedures referenced but never defined. Resolves valid procedures.
3. **`backpatch_labels()`**: Walks the `label_constants_tree`. Updates the placeholder `bin_code` indices mapped in the `refs` linked list with the actual instruction offsets.

B.4 4. Binary Emission (.rxbin)

Once parsing and backpatching conclude without errors, the assembler flushes the Assembler_Context state into an RXBIN 007 container.

The structural output consists of:

1. **Magic Header & Version**: Validates the file format.
2. **Module Directory and Section Sizes**: the module identity, global-register count (context->binary.globals), six fixed section ranges, and exact stored sizes.
3. **Portable Constants and Metadata**: the native in-memory pool is converted to fixed-width, little-endian, ID-linked records.
4. **Bytecode and Semantic Graph**: the resolved instruction slots are variable-integer encoded, constant operands become record IDs, graph-bearing operands become dense graph IDs, and graph facts/indexes occupy their own sections.

As of format version 002 and later, float operands are no longer stored inline as raw double payloads in operand slots. Instead, the operand slot carries an index to a deduplicated FLOAT_CONST record in the constant pool.

rxdas emits float operands and FLOAT_CONST pool entries with enough significant digits to distinguish binary64 values that would otherwise share a six-decimal rendering, while keeping integer-looking values parseable as RXAS float tokens (for example, -0.0 rather than -0).

RXAS still builds the normal in-memory bin_code[] and raw constant pool first. write_module() then emits only 007: signed integers use ZigZag variable integers, portable records replace native pool layouts, and the base sections are uncompressed. There is no legacy-output mode; 006 is rejected and repository artifacts are rebuilt. The 007 container and semantic graph are specified in RXBIN_007_SEMANTIC_GRAPH.md.

The fixed direct-bytecode call forms call1 through call4 name each caller argument register explicitly. RXAS sets RXBIN007_FEATURE_FIXED_CALLS whenever one is present; call with a count register remains the general fallback for higher arity, imported/native and dynamically selected targets. The existing two-operand call result,procedure() remains the zero-argument direct form.

RXAS feeds the common `rxbin` graph builder from class, interface, implements, member, callable, factory, and signature facts. After ordinary backpatching it resolves local procedure references, finalizes the indexed module graph, and emits the same sectioned 007 container used for linked images, with a one-entry module directory. Unknown external type/member references are complete opaque nodes, not dangling strings.

Parameter and return types remain canonical text on type nodes, including preseeded built-in nodes such as `.float`; signatures refer to those nodes with module-local dense IDs. Type-, member-, and factory-bearing serialized instruction operands also use module-local graph IDs. RXAS/RXDAS text syntax remains human-readable and descriptor based.

B.5 5. Current Interface-Dispatch Additions

The current interface runtime slice relies on three assembler-visible opcodes and the metadata records above:

- `setobjtype rX,"fully.qualified.class"` stamps a newly created object value with its concrete runtime class identity and clears any uninitialized-object marker
- `setobjuninit rX,"fully.qualified.class"` writes a typed object placeholder and marks it uninitialized; the compiler uses this for bare object class defaults such as `.SomeClass`
- `assertinitialized rX` raises `OBJECT_NOT_INITIALIZED` if `rX` is a typed uninitialized object value
- `isinitialized rOut,rX` stores `0` for a typed uninitialized object and `1` otherwise
- `srcmethodsel rProc,rObj,"rxsig1|member|return_type|args"` resolves a concrete procedure from an object value plus a method descriptor
- `srcfprocsel rProc,"rxsig1|fully.qualified.interface|return_type|args",rArgs` resolves the default `*` factory provider for an interface
- `srcfprocsel rProc,"rxsig1|fully.qualified.interface..factory_name|return_type|args",rArgs` resolves a named factory provider for an interface
- `typeof rOut,rObj` returns the canonical source type name of an object value

- `istype rOut,rObj,"type"` checks an object value against an interface, class, or `.object`
- `asserttype rObj,"type"` raises `CONVERSION_ERROR` if the object value does not match the requested interface, class, or `.object`

`istype` and `asserttype` check only object/class/interface compatibility. Initialization is a separate lifecycle check. `assertinitialized` is the raising check; `isinitialized` is the non-raising probe.

`srcfprocsel` supports both the default `*` surface and named factory selectors. Provider selection is a VM concern: the assembler simply emits the opcode and the interface/class metadata needed for runtime lookup. The selector string is a callable descriptor (`rxsig1|name|return_type|args`), which lets the VM and linker reject providers whose metadata has the right name but the wrong signature.

For `call ... , rArgs`, `dcall`, and `srcfprocsel ... , rArgs`, the trailing register operand names the argument-count register. The actual argument values are taken from the contiguous registers immediately after that count register. That hidden contiguous argument block is semantically part of the instruction use set, so optimiser passes must treat those opcodes as barriers unless they fully model the implicit register consumption.

The peephole optimiser also consumes instruction metadata from `binutils/include/rxops.h`. `FLOW_JUMP`, `FLOW_COND`, and `FLOW_TERM` block `NO_HAZARD` rule skips automatically. `FLG_OPT_BARRIER` is for `FLOW_NEXT` instructions that still must not be skipped, such as calls, signal handler configuration, explicit signal/check instructions, and opaque argument-block operations. `FLG_IMPLICIT_REG_USE` is for linear instructions whose register effects are not represented as normal operands. For example, `inc0`, `dec0`, `inc1`, and friends are still linear execution, but the optimiser treats them as using the corresponding fixed local register when checking whether an intervening instruction is relevant to a rule.

The opcode metadata inventory is split deliberately without duplicating facts. `rxops.h` remains authoritative for the dense opcode list, operand format, control flow, opcode flags and source-mnemonic status. `binutils/include/rxopeffects.h` is the one-to-one semantic sidecar: it has one ordered entry for every opcode slot, including reserved and internal slots. `binutils/include/rxopsignals.h` is a second one-to-one sidecar for signal capability, failure phase, signal-name source,

failure-visible writes, dependencies, continuations and observability properties. The former coarse RXOP_SEM_MAY_THROW flag has been retired; generic effects no longer duplicate or approximate signal behavior. Handler-policy instructions additionally name their normal-path action (ignore, halt, branch/call/return, breakpoint, push/pop), exact static or literal signal-name source and source operand. An unknown signal set is distinct from an unknown policy write: a known call may have conservative signal behaviour without changing its caller's handler table. binutils/rxopmeta.c combines both sources through the stable rxop_effects() and rxop_signal_contract() C APIs; their count functions expose the mechanically checked inventory sizes.

The K04e handler audit records STRLEN as a string read with failure-atomic UNICODE_ERROR before its integer write, all three ISUB forms as failure-atomic OVERFLOW_UNDERFLOW before their integer write, and the three-form loose REQ through RLTE family as non-signalling string reads with an integer result. Integer subtract and loose comparison use the VM integer setter and therefore clear reference/native payload components; STRLEN preserves other destination components. These entries describe the existing handlers and do not change VM or RXBIN semantics.

RxOpEffects uses the following guarantees:

- state is CLASSIFIED, CONSERVATIVE, RESERVED, or INTERNAL. Conservative, reserved, internal and unknown/out-of-range queries are optimizer barriers and never claim a kill.
- reads and writes are conservative possible-access sets over explicit operand positions. Existing instructions retain the compact three-bit masks; wider instructions use one-bit-per-operand signature strings. Consumers use rxop_effect_reads_operand() and rxop_effect_writes_operand() rather than interpreting either representation directly. A consumer must account for two positions naming the same physical register.
- kills is a proof set: every named operand is definitely overwritten without reading its previous value. It is always a subset of writes and is deliberately narrower than possible writes.
- implicit covers fixed local-register read/modify/write, integer-coded local copy/target registers, argument-register indexes and the runtime-sized local range after operand 3 used by call, dcall, and srcfprocsel.

- `branch_targets` identifies explicit label operands through the same generic query representation. Packed jump-table instructions instead carry `RXOP_SEM_INDIRECT_BRANCH`.
- `semantics` covers direct and dynamic call boundaries, returns, alias creation/release, reference creation/read/write/release, storage-lifetime end, indirect writes, indirect branches and opaque side effects. Signal behavior comes only from `RxOpSignalContract`.
- `flow` and `optimizer_barrier` are returned in the same object but are derived from the canonical `rxops.h` flow/flags. A non-classified state also forces the returned barrier bit.

The current inventories each have 650 entries: 591 source opcodes (585 classified and six explicitly conservative process/redirect operations), 56 reserved slots and three internal handlers. Coverage is not inferred from an instruction name or format. The audit used the VM handlers between `START_OF_INSTRUCTIONS` and `END_OF_INSTRUCTIONS`, the assembler/compiler behavior described here, and focused semantic tests. The tests mechanically validate table alignment, operand-mask legality, flow/branch/call/implicit consistency, source coverage, reserved/internal treatment and fail-closed unknown queries; they do not claim to parse arbitrary C handler bodies formally.

RXAS optimisation routing is explicit in `assembler/rxas_flow_pass.c`. Every current rule family has one stable owner---local mechanical scan, immutable CFG, component SSA/use proof or diagnostic-only---and a capability mask over local scan, CFG, dominance, signal, storage, value, use and loop facts. One linear opcode/metadata census conservatively identifies potential consumers before whole-procedure analysis. A zero candidate count may avoid a consumer, but the census is never a correctness proof and is deliberately allowed to overselect. K05 branch threading and M00 reachability request CFG only; exact K06 copy/ status subsumption remains local; M01-M06 and K01-K04 retain their component proof ownership. M01-M04 request the component base, while M05, M06 and K01-K04 additionally request the sparse use index. The established `-d` flow dump has its own explicit diagnostic route. No current production consumer requests loop analysis.

D0.5 adds an exact linear write-once/single-use typed-copy route. One local- register census records explicit occurrences plus metadata, TRACE, implicit regis-

ter and call-window observations. ICOPY, FCOPY or SCOPY may bypass SSA only when its destination occurs exactly as that copy destination and one immediately adjacent, non-signalling component read; the read is redirected and the copy is deleted in the same sparse transaction. Reused registers, mapping operations, calls, TRACE/meta observations, signalling consumers and all non-adjacent uses remain on the SSA route.

NR-27 adds a transient whole-procedure machine-flow layer after the bounded 100-item local peephole and before ordinary RXBIN emission. Stable peephole output is moved, with its token ownership, into a growable procedure stream. One immutable RxasFlowProcedure is then built for each rewrite epoch; surviving records still pass through the same assembler emission functions, so RXAS syntax, canonical RXBIN and the public ABI do not change.

The peephole is a permanent preprocessing owner, not legacy code scheduled for removal. Exact local rewrites belong there when opcode metadata proves complete equivalence without procedure-wide liveness, alias/storage, signal-path, hidden-cleanup or TRACE-observation reasoning. Adjacent non-signalling algebra and mechanical encoding selection are the normal cases. CFG/SSA remains the owner when any of those wider facts are required. Increasing the bound must be measured against the same input for pre-graph census, graph/SSA size, emitted image, assembler time and peak memory; the bound is not a substitute for a procedure analysis.

assembler/rxas_flow_graph.c is the sole owner of stable record, instruction, basic-block, typed-edge and reachability descriptors for that epoch. M00 uses its entry/same-frame/asynchronous-root reachability to remove dead opcode, TRACE and source-step records. Existing semantic consumers share its lazy proof manager, and K05 collects a complete compatible branch-thread batch from the same CFG only after those consumers decline the epoch. Any mutation ends the epoch and rebuilds the graph, except that compatible semantic edits are one validated transaction whose proof-facing records remain pinned to the epoch originals. The original queue index remains the record identity; every opcode record maps to a stable instruction ID, block ID and exact pre-emission RXBIN address. Source-step, TRACE, label and metadata records keep their own IDs and map to the block/address at which they are observed.

Code blocks start at procedure entry, labels, branch and handler targets, and after

calls, signalling instructions and other control terminators. Seven stable synthetic blocks represent entry, handler dispatch, asynchronous handler entry, normal return, unwind, terminal exit and unresolved control. Edges distinguish normal fallthrough, branch, signal skip, handler, unwind, terminal and explicit fail-closed unknown flow. Signal edges come from `RxOpSignalContract.continuations`; handler-policy instructions with label operands feed the handler and asynchronous roots rather than inventing a normal branch from the registration instruction.

Construction snapshots record/opcode metadata, builds an open-addressed label index, forms blocks, appends edges and computes compact rooted reachability. It is expected linear in queued records plus emitted edges. There is no parallel legacy edge graph, dense record-by-register use/kill allocation or dense M07 storage environment. M07 is a debug-only executable oracle over sparse SSA storage nodes and queries. Read APIs require the owning epoch and return no descriptor for a stale epoch. `rxas -d` prints deterministic `PERF3 flow-*` and `PERF3 sparse-storage-oracle` records without pointer values, so graph, edge and retained-analysis identities can be compared mechanically.

Stage 3 layers a demand-driven structural-analysis manager over each immutable procedure epoch. `assembler/rxas_flow_analysis.c` retains successor edge IDs and unique predecessor block sets in sparse CSR form, then computes reachable reverse postorder from a virtual root joining the procedure-entry, registered- handler and asynchronous-handler roots. Unreachable blocks remain represented by the graph but are excluded from dominance, SCC and loop proofs.

The cached base structural result contains Cooper-Harvey-Kennedy immediate dominators, dominance-tree intervals and sparse dominance frontiers. SCC, backedge and loop-forest construction is a separate monotonic capability on the same cached object. When explicitly requested it adds iterative Kosaraju SCCs, dominance-classified backedges and a loop hierarchy. Natural loops with the same header share one region; irreducible SCCs remain explicit conservative regions. Instruction-level signal retry was retired on 2026-08-03, so backedges and loop regions represent source/control flow rather than synthetic retry cycles. Source order remains a diagnostic mapping and is never treated as dominance evidence.

Every structural query requires the graph epoch. The first successful result is cached until graph destruction; a failed deliberately small work budget may be

retried with a larger budget. Allocation failure, invalid control or budget exhaustion leaves the analysis unavailable and cannot enable a rewrite. Deterministic `PERF3 flow-analysis*` and `PERF3 flow-loop` diagnostics expose work, retained-byte, reachability, predecessor, dominator-iteration, frontier, SCC, backedge and loop counters; dormant loop facts print as unavailable rather than being inferred. Post-dominance is deliberately deferred until a consumer needs a must-execute query. Ordinary assembly does not solve unused loop analysis: tests and future optimizer consumers request it explicitly. Stage 3 therefore changes neither queued records nor RXBIN output.

Stage 4 adds `assembler/rxas_flow_signal.c`, a second demand-driven epoch cache layered on the structural result. Handler policy is modeled as a sparse write-once chain with entry, write, phi and clobber definitions. Procedure entry is an explicit inherited-unknown policy parameter. Static/literal handler installation produces an exact named action; unresolved opcodes clobber the whole policy. Parallel normal and signal-skip edges remain distinct phi inputs even when Stage 3 correctly reports one unique predecessor block. Cyclic phi queries defer the backedge identity and merge external definitions, which preserves an unchanged loop policy without using source order as proof.

Each signal edge selects policy from the instruction's recorded failure phase: before-write edges see the incoming policy, after-write edges see the normal transfer, and partial/unknown policy writes fail closed. Return and unwind edges expose the procedure-entry policy parameter because handler-table changes are local to the callee frame. The VM initially shares the caller's handler table and copies it on a child-frame write. This frame-local policy rule must not be confused with call arguments: call-window argument slots are pointers to caller-owned value storage, so callee writes and reference effects can remain caller-visible after return.

`sigpush` leaves the active handler unchanged, but the VM may silently fail to allocate its saved stack entry. A later `sigpop` therefore yields explicit stack-unknown policy unless a future proof establishes a successful matching push; the analysis never assumes restoration merely from lexical pairing.

Numeric context, plugin, locale, external state, reference-visible state, TRACE and call boundaries use separate sparse effect identities. Calls advance the call/reference/external/plugin/ identities while preserving handler policy. TRACE records advance only TRACE

identity. Before/after/partial signal edges select the corresponding effect versions; an unproved partial effect gets an explicit unknown definition rather than one global barrier. `rxas_flow_policy_at_instruction()`, `rxas_flow_policy_on_edge()`, `rxas_flow_effect_at_instruction()` and `rxas_flow_effect_on_edge()` are the initial narrow query surface. A deliberately small budget, stale epoch, allocation failure or malformed graph returns no usable result. Ordinary assembly does not request the cache; tests, `rxas -d` and future consumers do. Stage 4 is output-neutral and preserves exact canonical RXBIN images.

Stage 5 adds `assembler/rxas_flow_ssa.c`, a third demand-driven epoch cache over the same immutable procedure. It separates a register name from the storage currently named by that register. Local, argument and global registers begin with distinct base `StorageId` definitions; `link`, `linkarg`, attribute/reference links, `swap`, their fused forms and `unlink` then create, move or restore symbolic storage identities. In particular, `linkarg` names caller-owned argument storage rather than inventing callee-local value state. Unknown aliases and unsupported indirect mutation remain explicit unknowns. Calls preserve mappings that are structurally unchanged but advance the Stage 4 reference/effect identities needed to prevent a false unchanged-value proof. For a fused opcode that both remaps registers and writes values, the mapping is retained but component values fail closed until canonical metadata describes the order of its intra-instruction sub-events; the analysis does not guess whether the write named the pre- or post-remap storage.

Each storage component is represented by a write-once `ValueId`. The tracked set is integer, float, string, decimal, binary, attributes, reference, host-owned native payload and the distinct attribute-count component. The count is not a value-status flag: `setattrs`, `minattrs`, `getattrs` and attribute links describe it explicitly in canonical opcode metadata. Values distinguish procedure-entry, direct write, constant, copy, derived, known absent, phi and unknown definitions; a null/absent value is not the same as an unavailable proof. Copy propagation therefore carries presence as well as the source identity, including a `dcopy` of a null decimal. Join values and storage mappings are materialized lazily and cyclic phis canonicalize an unchanged loop identity without using source order. An unknown storage input has a separate value identity and cannot collide with a valid zero-based `ValueId` or simplify a join into a false proof.

Normal instruction states consume the canonical component read/write and

derivation metadata. Signal skip, handler and unwind edges instead select the Stage 1 failure-visible writes and the Stage 4 edge state. Derived `itos`, `ftos` and `dtos` string values name their integer, float or decimal source `ValueId` and the exact numeric/plugin/effect identities on which the operation depends. Derivation metadata also names the source operand, so a two-register conversion such as `itof rTarget,rSource` does not accidentally use the destination's old value as its proof source. This is storage/component infrastructure only: no Stage 5 consumer changes queued records or emitted RXBIN.

Persistent states retain only mapping and component definitions. Point queries walk and cache the required storage/value chain; they do not allocate a `nodes x registers x components` table. `rxas -d` deliberately materializes only actual derivation sites and prints deterministic `PERF3 flow-ssa*` counters, so diagnostics exercise the first proof candidates without forcing every possible point/component query. Work-budget exhaustion, allocation failure, a stale epoch or invalid control returns no usable analysis. Ordinary consumer-free assembly does not request this cache.

Stage 6 adds `assembler/rxas_flow_proof.c`, a fourth per-epoch cache and the only authority for migrated proof consumers. It exposes bounded queries for dominated successful repetition, equivalent scalar-constant writes, speculatability, loop must-execute and loop component invariance. Proof keys name symbolic `StorageIds`; result records name the generator/candidate `ValueIds`, effect identities and a stable failure reason. Value and effect phis are compared through conservative reduction and equivalent leaf sets rather than numeric identity or source order. Stale epochs, invalid control, unsupported signal dependencies, allocation failure and work-budget exhaustion return an unavailable or rejected proof and cannot enable a rewrite.

The proof cache is a capability-lazy service rather than an eager bundle. Every M01-M06 and K01-K04 call site presents its stable routing ID; the service acquires the route's CFG, dominance, signal, storage, value and optional use facts monotonically for the immutable epoch. Public query entry points reject when their required capability was not declared, so a query cannot silently expand analysis. A failed stronger capability does not invalidate already acquired lower-capability facts: that route rejects, while later base consumers remain safe and available. Loop queries likewise require the separate loop capability. The compatibility constructor still requests the complete service for tests and external callers that explicitly

need it.

Storage resolution creates a provisional phi only when recursive lookup of a join requires a cycle placeholder or when predecessor storages genuinely differ. Equal acyclic predecessors resolve directly to their common `StorageId`; diagnostics report the elided phi, retained input and cache counts. Whole-procedure semantic consumers are admitted only when the indirect-value work bound and the 262,144 block-by-register join bound both pass. An over-bound procedure continues through local mechanical rewrites, M00 and K05, but SSA consumers and diagnostic SSA/use materialization fail closed. Future recovery of valuable advanced cases must use an explicitly bounded candidate-sliced or region proof rather than raising this limit.

Compatible semantic rewrites are committed through the sparse transactional manager in `assembler/rxas_flow_batch.c`. M04, K02/K03, M05, M06, K04, M02, M03 and M01 retain their established priority, but disjoint typed plans may be collected from one immutable epoch before one rebuild. Only records actually edited are snapshotted. Proof-facing `RxasFlowRecord` pointers are pinned to those originals while later consumers inspect the provisional queue; complete validation either commits the registered records together or restores all of them. A delete-only consumer additionally requires its target to remain unchanged from the epoch. K01 remains a separately budgeted storage- permutation epoch, and K05 remains a later CFG-only batch because topology- changing rewrites require a different compatibility proof from semantic SSA rewrites. Deterministic `PERF3 semantic-batch` diagnostics report plans, changed/deleted records, opcode replacements, operand rewrites and private locals provisioned. A transaction may retarget the register operand of an otherwise unchanged TRACE record when an immutable proof plan owns that exact event. Procedure `.locals` and the assembler's current-local count advance only after every record in the transaction validates and commits; a rejected or zero-candidate plan cannot leak a local or leave a partial TRACE rewrite.

An immediate one-based `linkattr1` can additionally name an interned attribute path. Its identity contains the owner `StorageId`, exact attribute-count `ValueId`, reference-effect generation and slot. The path is reusable through owner register aliases, but a dynamic/non-positive slot, unknown or merged count, different owner, or changed reference generation remains fail-closed.

The M05 migration adds assembler/rxas_flow_use.c, a fifth demand-driven per-epoch cache over component SSA. It indexes direct ValueId uses, storage observations and reverse phi dependencies once per procedure; edge-aware liveness then propagates only through those sparse dependencies. Explicit operands, read/write operands, metadata, register-backed TRACE, implicit reads, call windows and opaque observations remain distinct use kinds. Component writes are indexed separately from observations. Writes through a storage phi are expanded once to every possible leaf storage; an unresolved target becomes an opaque-write barrier, not a false read of an unrelated ValueId. A range call is retained as one bounded call-window observation rather than expanded across every local/component pair.

The first use-index consumer replaces the dense per-candidate typed-copy solver. Exact icopy, fcopy and strict scopy deletion requires a local, unaliased destination, the copy's write-once result ValueId, an equivalent source value at every redirected use, no metadata/TRACE/read-write or live call-window observation, and at least one exact component-only operand rewrite. The proof service returns an immutable all-or-nothing rewrite plan; the optimizer validates the complete plan before changing any operand or deleting the copy. Budget exhaustion, stale epochs and incomplete use facts reject the whole plan. The old availability/may-reach solver and its repeated dense register scans have been removed.

X01 is the first component-placement consumer built on the same sparse use index. It recognizes an exact adjacent typed copy followed by a one-register metadata-derived XTOY operation, but asks the proof service---not adjacency---to authorize the rewrite. Its immutable plan proves distinct local unaliased storage, the copied input and derived result ValueId relationships, dead displaced components, redirectable result uses, and compatible call, signal, context, reference, metadata and TRACE observations. The consumer validates the complete plan, retargets the derivation to the original source, redirects all proved result uses, removes only matching derived-value TRACE events, and deletes the copy as one transaction. The first production case is dcopy temporary,source followed by dtos temporary; the mechanism is component- and derivation-metadata driven rather than mnemonic-specific. Reference objects are a separate observation from register-to-storage aliases: an earlier mkref may keep either storage visible even when the register alias census is one. X01 therefore rejects a placement when such a reference-creation path can reach the candidate; reference lifetime/release is not

guessed.

H01 is the first loop-capable value-reuse consumer. It recognizes repeated joined string keys only after exact CONCAT component metadata identifies an equivalent literal/constant left component and right-component `ValueId`. The proof requires a dominating lazy seed and candidate within one common reducible natural loop, private destination storage at both construction sites, exact signal/effect compatibility and a complete redirectable use set. Source order and register-number equality are not proofs. A changed component, reference or alias exposure, relevant call-window observation, unowned TRACE event, irreducible/no-common loop, stale epoch or exhausted capability rejects the candidate.

Accepted candidates sharing one seed are planned against the same immutable epoch and receive one new private local. The seed CONCAT, its exact TRACE event and direct proved seed uses are retargeted to that local; later equivalent CONCAT and owned TRACE records are deleted and their stem-key uses are redirected. This permits the compiler's former temporary register to be reused or overwritten later without changing the cached generation. Placement remains lazy: the current production case deliberately rejects preheader hoisting because the seed is conditional, the right component is loop-variant and the TRACE event is ordered. The mechanism is metadata/value/use driven rather than specific to RexxCPS, a stem name or a register number.

The first migrated authority was repeated one-register `itos`. A deletion requires the generator's successful continuation to dominate the candidate, the same storage and equivalent integer source/string result, and equivalent numeric-context and reference-visible effect dependencies. The old private per-generator availability solver has been removed. The new service is allowed to prove a larger safe domain; the old solver's decisions are a retained minimum-capability baseline, not a parity oracle.

Calls model caller-owned argument mutation without becoming a universal local barrier. `call1` through `call4` create unknown component definitions for their explicit actual arguments. A range call records its local call-window base and uses a statically constant count when available; an unknown count conservatively covers every later local. Storage outside the argument window may remain provably unchanged. Normal and failure continuations use the same argument exposure rule, so signal edges cannot recover a value the callee may have changed.

The NR-27 register universe distinguishes local (r), argument (a) and global (g) register classes and tracks integer, float, string, decimal, binary, attribute and reference views. Explicit and implicit accesses come from rxop_effects(). Register metadata and register-backed TRACE records are observations. Alias/reference/lifetime/indirect/op operations taint involved registers and invalidate the relevant facts. Legacy consumers still treat calls and other modeled barriers conservatively; migrated component-SSA consumers may retain facts only for storage proved outside the call argument window and unaffected by reference-visible effects. Registered branch SIGNAL handlers are conservative asynchronous observation targets for every executable instruction in the procedure. An unresolved label or unknown opcode makes control flow incomplete and disables all NR-27 rewrites for that procedure. A packed jump-table branch is complete only when its procedure-local declared table, every .jcase label and the mandatory miss fallthrough are all present in the same retained procedure stream. The flow graph then includes every case edge plus the miss edge. Undeclared, malformed, cross-procedure, unsupported or off-graph-miss tables remain fail-closed.

Component information is a canonical opcode-metadata service rather than an NR-27-local mnemonic switch. rxop_component_reads() and rxop_component_writes() distinguish integer, float, string, decimal, binary, attribute, reference and native-payload components for each explicit operand. rxop_component_clears() records components made provably absent by a write. For example, integer and float constant loads clear both reference and native payloads; bcopy carries binary and native payload together. rxop_value_derivation() identifies proved representation relationships; rxop_derivation_context_reads() and rxop_context_writes() make numeric context part of those facts. rxop_signal_contract() distinguishes proven non-signal, known signal and fail-closed unknown behavior and records pre-write, post-write, partial-write or unknown failure phases. Unproved opcodes and signal locations remain conservative. The contract inventory marks the VM concat/sconcat family non-signalling, stem writes failure-atomic before writes, and decimal comparisons plugin-signalling after their integer result write. Alias-topology effects are independently versioned from mutation observable through an existing reference. An unknown exceptional signal set does not invent a normal numeric-context write for an otherwise classified opcode. A register-backed TRACE event observes only the component named by its value type; the event remains present and does not automatically observe every repre-

sentation inside the value.

TRACE observation is profile-sensitive. In an optimised assembly, a rewrite may omit, combine or relocate an event when it removes or moves the operation or value represented by that event. It must delete a removed value's matching event atomically, retain unrelated events, and never leave metadata reading a deleted, stale or mismatched component. Multiple retained events may share one reached address and remain ordered, so `CNOP` is not needed merely to separate them. `rxas -n`, together with compiler no-opt mode, provides the source- correspondent trace path; ordered batching is not a requirement to preserve every pre-optimisation event.

The graph-owned storage-identity service follows direct `link`, `swap` and `unlink` mapping rather than equating raw register numbers. The first component consumer uses this identity to remove a later one-register `itos` only when an earlier `itos` for the same storage proves both the unchanged integer source and unchanged string result under the same numeric context on every incoming normal path. Relevant component writes, context changes, ambiguous/tainted references, caller-owned argument mutation, opaque or indirect effects and unproved signal phases reject the proof. An unrelated no-argument call does not invent a local-value write. TRACE records are retained; ordered same-boundary delivery lets the producer event survive even when the redundant executable conversion is removed. The proof service is generic: after `itos`, M01 deleted the old repeated-`itof` authority and made a metadata-driven one-register `xtoy` consumer its successor. All 20 such conversions now name their exact source component, result component, derivation and context dependencies in canonical metadata; the consumer has no conversion-mnemonic allow-list. The proof admits only a dominating successful repetition with equivalent component values and effects.

The first larger-domain case is `itod`. Both bundled decimal plugins implement integer-to-decimal conversion for every integer, with allocation failure handled by the VM panic convention. `itod` and the same-function `btod` therefore clear stale backend diagnostics and are total non-signalling operations; their values still depend on numeric context and plugin identity. Signalling conversion families remain rejected where the first successful continuation does not dominate the repetition. `btoi` and `itob` remain rejected pending a distinct proof that repeating their same-component normalization does not change the value. Later loop hoisting remains a separate consumer.

M02 migrates repeated integer and exact-bit float constant writes. The legacy raw-register availability solver is deleted. A candidate is removable only when its before/after `StorageId` is unchanged, every reachable scalar `ValueId` leaf equals the candidate constant, and reference/native-payload components are already absent on every path. This last condition preserves the VM assignment side effects that release a reference and finalize host-owned binary state. Equal-value phis, direct link/swap mappings and ordered TRACE paths can therefore prove even though raw register or source order differs; different phis, signed-zero bit changes and hidden cleanup fail closed. A cheap syntactic demand filter only decides whether to ask the proof service and never authorizes deletion. Migrated consumers share one immutable proof session per fixed-point epoch, and retained-byte metrics are refreshed after lazy query materialization.

M03 migrates repeated `NULL`. The legacy raw-register all-view availability solver and its blanket TRACE guards are deleted. A candidate must be an exact classified non-signalling `NULL`, its pre-instruction target must resolve to a known `StorageId`, and every reaching leaf of all eight tracked components must already be `ABSENT`. This proves that deletion skips no scalar/payload destruction, reference release or native-payload finalization. Distinct write-once absent leaves may agree through a phi, direct aliases follow storage identity, and explicit TRACE records remain ordered observations. An intervening component write or unknown storage/value leaf fails closed. The proof deliberately uses pre-instruction storage: VM `NULL` clears the value reached by a register but does not change the register's link topology.

M04 migrates exact same-storage copies. The canonical `rxop_same_storage_copy_is_noop()` contract covers `copy`, `icopy`, `fcopy`, `scopy`, `dcopy`, `acopy` and `bcopy`: when both operands denote the same physical value storage, the VM path performs no write, allocation, signal or externally observable effect. This conditional contract is more exact than the opcode's general signal contract; different-storage `bcopy` remains conservative. Identical physical register operands prove the old floor directly. Distinct registers require equal pre-instruction `StorageId`, or storage phis with the same unique write-once leaf. LINK-established and agreeing-phi aliases can therefore prove, while divergent, different or unknown storage fails closed. Explicit TRACE records remain; an event after a semantically empty copy does not make that copy observable.

The admitted first transformation panel is deliberately narrower than the fact en-

gine:

- `icopy`, `fcopy` and strict-comparison uses of `scopy` may be redirected only when the typed source view is unchanged on every path and every may-reaching destination observation can be redirected atomically; decimal copy remains excluded from that established consumer pending decimal-lifetime proof, although `dcopy` itself is now a proven total non-signalling operation;
- all seven exact same-storage copy families use the M04 conditional metadata and StorageId proof, while nonidentity full-value copies remain excluded pending ownership, payload, attribute, flag and cleanup proof;
- repeated integer/bitwise-equal float constants use the M02 storage/value and hidden-cleanup proof; repeated `null` uses the M03 known-storage and all-component-absence proof;
- repeated metadata-admitted one-register XTOY derivations use the generic dominated-success repetition proof;
- a repeated one-register `itos` is removed only under the storage-identity, integer/string-component and numeric-context proof described above;
- unreachable instructions, TRACE records and source-step records are removed only when the entire procedure CFG is complete; and
- dead-result deletion remains disabled because a nominal numeric destination write may release hidden reference or native-payload state even when its numeric result is dead.

Every admitted rewrite removes at least one executable instruction and inserts none, so fixed-point iteration terminates by a strictly decreasing instruction count. `rxas -d` prints per-candidate accept/reject reasons and a per-procedure summary. `-n` bypasses both the local and whole-procedure optimizers.

NR-18 adds one reverse direction for surviving `icopy`/`fcopy` records. For an immediately adjacent, metadata-classified, non-signalling, context-neutral, single-component producer, the proof service may return an immutable plan that retargets operand 1 from a disposable local temporary to the copy's final local and deletes the copy. The exact producer result must have only the typed copy as a sparse direct or dependent use. Both storages must be local, known and unaliased; the producer must not read the final storage through another register mapping; and TRACE/register metadata, calls, opaque uses, asynchronous handlers and source-address observations reject the candidate.

Scalar producers such as load and integer comparisons release reference and native-payload state, whereas icopy/fcopy write only their typed component. Producer forwarding therefore also proves that every component cleared by the producer is already absent in both the discarded temporary and the final destination. Fresh local entry storage supplies a known empty state; arguments, globals, aliases and unknown or previously populated storage fail closed. This hidden-cleanup condition is part of opcode metadata and prevents typed liveness alone from changing lifetime behaviour.

Copy rewrites proved from one graph may be applied in the same iteration only when their complete physical source/destination register pairs are disjoint. Their substitutions then commute and cannot invalidate one another's SSA/use proof. Procedures newly admitted through exact jump-table edges always receive reachability cleanup. Global typed-value analysis is additionally bounded to one million queue-item x liveness-word cells for those procedures; above the bound, rxas -d reports scope=reachability-only. This preserves the linear high-yield cleanup without turning large generated dispatch procedures into quadratic assembly work. Procedures without resolved indirect tables retain the established NR-27 behavior without that bound.

Integer compare/branch fusion is also a whole-procedure proof-service consumer. Canonical opcode metadata maps each IEQ/INE/IGT/IGTE/ILT/ILTE plus BRT/BRF form to BEQ/BNE/BGT/BGE/BLT/BLE, including normalized immediate/register operand order. The immutable plan requires the compare and branch to be consecutive executable instructions in one CFG block, all three opcodes to be total and context-neutral, and the local result storage to be unaliased. A source may share that storage only when it is the same physical result register and component SSA supplies the exact pre-write integer ValueId; the fused branch then reads that incoming value rather than the materialized Boolean. Every reference/native payload cleared by materializing the Boolean must already be absent. The result integer ValueId may have only the branch read and, under the selected TRACE policy, any number of precisely matching result events between compare and branch. The immutable plan records and revalidates those events, deletes them atomically with the compare, and leaves unrelated same-boundary events present. Live non-trace uses, metadata and caller windows reject fusion; unrelated TRACE events remain. Call-window handling now resolves the exact local bounds when the count has a constant integer ValueId, otherwise conservatively extends to the highest local.

Classified fixed-arity and zero-argument calls expose their complete caller interface through explicit operands; only range-call forms add an implicit base/count local window. The proof follows the compare result through copy/derived values and phis and rejects only when that identity is visible in the resolved window or the query fails closed. Counted CALLs had deliberately unknown signal contracts in the K04c baseline. Their former retry edge turned an otherwise constant count into an unknown phi and widened the window. K04d retires that nonstandard, unused continuation and adds propagated-call partial-state metadata for the remaining skip/handler/unwind failure paths. The old bounded recursive read-before-kill solver and its keyhole rules have been removed.

`test_rxop_metadata` is generated from `op_table` and both complete sidecars so instructions cannot be added without mechanically aligned effect and signal entries. The signal sidecar records the integer fused branches as non-signalling, decimal literal load as plugin-signalling and decimal copy as total/non-signalling, while one- and two-register integer-to-float conversion and float comparison are also non-signalling in the current VM handlers. The legacy `FGT/BRT` to `FGTBR` shortcut is not selected until the fused float opcode receives an explicit signal contract and equivalent component proof.

Attribute/register-view cleanup also retains one small mechanical keyhole. A full `copy` already copies the VM value status word, so `copy rA,rB` followed immediately by `acopy rA,rB` is reduced to the full copy for hand-written or legacy RXAS. This is the closed K06 classification, not a proof-service consumer. Opcode metadata records full `COPY` as reading/writing every component and `ACOPY` as reading/writing only `RXOP_COMPONENT_ATTRIBUTES`; both are non-signalling. The VM handlers confirm `copy_value()` assigns `status.all_type_flags` before `ACOPY` would repeat that exact assignment. The declarative rule captures the same source/destination pair and permits no intervening executable instruction. `test_rxop_metadata` locks the subset/non-signalling invariant, while the focused optimizer fixture locks different-source, different-target and executable-gap negatives. Moving this exact local algebra into CFG/SSA would add analysis cost without adding a correctness fact. Compiler-generated RXAS should not emit that pair for typed payload access. A future explicit flag-copy operation would use `acopy`, but there is no current compiler use case. Typed payload copies are `icopy`, `fcopy`, `scopy`, `dcopy`, and `bcopy`; `bcopy` copies only the binary payload, not public/compiler/library status flags.

Duplicate link/typed-copy/unlink reads and equivalent one-based linkattr1 reads are no longer keyhole authorities. One immutable proof plan validates both exact triples, proves the full typed-copy component mask equivalent, and rewrites the later link to copy the first detached value before deleting its original copy/unlink. An attribute plan also proves the candidate slot is in range so deletion cannot suppress an OUT_OF_RANGE signal. Calls through aliased arguments, divergent source phis, changed source/detached components, changed count/reference generation, different owners or slots, component writes and unproved signal paths reject. Register metadata and TRACE reads of the unchanged first detached value may remain ordered while the redundant second executable read is removed. The twelve former syntax-expanded rules have been deleted.

Cancelling swap pairs are likewise owned by the proof service rather than the legacy bounded NO_HAZARD keyholes. A pair-key demand filter only asks the question. The immutable proof verifies the exact physical operands, dominance, restored pre-first/post-second StorageId bindings and every intervening raw-register use from the shared use index; relevant component reads/writes, register metadata, TRACE, call windows and opaque state reject. Different registers that already name the same storage are an exact identity case, while a disjoint intervening permutation or unrelated external effect need not block deletion. The consumer has no source-queue distance limit and revalidates the complete plan before deleting either instruction.

Potential signals remain explicit CFG edges. A source-linear normal/skip spine may cross a known static signal when its handler policy is still the inherited entry policy: resumption reaches the restoring swap, while branch, return, unwind and terminal actions leave the current frame's temporary mapping. A handler installed, merged or clobbered in the procedure rejects. Uses reachable from the asynchronous-handler root also reject even when their records lie outside the lexical interval. Ordinary control splits and unknown signal-name sets remain closed. This is narrower than treating signals as invisible and more capable than requiring one basic block.

NR-09 may encode two adjacent or metadata-separated swaps as one ordered four-operand swapn. Repeating the same unordered physical pair composes to identity algebraically for every input mapping and is deleted by the same K01 proof. Component SSA independently composes all overlapping swapn pairs in operand order into final input-state sources; it does not interpret them as a par-

allel assignment. Larger arbitrary permutation simplification remains a separate future rewrite consumer.

Signal support adds action-aware and dynamic-name forms:

- `sigcalla proc(),"NAME"` installs a handler that returns an internal action marker interpreted by the VM as `skip` or `fail`; the unused non-standard `retry` action has been retired
- `sigpush "NAME"` saves the current handler entry for `NAME` on the current frame's handler stack
- `sigpop "NAME"` restores the most recent saved handler entry for `NAME`
- `sigbrv label,rSignal,"NAME"` installs a branch handler that wraps the raw interrupt object as a Level B `.signal` value in `rSignal` before branching
- `signal rName` raises a signal whose name is read from a string register
- `signal rName,rPayload` raises a dynamic-name signal with a payload object

Unknown dynamic names raise `INVALID_SIGNAL_CODE`. Literal `signal "NAME"` forms are still assembled directly against the static signal table.

Optimiser rule operands use lowercase `r` for a captured register. Uppercase `R`, `G`, and `A` match literal local/global/argument register numbers. The assembler uses this to express rules such as `inc r0 -> inc0` without adding mnemonic-specific C code to the optimiser engine.

Legacy rules retain three inline operand pairs for source compatibility. New superinstruction rules select a variable-length operand-pattern side table; matching and output generation iterate the whole pattern, and captured values are stored in a dynamically grown map rather than ten fixed slots. The wide `cnop` regression exercises nine captured input and output operands through this path.

Opcode arity is likewise defined by the complete format string rather than a three-operand ceiling. RXAS parsing, RXBIN writing and reading, disassembly, link relocation, VM decoding, metadata inspection, compiler emission, and the `rxc ASSEMBLE` path all iterate that declared operand sequence. The NR-09 large-instruction batch uses forms up to eight operands; adding another arity does not require a new fixed-width transport structure.

The NR-09 fused instructions are catalogued in `docs/reference/rxas/instructions/12-large-ins`
They preserve the ordered state changes and signals of their component sequence, including an intermediate write where that write is part of the public instruction

contract. Compiler-owned rewrites may instead choose a result-only form when the removed temporary has no source-level observation.

The NR-14 frozen-PARSE instructions are catalogued on the same reference page. `parsewords3`, `parsepos2`, and `parsewords3d` encode complete exact plans in their opcode identities; `parseplan` carries a compiler-generated versioned descriptor in an ordinary string-constant operand. RXBIN 007 writers set `RXBIN007_FEATURE_FROZEN_PARSE` when any of the four opcodes is present. Readers reject an image that uses one without that feature bit, and older readers reject the unknown feature bit, so a provisional image cannot be silently decoded with different semantics. The descriptor adds no RXBIN section or relocation form.

The NR-15 native-stem family is catalogued under arrays, attributes, references, and objects. `steminit`, `stemget`, `stemset`, `stemreset`, `stemget2`, `stemset2`, `stemsize`, `stemkeyat`, and `stemvalueat` use ordinary register operands and set `RXBIN007_FEATURE_NATIVE_STEM`. The feature declares the instruction contract only: the receiver's private hash-table bytes remain ordinary process-local VM value state and are never serialized as an RXBIN section. Two-segment forms stream `left || "." || right` during lookup and materialize a canonical key only for a proved new insertion.

`typeof`, `istype`, and `asserttype` are object-contract operations. Compiler generated code uses them for object casts/tests/introspection; scalar `typeof`/`is` cases are folded earlier by `rxc`.

Interface default methods use that same path. The assembler does not introduce new opcodes for them; it simply carries `META_MEMBER` kind `method final` and emits the interface method body as an ordinary procedure. The VM method registry then decides whether `srcmethodsel` should bind `class.member` directly or fall back to the interface's emitted default-body procedure.

At the current Level B stage, the runtime selection rule is:

- each provider row may carry an optional resolved class-side `§match` or `§match.member` procedure
- `srcfprocsel` causes the VM to call that effective `match` on every candidate with the same argument list that will later be passed to the selected factory
- if no explicit `match` exists, the candidate behaves as if it had an implicit `match` returning `1`

- candidates with score `<= 0` are rejected
- the highest positive score wins
- tied highest scores break alphabetically by fully qualified concrete class name

B.6 6. Core Socket Instructions

Core TCP sockets are exposed as normal RXAS mnemonics generated from `binutils/include/rxops.h`. They use VM-managed integer handles rather than OS file descriptors or pointers, so programs cannot accidentally retain a native socket after the VM context is freed. All socket instructions are `FLG_OPT_BARRIER` because they perform external I/O or mutate context-owned handle state.

The current instruction surface is intentionally raw TCP with optional client TLS connect support:

- `socknew rSock`
- `sockclose rRc,rSock`
- `sockconnect rSock,rHost,rPort`
- `sockconnecttls rSock,rHost,rPort`
- `sockbind rSock,rHost,rPort`
- `socklisten rRc,rSock,rBacklog`
- `sockaccept rClient,rServer`
- `sockshutdown rRc,rSock,rHow`
- `sockstarttls rRc,rSock,rHost`
- `socksend rBytes,rSock,rText`
- `socksendb rBytes,rSock,rBin`
- `sockrecv rText,rSock,rMax`
- `sockrecvb rBin,rSock,rMax`
- `sockpending rBytes,rSock`
- `socktimeout rRc,rSock,rMillis`
- `sockblocking rRc,rSock,rFlag`
- `socknodelay rRc,rSock,rFlag`
- `sockkeepalive rRc,rSock,rFlag`
- `sockpeer rText,rSock`
- `socklocal rText,rSock`

- `sockstatus rStatus,rSock`
- `sockerror rText,rSock`

`sockconnect`, `sockconnecttls`, and `sockbind` keep the socket handle in their first operand and record the result in the handle's status slot; use `sockstatus` or `sockerror` immediately after those operations. The higher-level `rxsocket.rexx` wrapper turns these into function return codes for ordinary Level B code.

`sockconnecttls` is the portable client TLS connect primitive. It connects to `rHost:rPort`, starts TLS before any application bytes are exchanged, and uses `rHost` for SNI and certificate name verification. `sockstarttls` remains the lower-level true STARTTLS primitive for protocols that exchange clear-text bytes before TLS; backends that cannot upgrade an existing connection in place return a negative unsupported status instead of reconnecting behind the caller.

Both TLS instructions are present in all builds. When no TLS backend is compiled in, they record a negative socket status; `sockstarttls` also returns that code in `rRc`. Backends are selected at CMake configure time. Fresh builds default to `CREXX_ENABLE_TLS=NETWORK` on Apple platforms, `CREXX_ENABLE_TLS=OPENSSL` on non-Windows Unix-like platforms, and `CREXX_ENABLE_TLS=SCHANNEL` on Windows. The Network backend uses Network.framework, Security.framework, CoreFoundation.framework, and the system trust store for `sockconnecttls`. The OpenSSL backend supports both direct TLS connect and true STARTTLS. The SChannel backend uses Windows SChannel/SSPI and the Windows trust store, and supports both direct TLS connect and true STARTTLS.

B.7 7. Core Channel Instructions

Core concurrency and future transport providers use one transport-neutral RXAS family. Level B classes must use these instructions; there is no parallel RXPA task-start/task-wait ABI and providers do not add provider-specific opcodes.

The current public instructions are:

Opcode	RXAS form	Outputs	Inputs
650	`chanopen rStatus,rChannel,rProviderType,rRequiredCapabilities,rOptions`	status, channel	provider type `.int`, capability mask `.int`, canonical RXCV `.binary`
651	`chanstart rStatus,rTicket,rChannel,rEnvelope,rWaitMicroseconds`	status, ticket	channel `.int`, RXCV envelope `.binary`, relative wait `.int`
652	`chanwait rStatus,rCompletion,rChannel,rWaitMicroseconds`	status, completion	channel `.int`, relative wait `.int`
653	`chancancel rStatus,rChannel,rTicket,rReason`	status	channel/ticket `.int`, RXCV reason `.binary`
654	`chanclose rStatus,rChannel,rMode`	status	channel `.int`, mode `.int`
659	`chanrelease rStatus,rChannel,rTicket`	status	channel/ticket `.int`

The two-output forms require distinct output registers. Outputs may alias input registers because the VM snapshots inputs before writing failure defaults or results. A failed operation returns a status and leaves the companion output at `0` or empty binary; the instructions have `RXSC_NONE` signal contracts.

All six instructions are `FLG_OPT_BARRIER` with classified opaque effects. Optimizers must not move, duplicate, fold or remove them. TRACE, debugger and profiler identity remains the canonical opcode. The handlers are outlined/cold under the current `profile-20` policy.

Any image containing one of these opcodes requires `RXBIN007_FEATURE_CHANNELS` (`1 << 3`). RXAS emits it, RXLINK retains the validated union, and RXBIN readers reject either a channel opcode without the bit or an unsupported feature bit. `chanrelease` also requires `RXBIN007_FEATURE_CHANNEL_RELEASE` (`1 << 7`); its combined requirement is `0x88`. RXAS/RXDAS round trips preserve the six mnemonics and operand order.

Observation does not release request authority. `chanrelease` succeeds only after successful terminal observation, destroys the provider request, and invalidates the ticket while leaving the channel and saved completion binary valid. Pending/unobserved requests return status `10`; later use of a released ticket returns stale status `13`. Release is provider-neutral and never needs another ticket. See the life-

cycle reference for cleanup-failure retry rules.

rProviderType is one mutually exclusive implementation code; rRequiredCapabilities is a bit mask. Type 0 is invalid; type 1 is the core local-thread provider, type 2 is the isolated-process provider, type 3 remains the reserved open-host provider, type 4 is the reusable byte endpoint and type 5 is the structured child-process provider. Extension codes begin at 65536. Type 1 advertises bounded admission (0x0001), cancellation (0x0002), deadlines (0x0004) and completion-order observation (0x0008). Unknown required bits fail instead of being ignored. A private validated registration seam and fake-provider conformance fixture exist; no public provider-plugin ABI is implied.

Configuration, envelopes, reasons and completions are canonical versioned RXCV documents. The current implementation includes the complete value tree, typed task register images, provider-owned deadlines/scopes, Level B classes and Level G lowering over sealed task procedure, transferable receiver and .taskwork factory targets. Assembler .task1, .task2 and .task3 metadata carry an 80-byte binding; RXAS resolves its image, callable, signature and adapter identity and RXBIN validation rejects malformed or stale bindings. Raw channel/ticket integers are execution-local capabilities, not transferable values. Concurrent HTTP/TLS is a Level G library consumer, not an opcode or provider type. Exact status codes, RXCV layouts and lifecycle rules are in CREXX_CONCURRENCY.md and the per-instruction RXAS reference.

C

cREXX Linker Architecture (`rxlink`)

The `rxlink` tool combines one or more assembled `.rxbin` modules into a linked image that keeps module boundaries while deduplicating compatible constant-pool leaves into one shared pool.

C.1 Purpose

Use `rxlink` when you want to:

- bundle a root module with the providers it needs
- turn a loose module set into one deployable linked image
- shrink downstream `rxcpack` / wrapped artifacts by removing duplicated pool entries
- optionally strip source/TRACE debug metadata from deployable images

This is now the normal native-packaging route for the shipped drivers too: `crexx`, `crxc`, `rxpp`, and related wrapped tools link a deployable image first and then pass that linked image to `rxcpack`.

`rxlink` is not a replacement for the VM loader. The output still contains multiple module records, and `rxvm` still performs the final runtime link/load work.

C.2 Native-provider requirements

Selected META_PROVIDER records are preserved and checked against their matching META_FUNC signatures. rxlink rejects the same callable when inputs associate it with different stable providers, return types, or argument signatures. Requiring-module provenance is retained for diagnostics.

Use -p requirements-file, or PROVIDERS requirements-file in a control file, to write the packaging projection without loading native code:

```
CREXX-RXPA-REQUIREMENTS 1
required<TAB>provider<TAB>callable<TAB>return-type<TAB>arguments<TAB>
    module
```

This is the authoritative input to crexx -native. Each provider value is the canonical artifact stem: native packaging prefers <provider>.a/.lib and falls back to the historical <provider>_static.a/.lib name. A native package therefore does not maintain a second hand-written provider list.

C.3 Packaged bytecode autoload hints

Selected META_AUTOLOAD records are runtime convenience metadata. rxlink remaps and preserves their callable-symbol and packaged-filename-stem references, but does not use them to discover linker inputs. Explicit link inputs and ordinary symbol/signature selection remain authoritative.

It is safe to retain a hint whose provider has been selected into the linked image. At runtime the VM links all explicit and embedded modules first and follows only hints attached to callables that are still unresolved. Thus a deployable linked image does not reopen or duplicate a dependency merely because the original separately compiled consumer recorded its package stem.

C.4 Module initializers

Selected META_INITIALIZER records are runtime contract metadata. rxlink checks that each record names a local META_FUNC/procedure with .void return and no arguments, remaps its symbol and procedure references into the output pool, and

preserves metadata order. Initializers do not make their procedures exports or selection roots.

Linked images retain their individual module records. This is essential for the once-per-mutable-module-instance state machine: multiple initializers in one module retain declaration order, while the VM can still initialize and poison each module overlay independently. Source and inline stripping never removes initializer metadata.

C.5 Output Format

RXLINK reads and writes only RXBIN 007. The complete toolchain moved atomically; 006 inputs are rejected and rebuilt rather than decoded through a compatibility path. The format and semantic graph design are in `RXBIN_007_SEMANTIC_GRAPH.md`.

The 007 output is one fixed-width sectioned container with a module directory, module instruction ranges, shared constant/canonical metadata data, and one image-wide semantic graph. The current shared-pool/module record stream is not carried forward.

Milestone 1 retains metadata-driven module selection. For the selected modules RXLINK rebuilds canonical text-backed type/member/callable/factory nodes, preserves declaration origin, assigns new image-local dense IDs, rewrites graph-bearing instruction references, and rebuilds every adjacency, name, declaration, dispatch, callable, factory, and provider index. It never concatenates module-local indexes.

Callable imports compiled without their provider may carry `.unknown` in a return or parameter type. When the selected defining module is linked, RXLINK refines only those unresolved imported fields from the definition. Known types, arity, and parameter flags remain strict contracts, and duplicate definitions remain errors.

Source-import class stubs must therefore preserve `expose` on writable arguments when they emit their imported `META_FUNC` projection. The compiler's callable registry and call ABI already retain that property; dropping it only from the emitted projection creates a false reference-mode conflict when RXLINK rebuilds the graph. RXLINK continues to reject a genuine value/reference mismatch and

adopts the selected definition's complete descriptor and procedure reference only after the import contract matches.

The common `rxbin` graph library owns structural merge, remap, validation, and fast rule-neutral traversal. Future language-policy adapters may decide inheritance, assignability, override/default conflicts, and provider selection without embedding those language rules in RXBIN itself.

C.6 Selection Model

Inputs are read as a container stream, so one input file may contain a linked multi-module container or concatenated standalone library containers. Module selection then happens in this order:

1. apply `OMIT`
2. apply `INCLUDE`
3. apply explicit `ROOT`
4. if no roots were chosen, select modules containing `main`
5. if there is still no root, select modules from the first input file
6. walk imports, `srcfprocsel` interface references, and interface relationships to pull in required providers
7. reject duplicate selected exports

Selectors match by:

- full module name
- basename
- filename stem with trailing `.rxas` removed
- `input_path::member`

C.7 What The Linker Reads From Metadata

`rxlink` does not need all metadata equally:

- `proc_head` is used to find procedures and detect `main`
- `expose_head` is used to discover imports and exports

- `meta_head` is scanned for `META_INTERFACE` and `META_IMPLEMENTS` so interface definitions and implementations pull each other in
- the instruction stream is scanned for `srcfprocsel` descriptor strings so modules referenced only through runtime interface-factory lookup are still retained
- the instruction stream is scanned for `srcmethodsel` member names. Because the receiver's concrete class can be known only at runtime, modules that expose or declare a matching member name are selected conservatively.
- selected class/interface contracts are then checked against callable descriptors so a provider with the right name but wrong return or argument signature is rejected before output is written.

The linker preserves the metadata chain in output because the VM and tooling still consume it at runtime. This includes packaged bytecode autoload hints; they are transport metadata, not linker discovery instructions.

Task metadata is also runtime contract metadata. `.task1`, `.task2` and `.task3` entries carry an 80-byte sealed binding containing image digest, callable id, signature digest and the optional adapter callable slot. Because RXLINK rebuilds and renumbers the semantic graph, it must regenerate these bindings from the linked graph rather than copy module-local bytes. Kind `2` relocates the receiver `from_channel` factory and kind `3` relocates the `.taskwork.run` method. Missing, malformed, stale or signature-incompatible bindings fail the link; they are never weakened to procedure-name dispatch. An imported task call contains the deterministic 80-byte relocation placeholder but does not duplicate the defining module's `META_TASK_TARGET`. The RXBIN writer therefore reseals both the metadata binding and every matching placeholder constant across all selected constant pools. This is required for separately compiled task-method clients: copying the defining module's old seal would retain the wrong final graph digest, while leaving the use-site placeholder would fail the `RXTB` magic check.

C.8 Constant-Pool Rewriting

Leaf constants are deduplicated across selected modules when their serialized bytes match:

- STRING_CONST
- BINARY_CONST
- DECIMAL_CONST
- FLOAT_CONST

Structured constants are rewritten into the shared pool with all referenced offsets updated:

- procedures
- exposed register/procedure entries
- metadata entries
- instruction operands that point into the pool

Instruction rewriting derives its operand count and kind from the canonical variable-length opcode signature. Linker scans and remaps therefore have no three-operand format switch; wide instructions retain every inline operand while constant and graph references are rewritten normally.

C.9 Strip Support

Current conservative strip support has two independent axes:

- CLI: -s
- control file: STRIP SOURCE
- CLI: -i
- control file: PRESERVE INLINE / STRIP INLINE

STRIP SOURCE removes:

- META_SOURCE_STEP
- META_TRACE_EVENT

Trace-event metadata is source-level debugging metadata. Without source-step anchors, classic TRACE value events are not coherent enough to keep in a deployable stripped image, and they may still expose variable names, compound names, constants, or live values. Keep the linked image unstripped when TRACE, RXDB source stepping, or source-level diagnostics are needed.

Inline-body metadata is different from runtime contract metadata. It is useful to libraries consumed by `rxc`, but it is not needed once a final linked image has been built. `rxlink` therefore strips `META_INLINE` by default. Use `-i` or `PRESERVE INLINE` only for diagnostic/tooling builds that need to inspect the inline transport after linking.

It intentionally does not remove runtime contract metadata such as:

- `META_CLASS`
- `META_ATTR`
- `META_INTERFACE`
- `META_IMPLEMENTS`
- `META_MEMBER`
- `META_INITIALIZER`
- sealed `.task1`/`.task2`/`.task3` bindings

That keeps interface/class dispatch and metadata-aware tooling behaviour stable while still removing source text/file path payloads and source-level TRACE value metadata.

C.10 Control Files

Supported directives are:

- `INPUT path`
- `ROOT selector`
- `INCLUDE selector`
- `OMIT selector`
- `OUTPUT path`
- `MAP path`
- `STRIP SOURCE`
- `STRIP INLINE`
- `PRESERVE INLINE`

C.11 Testing Guidance

When changing rxlink, keep three layers of coverage in mind:

1. format tests: shared-pool/shared-module record layout
2. behavioural tests: linked images run correctly through product rxvm; add the concrete rxbvm/rxtvm dispatch contract only for execution-image or dispatch-sensitive changes
3. toolchain tests: rxdas can still disassemble linked images, including stripped ones

For broader confidence, the runtime _opt path is now wired through linked images in normal ctest coverage. For a focused rerun, use:

- `cmake --build <build-dir> --target qa-prep-linked-opt-runtime`
- `ctest --test-dir <build-dir> -L linked_opt --label-exclude '^performance-measureme`
  `--output-on-failure`
- `cmake --build <build-dir> --target linked_opt_sweep`

The first two commands expose preparation and execution as separate stages; linked_opt_sweep is the convenience target that performs both in that order.

Be conservative with stripping. If a proposed change removes anything beyond the current source-level debug set (META_SOURCE_STEP and META_TRACE_EVENT), verify both runtime contract lookup and metadata introspection in rxvm before assuming it is safe.

List of Tables

Index

ISBN 978-94-039116-2-5