CREXX

Under the Hood:

Inside CREXX – A Guided Tour

Checklist

- Language & Frontend
- Execution & Runtime
- Language Evolution
- Performance
- Wrap-up

Table of Contents

Inside CREXX – A Guided Tour

Table of Contents

Inside CREXX – A Guided Tour

CREXX Architecture

Still REXX at the language level

Classic REXX	CREXX with a VM Backbone
Evaluate expressions	Execute bytecode instructions
Interpreter-driven flow	Register-based VM flow
Variables resolved at runtime	Registers allocated in stack frames

Goal: **Performance + control without losing REXX semantics**

CREXX Execution Model

- **Register-based** VM (values in registers)
- **Direct** instruction execution on values
- Execution within **stack frames**
- Instructions act on **registers**

CREXX opens up the VM

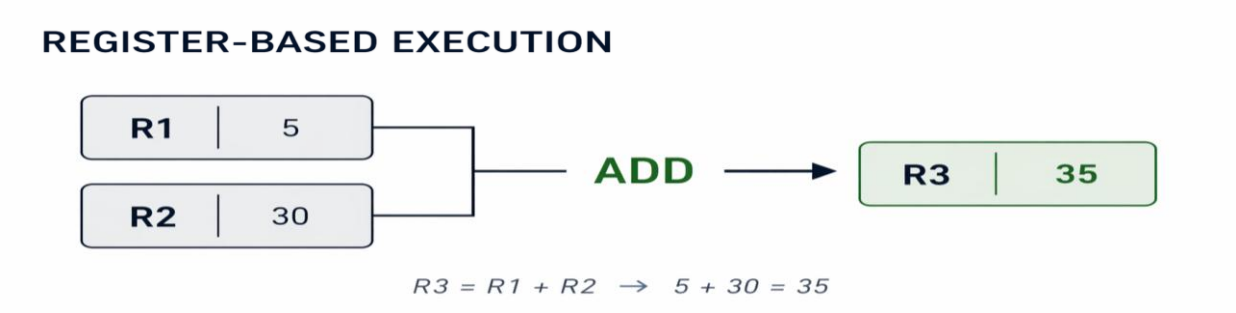
Runtime & Extensibility

- **Arithmetic** provided by plugins
- Numeric model is **selectable** (fast vs exact)
- Behavior defined at **runtime**, not fixed

The runtime is configurable

Bytecode Execution

REXX: c = a + b
VM: ADD R3, R1, R2



Compiled to simple instructions — executed directly.

Table of Contents

Inside CREXX – A Guided Tour

Q-Series Motivation

Q-Series: Quote-Aware String Functions

- Extends REXX with **quote-aware parsing**
- Ignores delimiters **inside quoted strings**
- Supports **nested structures** and **multi-character tokens**

Targets:

- source parsing
- RXPP preprocessing
- structured text handling

Key idea:

- Standard string functions see characters.
- Q-Series sees **structure outside quotes**.
- Parse structure, not characters

Q-Series Core Capabilities

- **QPOS()**
  Quote-aware search outside quoted regions
- **QSPLIT() / QWORD-family**
  Quote-safe tokenisation and word handling
- **QFINDPAIRS() / QEXTRACTPAIR() / QEXTRACTALL()**
  Match and extract nested blocks
  → supports **single- and multi-character delimiters**
- **QSTRIPCOMMENT() / QREMOVEALL()**
  Remove comments or delimited regions without breaking quoted text

QPOS() is foundation — all others build on the same quote-aware model.

Quote-Blind vs Quote-Aware

- **POS() vs. QPOS()**

```
s = 'say "a,b", x'

pos(',', s)   --> 7
qpos(',', s)  --> 11
```

- **WORD() vs. QWORD()**

```
s = 'say "hello world" next'

word(s, 2)   → "hello"
qword(s, 2)  → "hello world"
```

Classic functions scan text.
Q-functions respect structure.

Q-Series Motivation

Q-Series: Quote-Aware String Functions

- Extends REXX with **quote-aware parsing**
- Ignores delimiters **inside quoted strings**
- Supports **nested structures** and **multi-character tokens**

Targets:

- source parsing
- RXPP preprocessing
- structured text handling

Key idea:

- Standard string functions see characters.
- Q-Series sees **structure outside quotes**.
- Parse structure, not characters

Q-Series & RXPP

- Distinguishes **code vs. quoted text**
- Prevents macro expansion inside strings
- Improves:
  - preprocessing correctness
  - macro handling
  - parsing robustness

Enables structure-aware preprocessing

Back on the Rails

Formatting

Classic REXX Manual formatting

```
do i = 1 to name[0]
    total = qty.i * price.i

    line =
       right(i, 3) || " " ||
       left(name.i, 12) || " " ||
       right(qty.i, 5) || " " ||
       right(format(price.i, 6, 2), 6) || " Total=" ||
       right(format(total, 6, 2), 6)
    say line
end
```

- Verbose and hard to read
- Formatting logic scattered across functions
- Easy to introduce alignment bugs
- Mixing **layout + logic**

What FSAY adds

```
do i = 1 to name[0]
    total = qty.i * price.i
    fsay "{i:>3} {name.i:<12} {qty.i:>5} {price.i:>6.2} Total={total:>6.2}"
end
```

{i:>3} → right-align index in width 3
{name.i:<12} → left-align name in width 12
{qty.i:>5} → right-align quantity
{price.i:>6.2} → fixed width, 2 decimals
{total:>6.2} → formatted computed value

Yes, it looks familiar… that's because we borrowed the good ideas — and made them REXX."

In classic REXX, formatting is something you *construct*
In CREXX, formatting is something you *declare*.

fmtmask Picture-Style Formatting

Uses a **format mask** instead of inline specifiers

- Text slots: **X**
- Numeric slots: **9**
- Literals stay in place: labels, punctuation, currency symbols
- Good for:
  - fixed-width reports
  - forms
  - classic business-style output

```
say fmtmask("Name: XXXXXXXX  Qty: 999  Price: 9999.99",,
      "Fred", 12, 64.31)
```

The mask defines the layout first, the values fill it.

fmtmask Curated Examples

fmtmask: Picture-Based Formatting (COBOL-style)

1. Structured Output ➡️ Fixed layout, aligned fields

```
say fmtmask("Name: XXXXXXXX  Qty: 999  Price: 9999.99","Fred", 12, 64.31)
```

2. Currency Formatting ➡️ Currency symbols embedded in the mask

```
say fmtmask("US: $$$9.99  EU: €€€9.99", 12.45, 10.50)
```

3. Mixed Layout + Data ➡️ Labels and values combined cleanly

```
say fmtmask("Order[999]: XXXXX", 27, "Widget")
```

4. Edge Handling ➡️ Overflow and truncation become visible

```
say fmtmask("Qty: 999", 12345)
say fmtmask("Name: XXXXX", "Maximilian")
```

The layout is defined first — the data must fit into it.

Three Formatting Approaches — Same Result

- **Classic REXX**

```
line = right(i,3) || " " ||
   left(name.i,12) || " " ||
   right(qty.i,5) || " " ||
   right(format(price.i,6,2),6)
say line
```

- **CREXX fsay**

```
fsay "{i:>3} {name.i:<12} {qty.i:>5} {price.i:>6.2"
```

- **CREXX fmtmask**

```
say fmtmask("999 XXXXXXXXXXXX 99999 9999.99",i, name.i, qty.i, price.i)
```

All three produce the same output: **1 Fred 3 12.45**

Why Practical OO Support?

- Objects encapsulate data and behavior via methods (put/get)
- Structure larger programs
- Bind data and operations together
- Encapsulate reusable components
- Move beyond flat procedure collections

Enough OO to organise code, without OO complexity

Objects in CREXX

HashMap: class

***: factory**
    arg expected = 1024
    val = stemcreate(expected, 'hashmap')
return

put: method
    arg key=.string, value=.string
return stemput(val, key, value)

get: method
    arg key=.string
return stemget(val, key)

CREXX Object Usage

```
order = .HashMap()

order.put("item", "Coffee")
order.put("price", 3.20)
order.put("quantity", 15)
order.put("currency", "EUR")

say order.get("price")
```

```
"item"     → "Coffee"
"price"    → 3.20
"quantity" → 15
"currency" → "EUR"
```

CREXX OO What Is There

Advantages

- **Structured access (no raw stems)**
- **Encapsulation via methods**
- **Reusable components**
- **Clearer program structure**

Design Approach:

- Composability, not inheritance
- ~~no interfaces~~
- NEW: Interfaces (recent addition)

OO focused on structure and usability

Using Objects in CREXX

- map = HashMap()
  ```
  map.put("key", "value")
  say map.get("key")
  ```
- .binds method calls to objects
- objects integrate naturally into normal CREXX code
- suitable for practical components such as maps

OO feels simple and REXX-like

Design Choice: Focused OO model

- No inheritance by design
- Interface-based composition
- Runtime selection of implementations
- Explicit, predictable behaviour
- Designed for practical abstraction
- Optimised for clarity and control

The goal is clarity and composability over complexity

What RXPP Does

- Parses code line by line
- Expands macros (##define)
- Resolves variables (##set, {var})
- Processes conditions (##if / ##else)
- Includes external logic (##include, ##use)

RXPP Macro Example

Code:

```
##define CUBE(x)          {x*x*x}

a=cube(3.14)
say "Cube of PI is="a
```

After preprocesing(RXPP):

```
a=3.14*3.14*3.14
 say "Cube of PI is="a
```

Macros are expanded before compilation — zero runtime cost.

RXPP Macro (Overview)

What is a macro?

- A **compile-time text substitution rule**
- Replaces code patterns **before execution**

How is it defined?

- **##define NAME**(x) {replacement using x}

  NAME → macro name
  arg → parameter (optional)
  {...} → replacement text

- Example:
- ##define SQUARE(x) {x*x}

How is it expanded?

- Occurs during preprocessing (rxpp)
- Arguments are **inserted literally** into the replacement
- Expansion is **recursive**
- Several Pre-Process options available:
  ##cflags def nset niflink n1buf n2buf 3buf nvars nmaclist includes

Execution Flow

RXPP (## directives)
↓
Generated CREXX code
↓
CREXX runtime (VM execution)

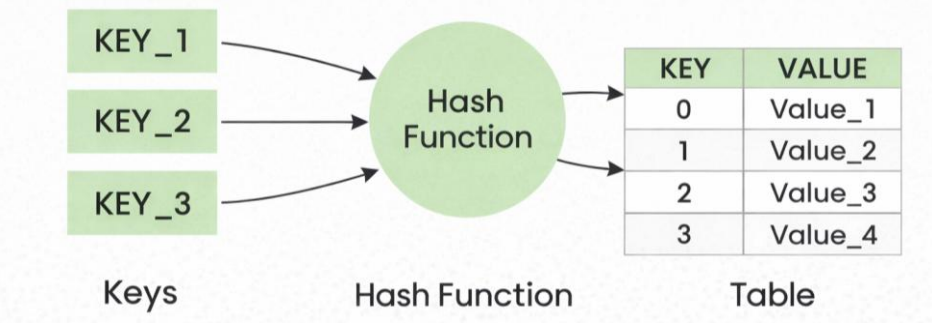
RXPP transforms code — the VM executes it

Table of Contents

Inside CREXX – A Guided Tour

Efficient Lookup

- **ARRAYs:** ordered → scan
- **STEMs:** key → lookup
  - No linear scans
  - Near-constant lookup time
  - Predictable performance



Arrays are for walking data.
Stems are for finding data.

PARSE

- **Conventional PARSE syntax remains downward compatible**

- **Classic templates still work:**
  - words
  - literals
  - absolute / relative positions
  - placeholders (.)

- **CREXX adds:**
  - shorter syntax
  - built-in result collection (INTO)
  - post-processing (UPPER / LOWER / TRIM)
  - diagnostics (LOG / TRACE)

Classic PARSE remains intact; CREXX extends convenience and visibility.

PARSE New syntax forms

PARSE: New Forms

- **Direct source form**

  parse **fred** first second

- **Explicit WITH form**

  parse **var fred** with 1 first ',' second ',' third

- **INTO form**

  parse **into abc** var fred first second

- Source can be given directly
- **WITH** makes source and template separation clearer
- **INTO** variable-name collects parsed results into an array

Less ceremony, plus optional result capture.

PARSE Processing Options

- **UPPER** convert parsed text to upper case
- **LOWER** convert parsed text to lower case
- **TRIM** trim parsed results
- **INTO var-name** push parsed items into an array

Examples

```
parse upper fred a b c
parse lower fred a b c
parse trim fred a b c
parse into abc var fred first second
```

PARSE can normalize and collect results directly.

SELECT as Dispatch Mechanism

code

```
select
  when b = "" then do
    /* monadic use */
  end
  when words(a) = 1 then do
    /* scalar a with vector b */
  end
  otherwise do
    /* dyadic use */
  end
end
```

Now supported: SELECT **city** ... WHEN **"Munich"**

~~Coming soon: SELECT **city** ... WHEN **"Munich"**~~

Table of Contents

Inside CREXX – A Guided Tour

Plugins

- **Maps** *Robin Hood hashing — high performance*
- **Trees** *structured / ordered data*
- **Sockets** *networking support*
- **Regex Lite** *lightweight pattern matching*
- **+** additional extensions

Focus shifted to language features (RXPP, PARSE, extensions)

Numeric Models

[Hardware]

✓ fast

✗ rounding errors

Native INT64 + FLOAT64

Uses CPU arithmetic (hardware)

- Lowest overhead
- Best throughput
- Good for general workloads

Range:

~10^{18} (INT64),
~10^{308} (FLOAT64)

[Decimal]

✓ exact

✗ slower

Arbitrary-precision decimal

Implemented via Decimal Plugin

- Exact decimal semantics
- Predictable results
- Good for finance and correctness-critical tasks

Range:

controlled by **NUMERIC DIGITS**

Arbitrary precision is powerful — but most workloads fit comfortably within INT64 and FLOAT64.
Unless you count the atoms of the universe (~10^{80} atoms)

Precision vs Performance

Exact decimal arithmetic ensures correctness,
but native INT64 and floating-point cover most practical ranges with significantly better performance.

- Decimal → **maximum correctness**
- Hardware → **maximum performance**

0.1 + 0.2 = 0.30000000000000004 ❌

0.1 + 0.2 = 0.3 ✓

- Choose based on **workload requirements**

I'm probably biased — I come from a time where efficiency meant counting bytes and eliminating every unnecessary instruction.

Same language, two execution models: performance or precision

Compiler Exits

- Hook into the **compiler pipeline**
- Add **custom commands**
- Extend or adapt **existing syntax**
- Transform source → **canonical form** before execution
- Can call **external code** when needed

Compiler exits are written in REXX — no external tool required.

EXECIO Example

Content (simplified!)

```
EXECIO 10 DISKR input (STEM data.)
```

➞ becomes internally:

```
rc = _execio(10, 'DISKR', 'input', data)
```

Behavior

- Recognizes **classic syntax**
- Validates structure and parameters
- Rewrites into a canonical function call (internal / external)
- Optional **post-processing hook**
- Integrates seamlessly with the runtime

Same mechanism can introduce entirely new language constructs.

Back on the Rails

CREXX Build Pipeline

CREXX Compiler Driver (crexx.exe)

- Drives the full processing pipeline
- Accepts files and options in any order
- Detects RXPP sources automatically
- Supports:
  - preprocess
  - compile
  - assemble
  - native build
  - execute

Example:

crexx --nodecimal myrexx.rexx
RXPP → Compile → Bytecode →[Native] → Execute:

decimal | nodecimal

Not just a compiler — a pipeline driver.

CREXX Compiler Driver

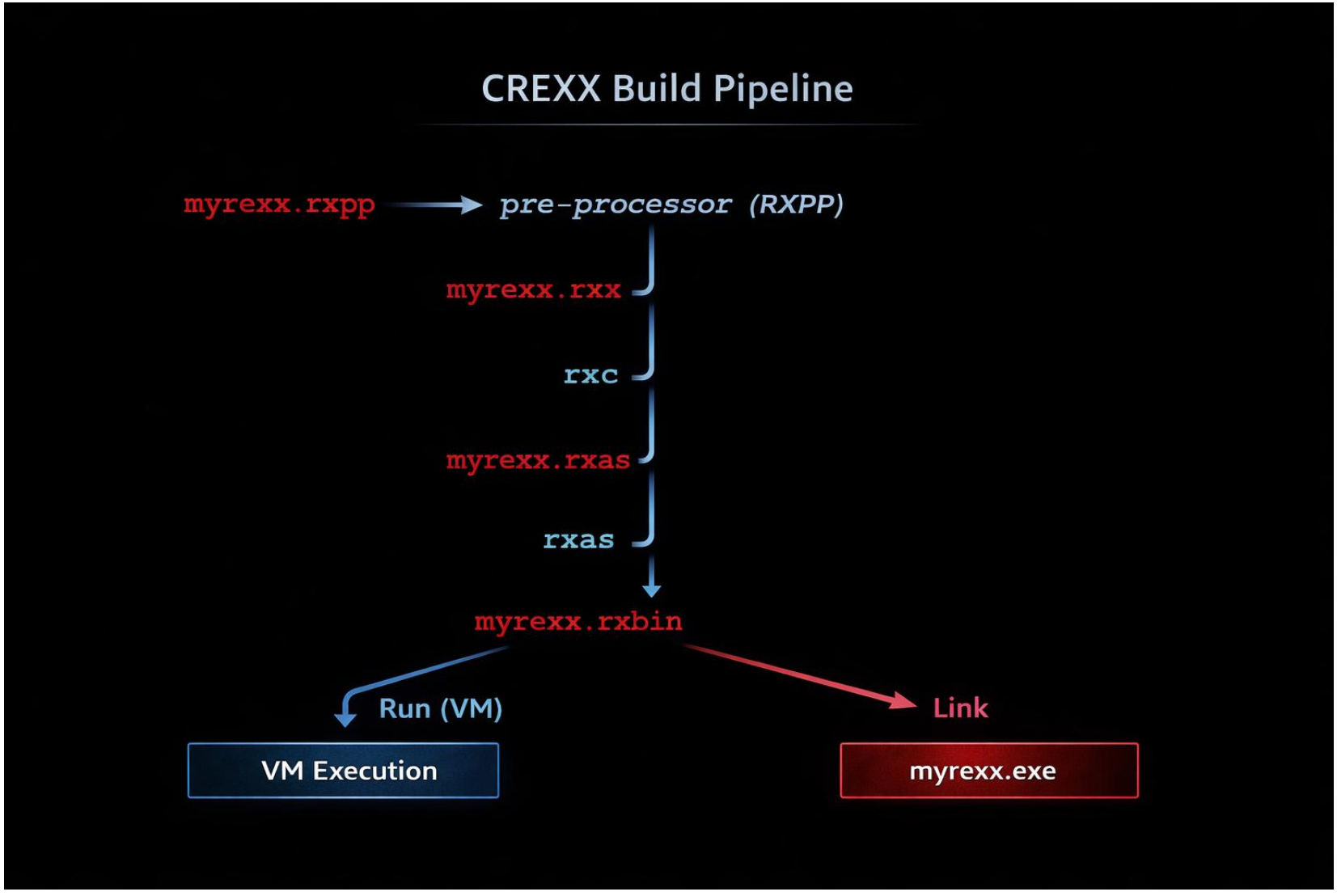


Table of Contents

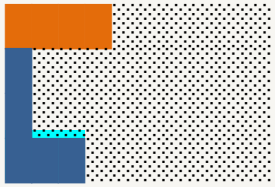
Inside CREXX – A Guided Tour

Significant Performance Improvements

- Optimized BIFs (up to 70%)
- Compile-time code optimisations
- INLINE procedures

Faster execution without changing your code.

BIF Optimisation

Operation	New Time	Factor	Visual
SUBSTR(..., 1, 45)	0.493 s	3.91×	
SUBSTR(..., 19, 5)	0.517 s	1.13×	
SUBSTR(..., 52, 5)	0.544 s	1.17×	
SUBSTR(..., 9, 60)	0.919 s	2.67×	
STRIP(..., B)	1.867 s	5.67×	
STRIP(..., L)	1.399 s	4.48×	
STRIP(..., T)	0.686 s	8.48×	
POS("only")	0.246 s	3.37×	
POS("about")	**0.316 s**	**19.64×**	
POS("about", 70)	0.253 s	4.83×	
LASTPOS("only")	0.446 s	12.90×	
LASTPOS(„middle“)	0.443 s	7.85×	
TOTAL	**7.683 s**	**5.96×**	

String operations dominated runtime — we removed the bottleneck.

Compile-time Optimisations

These are examples of compile-time code optimisations:

- **Constant folding** → compute at compile time
- **Constant propagation** → replace variables with known values
- **Dead-code pruning** → unreachable or unused code
- **Copy elision** → avoid unnecessary data copies

Inlining (Compile-Time Optimisation)

New: Procedure Inlining

- Embeds the procedure body at the call site
- Runs in multiple passes (iterative inlining)
- Followed by a **pruning/cleanup**

Why it matters

- Exposes more code to **constant folding**
- Removes call overhead entirely (no stack, no argument setup)
- Enables further **dead-code elimination**
- Helps the compiler produce **smaller, simpler hot paths**

Inlining unlocks further optimisation — each pass enables the next.

Table of Contents

Inside CREXX – A Guided Tour

CREXX Today

- Still REXX at the surface
- Structured and extensible underneath
- Multiple execution models (performance vs precision)
- Open extension points (plugins, compiler exits)
- One system — many use cases

Kept the language — expanded what happens behind it.

On the surface, nothing changed —

underneath, CREXX
turns REXX into a platform.