CREXX 1.0.0 Beta 1

Rexx readability with a visible compiled path.

- Release 1 baseline for early users
- Level B language contract
- 45 minutes of syntax, 15 minutes of architecture

Paper record

A paper-record slide deck distilled from the live THE/cREXX symposium demo.

The Hour Was Built For Rexx Programmers

Start with the machine, then spend most of the time in code.

15 min architecture

Driver, compiler, assembler, linker, VM, libraries, native packaging.

45 min syntax

Typed Level B examples: parse, stems, ADDRESS, signals, objects, factories.

One invitation

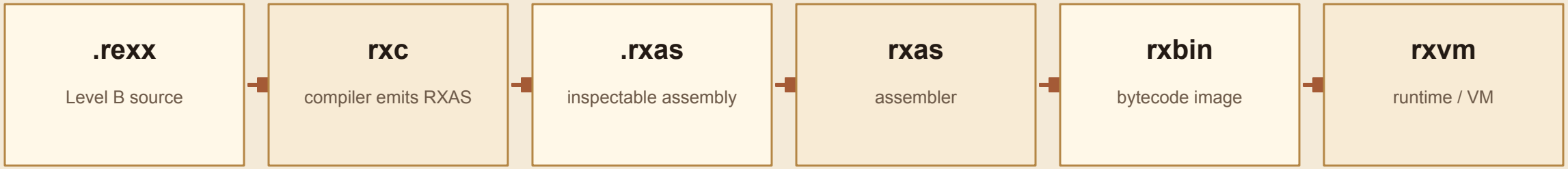
Keep Rexx readability, add visible stages and deployment machinery.

Session promise

The talk deliberately uses small programs: each one is a concept you can run, inspect, and explain.

The Toolchain Is Visible

The compiled path is not hidden behind a single opaque command.



Driver

`crexx` orchestrates common build/run flows.

Libraries

Core functions and class surfaces are mostly cREXX code.

Native edge

Plugins and native packaging keep integration open.

Beta 1 Contract

A useful beta has a surface, a toolchain, and honest limits.

Area	Included	Beta note
Language	Level B surface, typed declarations, procedures, objects	Future language levels remain direction
Tools	crexx, rxpp, rxc, rxas, rxlink, rxvm/rxvme, rxdas, rxcpack	rxdb remains experimental
Platforms	Linux x64, Windows x64, macOS arm64/x86_64	Windows binaries unsigned in Beta 1
Quality	Release candidate run: 1029 passing tests	Beta means early integrator feedback is wanted

Release link

https://github.com/adesutherland/CREXX/releases/tag/v1.0.0-beta.1

First Run: Familiar Shape, Compiled Path

`say`, words, and strings still look like Rexx.

10-first-run.rexx

```
options levelb
import rxfnsb

say "CREXX 1.0.0 beta 1"
say "source -> RXAS -> RXBIN -> VM"

pipeline = "compile assemble execute"
say "pipeline words:" words(pipeline)
say "middle stage:" word(pipeline, 2)

return 0
```

output

```
CREXX 1.0.0 beta 1
source -> RXAS -> RXBIN -> VM
classic shape, compiled path
pipeline words: 3
middle stage: assemble
```

Presenter cue

Use F5 in the live deck to show the generated RXAS for the same source.

Parse And Word-Oriented Code Still Matter

The syntax remains legible enough to talk through at the keyboard.

parse / words

```
line = "Ada Lovelace <ada@example.org>"
parse var line first last "<" email ">"

say "first:" first
say "last :" last
say "email:" email

phrase = "Rexx programmers like readable code"
do i = 1 to words(phrase)
  say i ":" word(phrase, i)
end
```

output

```
first: Ada
last : Lovelace
email: ada@example.org

1 : Rexx
2 : programmers
3 : like
4 : readable
5 : code
```

Structured Control With Traceable Flow

A good symposium demo is editable: this is where `trace normal` can be typed live.

14-control-flow.rexx

```
/* Demo prompt: type trace normal here. */
n = 11
select
  when n = 10 then kind = "ten"
  when n > 10 then kind = "larger than ten"
  otherwise kind = "odd"
end
say n "is" kind

total = 0
do i = 1 to 5
  total = total + i
end
```

output

```
11 is larger than ten
sum 1..5 = 15
kept 1
kept 2
kept 3
kept 5
kept 6
```

Talk cue

Type `trace normal` into this example, rerun, and let the VM show its work.

Stems Are Real Data Structures

Dotted names and computed keys both stay in the Rexx mental model.

15-stems.rexx

```
stock = .stem()
stock.tea = 12
stock.coffee = 7
stock["biscuits"] = 5

items = "tea coffee biscuits"
do i = 1 to words(items)
  name = word(items, i)
  say left(name, 10) stock[name]
end

stock.coffee = stock.coffee + 3
```

output

```
Stem values can use dotted names
tea: 12

They can also use computed keys
tea        12
coffee     7
biscuits   5

coffee after delivery: 10
```

Procedures Make The Boundary Explicit

Value and exposed references can be explained without magic.

16-procedures.rexx

```
x = 10
say "bump value:" bump(x)
say "after value:" x

call bump_ref x
say "after expose:" x

bump: procedure = .int
  arg value = .int
  value = value + 1
  return value

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

output

```
start: 10
bump value: 11
after value: 10
after expose: 11
```

ADDRESS Captures Host Output

The Rexx host-command idea becomes typed arrays for input, output, and error.

17-address.rexx

```
input_lines = .string[]
output_lines = .string[]
error_lines = .string[]

input_lines[1] = "rxvm"
input_lines[2] = "rxc"
input_lines[3] = "rxas"

address path "sort" input input_lines,
  output output_lines error error_lines

say "rc:" rc
do i = 1 to output_lines.0
  say output_lines.i
end
```

output

```
rc: 0
rxas
rxc
rxvm
```

Signals Are Structured Conditions

Compiler exits and structured handling give classic Rexx flow a sharper form.

18-signals.rexx

```
handled = ""

do
  signal other "bad customer number"
  say "not reached"
on signal other as problem
  handled = problem.name() ":" problem.message()
end

say handled
say "program continues"
```

output

```
OTHER:bad customer number
program continues
```

Objects Add Interfaces Without Losing Readability

Factories make selection a language-level idea instead of a hand-written switch.

19/20 objects and factories

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

fileasset: class implements .asset
  *: match
    arg name = .string
    if right(name, 4) = ".txt" then return 100
    return 0

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

output

```
one = .asset("log.txt")
two = .asset("memo")

file:log.txt
cache:memo
```

The Beta 1 Takeaway

The language remains recognisably Rexx, while the execution path becomes inspectable.

- Start with the `crexx` driver
- Inspect generated RXAS early
- Use small typed examples to learn Level B
- Treat Beta 1 as an invitation to try, test, and integrate

Link

CREXX 1.0.0 Beta 1: https://github.com/adesutherland/CREXX/releases/tag/v1.0.0-beta.1