# CREXX
# Application Programming Guide

**The CREXX team**

June 30, 2024

**THE REXX LANGUAGE ASSOCIATION**
**CREXX Programming Series**
**ISBN 978-90-819090-1-3**

# 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 978-90-819090-1-3

ISBN 978-90-819090-1-3

9 789081 909013 >

# Contents

**The crexx Programming Series** **i**

**Typographical conventions** **ii**

## I Guide 2

**1 Overview of the toolchain** **3**

**2 Running crexx on Linux and macOS** **4**

2.1 Running cRexx entails compiling 4

2.2 A first program - Hello World 4

2.3 Using built-in functions 5

2.4 Building a standalone executable 6

**3 Running crexx on Windows operating systems** **8**

**4 Running crexx on VM/370CE** **9**

**5 Intralanguage calls** **10**

5.1 At compile time 10

5.2 At runtime 10

**6 Interlanguage calls** **11**

**7 Tracing and Debugging** **12**

7.1 An example debugging session 12

## II Reference 13

**8 crexx Compiler** **14**

8.1 Command Line Options 14

8.2 Inline Assembler 14

8.3 Optimizer 14

**9 CREXX Assembler 16**
9.1 Overview 16
9.2 Program Structure 16
9.3 Input/Output 16
9.4 Character sets 16
9.5 Command Line Arguments 17
9.6 Optimizer 17
9.7 Assembler Directives 17
9.8 Examples 18
9.9 In-line assembly 18
9.10 Troubleshooting 20
9.11 Reference 20

**10 CREXX Disassembler 21**
10.1 Input/Output 21
10.2 Command Line Arguments 21
10.3 Example 21

**11 CREXX Debugger 25**
11.1 Command Line Options 25
11.2 Runtime Options 25

**12 CREXX C Packer 26**
12.1 Command Line Options 26

**List of Tables 27**

# The CREXX Programming 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.

| | |
|---|---|
| **User Guide** | This guide is meant for an audience that has done some programming and wants to start quickly. It starts with a quick tour of the language, and a section on installing the CREXX translator and how to run it. It also contains help for troubleshooting if anything in the installation does not work as designed, and states current limits and restrictions of the open source reference implementation. |
| **Application Programming Guide** | The Application 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. |
| **The CREXX 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 REXX publications:

- Body text is in this font
- Examples of language statements are in a `keyword` or **bold** type
- Variables or strings as mentioned in source code, or things that appear on the console, are in a `typewriter` 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'
```

# About This Book

The crexx language is a further development, and variant of the Rexx language[1]. This book aims to document the workings of this implementation and serves as reference for users and implementors alike.

## Application Programming Guide

This *Application Programming Guide* focuses on documenting the tools delivered with crexx, from a usage perspective. The technical background and design documentation is in the *crexx VM Specification* (this includes information on how to build the tools from source), while the programming language itself is defined in the *crexx Language Definition* document.

This document includes practical examples, best practices, and code snippets that illustrate how to perform common tasks, such as setting up the environment, handling data input and output, processing user input, and interacting with other software components.

## Audience

This application programming guide is aimed at developers who want to use crexx to create an application or integrate it into an existing system.

## History

**mvp** This version documents the Minimally Viable Product release, Q1 2022 and is intended for developers only. It documents the toolchain for crexx level B, which is a typed subset of Classic Rexx.

**f29** First version, Git feature [F0029]

[1] Cowlishaw, 1979

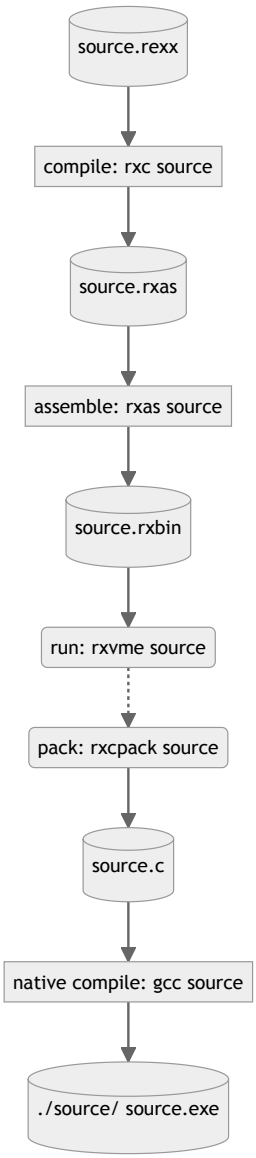
# Part I

# Guide

1

# Overview of the toolchain

In the binary distribution of CREXX for your platform, the main toolchain and a number of utilities are included; all programs are statically linked into their own executable, and no shared object libraries or dll's are needed.




Source can be edited with any text editor[2] of your liking. The sourcefile needs to have an file extension of `.rexx` and contains (Unicode, UTF-8) text. The `rxc` CREXX compiler produces a text file which will have a file extension of `.rxas`. The next file in this sequence is produced by the `rxas` assembler and is a binary `.rxbin` file. This file is executable by the CREXX `rxvme` virtual machine.

If you choose to compile a series of `*.rxbin` files to a native executable, the `rxcpack` program produces a `.c` file, which can be compiled by any C compiler toolchain.

In the following chapters the detailed workflow for the supported platforms is documented.

[2] Vi, Emacs, Xedit, VS Code, CLion, Eclipse, etc.

2

# Running cRexx on Linux and macOS

Linux and other Unix-like operating systems like Apple macOS behave in an identical way when compiling, linking and running a cRexx program.

## 2.1 Running cRexx entails compiling

All Rexx scripts you run with cRexx are compiled by `rxc` into Rexx assembler code (.rxas) and then assembled into an .rxbin file, which contains the Rexx bytecode for execution by the Rexx Virtual Machine `rxvm`. During both the compliation and assembler steps optimization of the code takes place. The rxvm executable takes care of linking separately compiled modules together and executing them, so one function can find another.

## 2.2 A first program - Hello World

Let's say you have a Rexx exec you would like to run. To not have any surprises, it is of the *hello world* kind. We have a file called hello.rexx, containing:

```
/* rexx */
options levelb
say 'hello cRexx world!'
```

```
hello cRexx world!
```

When cRexx level `C' (for `Classic') is available, the `options levelb' (on line 2) statement can be left out; for the moment, level B is all we have, and the compiler will refuse to compile without it.

With all our cRexx executables on the PATH, we only need to do:

```
rxc hello
rxas hello
rxvm hello
```

to see `hello cRexx world!' on the console, as include above this, run from the included program. Like all the programs in the cRexx documentation, these programs are compiled and run from the included source by the process that builds the document.

It might be a good idea to make a shell script to execute these three programs in succession, and perhaps call it `crexx'. But take into account that this really is a very simple case, in which no built-in functions are called. We can have a look at the generated rexx assembler (hello.rxas) file:

```
/*
 * cREXX COMPILER VERSION : crexx-f0049
 * SOURCE                 : hello.rexx
 * BUILT                  : 2024-06-30 18:26:56
 */

.srcfile="hello.rexx"
.globals=0

main() .locals=1
   .meta "hello.main"="b" ".void" main() "" ""
   .src 3:1="say 'hello cRexx world!'"
   say "hello cRexx world!"
   .src 3:25=""
   ret
```

and you can see here that the compiler actually has generated a `say' assembler instruction for the Rexx `say' instruction. (Assembler became a whole lot easier with Rexx assembler.) But we did not yet call any function.

## 2.3 Using built-in functions

Most Rexx programs use the extremely well designed built-in functions. Now with these functions written in Rexx, and not hidden in the compiler somewhere, we must tell it to import those from the library where we put them earlier during the build process. Let's say we want to add a display of the current weekday to our hello program. This will now be:

```
/* rexx */
options levelb
import rxfnsb
say 'hello cRexx world!'
say 'today is a' date('w')
return 0
```

Never mind the import statement, which you will not need when cRexx `Classic' level C is available. But in level B, we need this, because we need the flexibility it affords our plans for the future. See more about `import` on page 10.

We must tell the compiler where to find the signature of the date() function, so it can check if we call it in the correct way, with the right parameters. This is done with the -i switch, which points to the directory containing the library - which is called `library', by the way.

```
rxc -i ~/crexx-build/lib/rxfns hellodate
```

[todo]

The assembler runs unchanged, because it trusts the compiler to have checked

if the called function really sits in that library, and has the right parameters - the right code to call it has been generated.

```
rxas hellodate
```

To run it, we can employ the `rxvme` executable - this one is extended with linked-in versions of all the functions in the library:

```
rxvme hellodate
```

which yields:

## 2.4 Building a standalone executable

It is possible to build a standalone executable of this program. It is possible to run your compiled Rexx program, e.g. from a USB stick, without ever installing crexx, on the same OS and instruction set architecture.

For this, we need the next set of commands, expressed as a Rexx exec:

```
/* rexx compile a rexx exec to a native executable */
/* Classic Rexx and NetRexx compatible             */
crexx_home='~/crexx-build'
if arg='' then do
  say 'exec name expected.'
  exit 99
end
parse arg execName'.'extension
if extension<>'' then say 'filename extension ignored.'
'rxc  -i' crexx_home'/lib/rxfns' execName
'rxas' execName
'rxcpack' execName crexx_home'/lib/rxfns/library'
'gcc -o' execName,
'-lrxvml -lmachine -lavl_tree -lplatform -lm -L',
crexx_home'/interpreter -L'crexx_home'/machine -L',
crexx_home'/avl_tree -L'crexx_home'/platform'  execName'.c'
```

This exec is delivered in the source tree, bin directory.

The exec works by compiling the Rexx program specified (again without the .rexx file extension) to an .rxbin Rexx bytecode file, which is then serialized to a C source file, containing the cRexx Virtual Machine and the library rxbin files, by the rxcpack command. It is a rather peculiar looking C source, but nevertheless it will compile to a working executable, which is done by the last step in the exec, here using the gcc compiler. And you will be able to run it without the overhead of checking, compiling, tokenizing to bytecode and linking, so it will be quite fast:

```
hello cRexx world!
today is Saturday
./hello  0.00s user 0.00s system 61% cpu 0.009 total
```

3

# Running CREXX on Windows operating systems

4

# Running CREXX on VM/370CE

5

# 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, because 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.

## 5.1 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 to verify if it is possible to call the called code: it must be present in excutable form, and it needs to have the right *signature*[3]. The compiler will not automatically compile a callee of which the source can be located but the executable form is missing; existing systems based on interpreters will happily interrupt their work and tokenize another source file when called; the CREXX `rxc` compiler will not.

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.

## 5.2 At runtime

[3] with signature we mean the combination of parameters and return type

6

# 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:

$$todo : checkrelease$$

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

**RexxSaa** the traditional RexxSAA calling convention can be used for direct interfaces to executables that are designed to function as a Rexx library. In its most simple form, these can return REXX strings to the calling program.

**Generic Call Interface** In this RexxSAA extension, the type and length of the parameters can be specified by the caller[4].

[4]Which is considered unsafe but sometimes the only possibility for programs not designed to be called by CREXX

7

# Tracing and Debugging

## 7.1 An example debugging session

# Part II

# Reference

8

# CREXX Compiler

## 8.1 Command Line Options

```
cREXX Compiler
Version : crexx-f0049
Usage : rxc [options] source_file
Options :
-h Prints help message
-c Prints Copyright and License Details
-v Prints Version
-l location Working Location (directory)
-i import Locations to import file - ";" delimited list
-o output_file REXX Assembler Output File
-n No Optimising
```

## 8.2 Inline Assembler

On page 18 the inline assembler function of the CREXX compiler is discussed. This enables the incorporation of `rxas` assembler instructions into a REXX source file.

## 8.3 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 obviate the execution at runtime:

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

```
0.25 should be 0.25
```

```
/*
 * cREXX COMPILER VERSION : crexx-f0049
 * SOURCE                 : fpowtest.rexx
 * BUILT                  : 2024-06-30 18:26:56
 */
```

```
.srcfile="fpowtest.rexx"
.globals=0

main() .locals=1
   .meta "fpowtest.main"="b" ".void" main() "" ""
   .src 2:1="say 0.5**2 'should be 0.25'"
   say "0.25 should be 0.25"
   .src 2:28=""
   ret
```

9

# CREXX Assembler

## 9.1 Overview

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.

## 9.2 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.

## 9.3 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 adresses, 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`).

## 9.4 Character sets

The input file is assumed to be valid UTF8. 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.

## 9.5 Command Line Arguments

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

```
cREXX Assembler
Version : crexx-f0049
Usage : rxas [options] source_file
Options :
-h Help Message
-c Copyright and License Details
-v Version
-a Architecture Details
-i Print Instructions
-d Debug/Verbose Mode
-l location Working Location (directory)
-o output_file Binary Output File
-n No Optimising
```

## 9.6 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` options. When there is any doubt whether any encountered problem is caused by the optimizer, switching it off can help diagnosing the problem.

## 9.7 Assembler Directives

For machine instructions, see the *cREXX VM Specification*. 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
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 and fetch it                     */
ilt r5,r3,0          /* if <0, nothing found, end search    */
brt break,r5         /* word not 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 ... gn`. 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.

**.src** Used to document source lines. This is an optional directive that is added for every source line processed by the `rxc` compiler. It is used for TRACE and SOURCELINE.

**.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.

## 9.8 Examples

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

## 9.9 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*.

[5] When used with `options level b`

```
/* 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:

```
/*
 * cREXX COMPILER VERSION : crexx-f0049
 * SOURCE                 : pow.rexx
 * BUILT                  : 2024-06-30 18:26:56
 */

.srcfile="pow.rexx"
.globals=0

main() .locals=5
   .meta "pow.main"="b" ".void" main() "" ""
   .src 4:1="number = 10"
   .meta "pow.main.number"="b" ".int" r1
   load r1,10
   .src 5:1="power = 2"
   .meta "pow.main.power"="b" ".int" r2
   load r2,2
   .src 6:1="result = 0"
   .meta "pow.main.result"="b" ".int" r3
   load r3,0
   .src 8:1="say 'test IPOW'"
   say "test IPOW"
   .src 10:4="ipow result,number,power"
   ipow  r3,r1,r2
   .src 12:1="say \"result =\" result"
   itos r3
   sconcat r4,"result =",r3
   say r4
   .src 12:22=""
   ret
   .meta "pow.main.number"
   .meta "pow.main.power"
   .meta "pow.main.result"
```

The `.src` directives (intended for trace and sourceline) indicate where the work is done. 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 14 shows that the CREXX optimizer of the compiler can eliminate this code entirely.

The use of Assembler Directives is not allowed in inline assembly, so (as an example) is it not possible to define procedures in an inline assembler block.

## 9.10 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.

### 9.10.1 Adding say statements

It is easy to add `say` statements to your program. Unlike REXX, there is no trace statement for assembler programs. It is possible to disassemble (see page 21 an `.rxbin` module, and reassemble it with added statements.

### 9.10.2 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 page 25.

## 9.11 Reference

10

# cREXX Disassembler

A disassembler reverses the actions of an assembler; where the assembler turns a text file containing unstrictions and directives into a binary executable, the disassembler returns this binary file into its text form[6];in this case it delivers a disassembly which in itself can be re-assembled - and still works.

## 10.1 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.

## 10.2 Command Line Arguments

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

## 10.3 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

/*
 * cREXX COMPILER VERSION : cREXX F0044
```

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

```
 * SOURCE                  : sumLoop1000.rexx
 * BUILT                   : 2023-03-03 23:56:03
 */

.srcfile="sumLoop1000.rexx"
.globals=0

main() .locals=4
   .meta "sumloop1000.rexx.main"="b" ".void" main() "" ""
   .src 4:1="sum = 0"
   .meta "sumloop1000.rexx.main.sum"="b" ".int" r1
   load r1,0
   .src 5:1="do"
   .src 5:4="i=1"
   .meta "sumloop1000.rexx.main.i"="b" ".int" r2
   load r2,1
   .src 5:8="to 100000"
   load r3,100000
l7dostart:
   .src 5:8="to 100000"
   igt r0,r2,r3
   brt l7doend,r0
   .src 6:4="sum = i+sum"
   iadd r1,r2,r1
l7doinc:
   .src 5:4="i"
   inc r2
   .src 7:1="end"
   br l7dostart
l7doend:
   .src 8:1="say \"the sum of the numbers 1 to 100000 is:\" sum"
   itos r1
   sconcat r3,"the sum of the numbers 1 to 100000 is:",r1
   say r3
   .src 9:1="return"
   ret
   .meta "sumloop1000.rexx.main.i"
   .meta "sumloop1000.rexx.main.sum"
```

This program contains the generated assembler as the `rxc` produces it from the Rexx source. What follows is the disassembly from the assembled `.rxbin' file.

```
*******************************************************************************
* MODULE - examples/sumLoop1000
* DESCRIPTION - examples/sumLoop1000

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


* CODE SEGMENT - Size 0x21

.globals=0
.srcfile="sumLoop1000.rexx"

main()          .locals=4
                .meta "sumloop1000.rexx.main"="b" ".void" main() "" ""
                .src 4:1="sum = 0"
                .meta "sumloop1000.rexx.main.sum"="b" ".int" r1
                load r1,0                                   * 0x000000:00c2 Load op1 with op2
                .src 5:1="do"
                .src 5:4="i=1"
                .meta "sumloop1000.rexx.main.i"="b" ".int" r2
                load r2,1                                   * 0x000003:00c2 Load op1 with op2
                .src 5:8="to 100000"
                load r3,100000                              * 0x000006:00c2 Load op1 with op2
                .src 5:8="to 100000"
lb_9:           igt r0,r2,r3                                * 0x000009:0068 Int Greater than op1=(op2>op3)
                brt lb_18,r0                                * 0x00000d:00b4 Branch to op1 if op2 true
                .src 6:4="sum = i+sum"
                iadd r1,r2,r1                               * 0x000010:000f Integer Add (op1=op2+op3)
                .src 5:4="i"
                inc r2                                      * 0x000014:0038 Increment Int (op1++)
                .src 7:1="end"
                br lb_9                                     * 0x000016:00b3 Branch to op1
                .src 8:1="say \"the sum of the numbers 1 to 100000 is:\" sum"
lb_18:          itos r1                                     * 0x000018:00d1 Set register string value from its int value
                sconcat r3,"the sum of the numbers 1 to 100000 is:",r1 * 0x00001a:004e String Concat with space (op1=op2||op3
)
                say r3                                      * 0x00001e:00c6 Say op1
                .src 9:1="return"
                ret                                         * 0x000020:00ad Return VOID
                .meta "sumloop1000.rexx.main.i"
                .meta "sumloop1000.rexx.main.sum"
*******************************************************************************
```

### 10.3.1 Remarks

The zebra fanfold 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.

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

Also, the disassembler affixes the standard instruction documentation, which is the same as generated by `rxas -i`, in a line comment after the instructions.

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.

11

# cREXX Debugger

The debugger is the only program in the toolchain delivered with Rexx as its source code; the 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 very useful tool for debugging low-level problems.

## 11.1 Command Line Options

```
cREXX Embedded VM/Interpreter
Version : crexx-f0049
: Threaded Mode
Usage : rxvm [options] binary_file [binary_file_2 ...] -a args ...
Options :
-h Prints help message
-c Prints Copyright and License Details
-l location Working Location (directory)
-v Prints Version
```

## 11.2 Runtime Options

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

12

# 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.

## 12.1 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-f0049
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 and License Details
-v Version
-o output_file Binary Output File (.c is appended - default is input_file_1.c)
```

# List of Tables

# Index

bct, 18
br, 22
brt, 18, 22
copy, 18
do, 19, 21
end, 19, 21
fndblnk, 18
fndnblnk, 17
iadd, 22
ieq, 18
igt, 22
ilt, 17, 18
inc, 18, 22
ipow, 19
itos, 19, 22
load, 19, 22
options, 4, 5, 14, 19, 21
ret, 5, 15, 19, 22
return, 5, 21
say, ii, 4, 5, 14, 15, 19, 21, 22
sconcat, 19, 22
to, 21

ISBN 978-90-819090-1-3