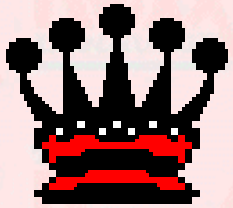
# GCI
# The Generic Call Interface

Florian Große-Coosmann

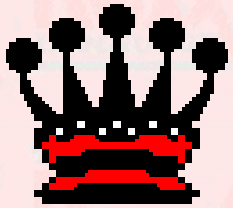
2004 Rexx Symposium
Böblingen-Sindelfingen, Germany
May 5, 2004



# Generic Call Interface

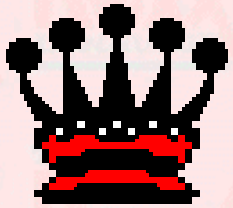
- GCI is an extension of the RxFunc... function package.
- It allows a REXX-only solution for calling external function packages without a wrapper library.
- Its home is:

  http://rexx-gci.sourceforge.net



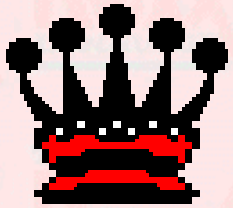
# Goals

- easy to use wrapper tool for packages
- reduction of error response time
- support for rapid prototyping
- flexible programmer's interface
- nearly system independent syntax
- usage completely in REXX
- reduction of the distance to state-of-the-art script languages

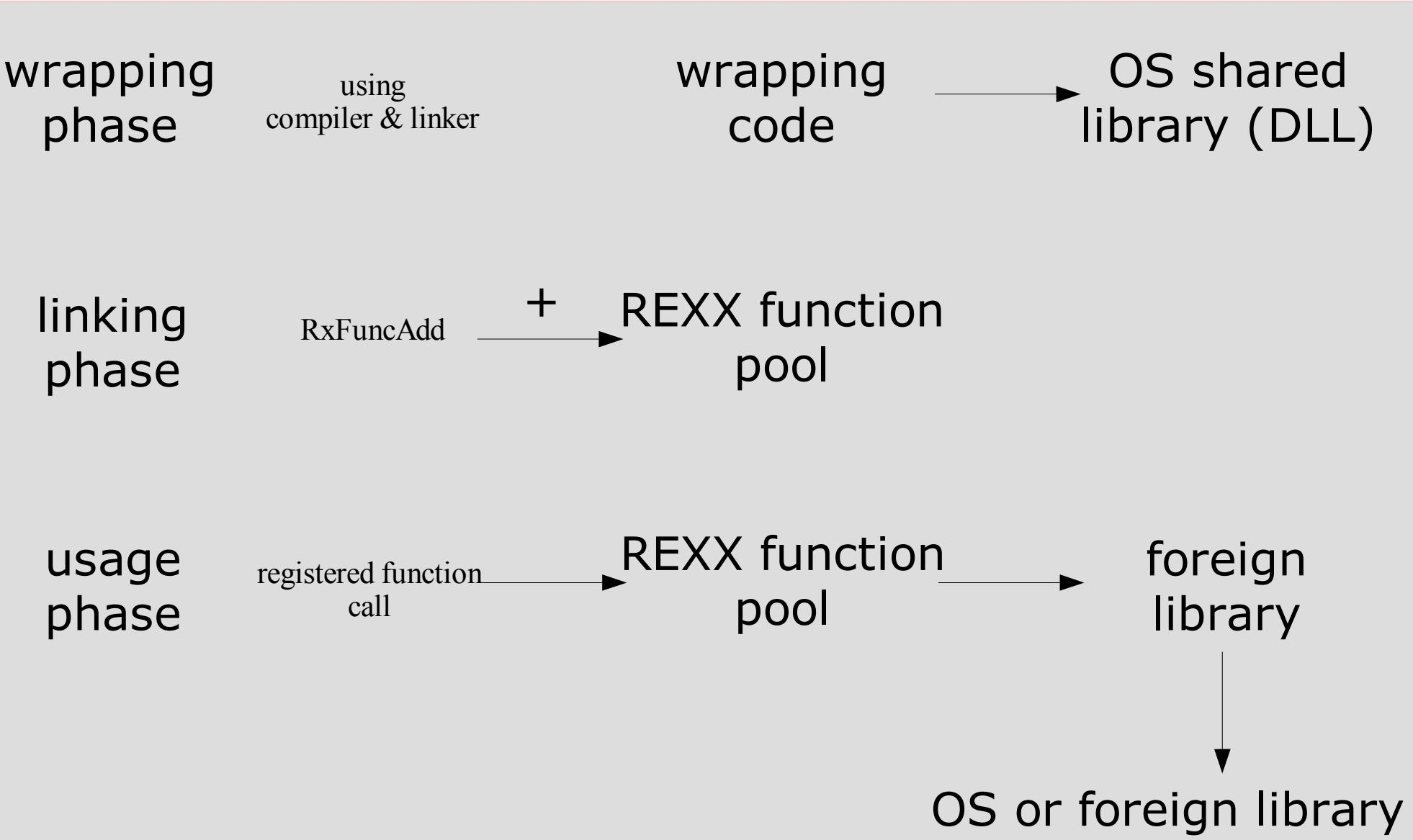


# Overview

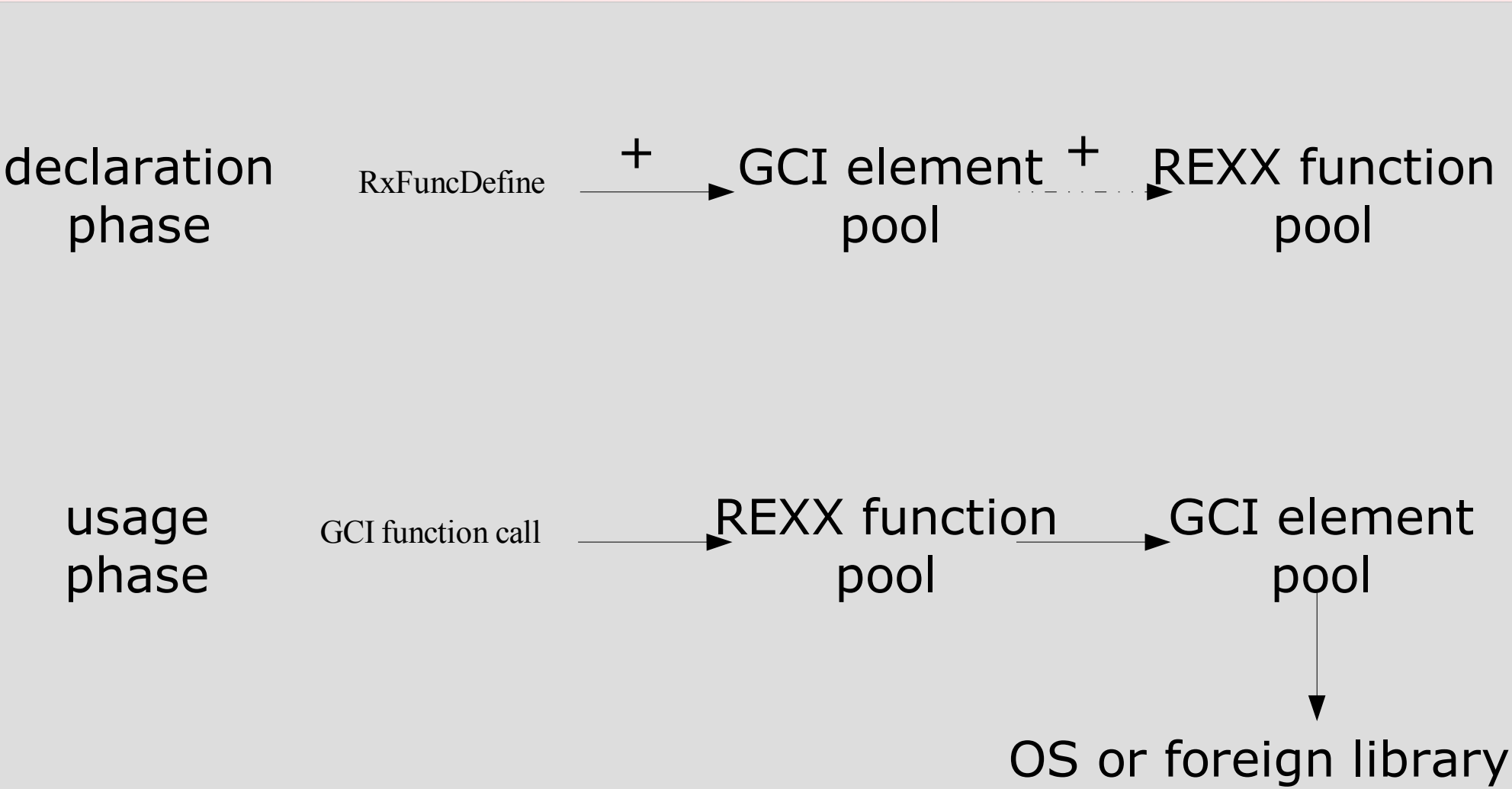
- GCI is a package like many others
- GCI runs on several operating systems
  - Win32, OS/2, unix
  - supports 64 bit systems
- GCI can be compiled for various interpreters
  - Regina, Object Rexx, Classic Rexx, RexxTrans, ...
- GCI is able to be compiled into the core of an interpreter
  - e.g. Regina
- GCI is open source and easy to configure or adapt
- GCI is extensible



# Former Standard



# Design

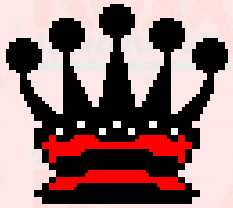

declaration phase
RxFuncDefine
+
GCI element pool
+
REXX function pool
usage phase
GCI function call
REXX function pool
GCI element pool
OS or foreign library

# Short example

```
aStem.calltype = cdecl with parameters as function
aStem.0 = 1                         /* # args */
aStem.1.type = float96              /* arg type */
aStem.return.type = float96
call RxFuncDefine sin, "libm.so.6", "sinl", aStem
/* do some error checking */
do i = 0 to 6
   say "sin(" || i || ") =" sin(i)
   end
/* --> sin(0) = 0.000000000000000000000E+00
 *     sin(1) = 8.414709848078965066646E-01
 *... */
```

# Caveats, Pitfalls, Traps

- Always use strings: CDECL may not be "`CDECL`"
- Stem names should be quoted, too
- Although not forced, names are highly recommended
- "`AS FUNCTION`" doesn't work for complex types
- "`WITH PARAMETERS`" doesn't work for complex types
- not all types are consistent between OSs, e.g. `FLOAT96`



# Short example 2

```
aStem.calltype = "cdecl with parameters as function"
aStem.0 = 1                          /* # args */
aStem.1.type = "float96"             /* arg type */
aStem.1.name = "radians"             /* convenient name */
aStem.return.type = "float96"
aStem.return.name = "sin of the radians"
call RxFuncDefine "SIN", "libm.so.6", "sinl", aStem
/* do some error checking */

do i = 0 to 6
   say "sin(" || i || ") =" sin(i)
   end
```

# Declaration

- The declaration phase is done by RxFuncDefine.
- The fourth parameter is the only difference from RxFuncAdd.
- The fourth parameter's **content** is a stem or a branch, valid values are:
  - `aStem`
  - `aStem.`
  - `aStem.branch`
  - `aStem.branch.`
- The value should be passed as a string e,g, "`aStem.`"

# RxFuncDefine's syntax

```
[RC =] RxFuncDefine(iName, Lib, lName, branch)
```

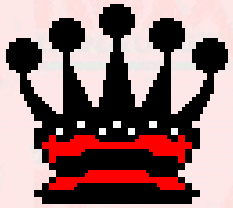
branch elements:

- .CALLTYPE
- .0 = <count arguments>
- .1 /* e.g. .1.TYPE = CHAR8 */
- .2
- ...
- .<count arguments>
- .RETURN

# *Arguments*

Each argument and `return` consists of

- .TYPE = [INDIRECT] <type>
- [.NAME = <convenient name>]
- [.0 = <array or container element count>]
- [.1 = <first container or array element>]
- [.*n* = <last container element>]



# Calltype

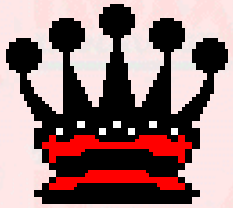
The **calltype** leaf describes the nature of the function

syntax: type [AS FUNCTION] [WITH PARAMETERS]

type: CDECL | PASCAL | STDCALL | <other known types>

- Wrong types may lead to program/system crashes.
- **AS FUNCTION** is convenient, but doesn't allow complex return codes and interferes with error codes
- **WITH PARAMETERS** is convenient, but doesn't allow complex arguments

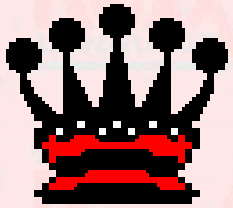
A parameter passing stem is normally used.



# Integer Types

- **Integer type**s are defined by the keyword "`INTEGER`" immediately followed by or blank separated by a bit count. Another type is a plain `integer` using the default integral type.

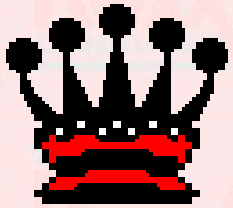
```
- INTEGER 8
- INTEGER16  /* may be equivalent to integer */
- INTEGER 32 /* may be equivalent to integer */
- INTEGER64  /* may be equivalent to integer */
```



# Integer Example

```
aStem.calltype = "cdecl as function with parameters"
aStem.0 = 1
aStem.1.type = "integer"
aStem.1.name = "character"
aStem.return.type = "integer"
aStem.return.name = "uppercased character"
call RxFuncDefine "TOUPPER", "libc.so.6",,
                  "toupper", aStem
/* do some error checking */

say "toupper(ü) =" d2c(toupper(c2d('ü')))
```



# Unsigned types

- **Unsigned type**s are defined by the keyword "UNSIGNED" immediately followed by or blank separated by a bit count. Another type is a plain unsigned using the default unsigned integral type.

```
- UNSIGNED 8
- UNSIGNED16 /* may be equivalent to unsigned */
- UNSIGNED 32/* may be equivalent to unsigned */
- UNSIGNED64 /* may be equivalent to unsigned */
```

# Unsigned example

```
aStem.calltype = "cdecl as function with parameters"
aStem.0 = 1
aStem.1.type = "unsigned"
aStem.1.name = "size"
aStem.return.type = "unsigned"
aStem.return.name = "mem block casted to unsigned"
call RxFuncDefine "MALLOC", "libc.so.6", "malloc",,
                  aStem
/* do some error checking */

say "5 byte allocated at" malloc(5)
```

# Float Types

- **FLOAT type**s are defined by the keyword "`FLOAT`" immediately followed by or blank separated by a bit count.

```
- FLOAT32
- FLOAT64
- FLOAT80   /* sometimes */
- FLOAT96   /* sometimes */
- FLOAT128   /* sometimes */
```

# Float Example

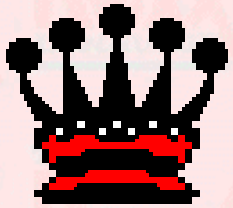
```
aStem.calltype = "cdecl as function with parameters"
aStem.0 = 2
aStem.1.type = "float64"
aStem.1.name = "X"
aStem.2.type = "float64"
aStem.2.name = "Y"
aStem.return.type = "float64"
aStem.return.name = "polar angle of (X,Y)"
call RxFuncDefine "ATAN2", "libm.so.6", "atan2",,
                  aStem
/* do some error checking */

numeric digits 16; say "pi =" 2*atan2(1,0)
```

# Char Types

- **Character type**s are either "CHAR" or "CHAR8" or defined by the keyword "STRING" immediately followed by or blank separated by a **byte** count.

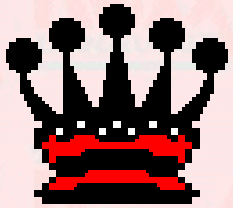
```
- char 8     /* = char = char8 */
- string 20 /* occupies 21 byte because a
             * hidden ASCIIZ-terminator is
             * appended. Use arrays of CHAR8
             * for true character buffers.
             */
```



# Char Example

```
aStem.calltype = "cdecl as function with parameters"
aStem.0 = 1
aStem.1.type = "integer"
aStem.1.name = "errno code"
aStem.return.type = "indirect string 100"
aStem.return.name = "errno literal description"
call RxFuncDefine "STRERROR", "libc.so.6",,
                  "strerror", aStem
/* do some error checking */

say "errno(13) means" strerror(13) /*double buffer*/
say "do you know errortext(100+13)?"
```



# Container

- **Container**s are defined by the keyword "CONTAINER" and have additional fields equivalent to arguments for grouping.

  - c.TYPE = "CONTAINER"
  - c.NAME = <convenient name>
  - c.0 = <element count>
  - c.1 /* e.g. c.1.type = char8 */
  - ...
  - c.<element count>

# Container Example

```
RxString.type = "container"
RxString.0 = 2
RxString.1.type = "unsigned32"
RxString.1.name = "strlength"
RxString.2.type = "indirect string 256"
RxString.2.name = "strptr"

/* Direct siblings are not aligned specially.
 * Be careful when using small subtypes.
 */
```

# Array

- **Array**s are defined by the keyword "ARRAY" and have additional fields equivalent to "CONTAINER".

  - c.TYPE = "ARRAY"
  - c.NAME = <convenient name>
  - c.0 = <element count>
  - c.1 /* e.g. c.1.type = char8 */

/* Just elements .0 and .1 */

# Array Example

```
anArray.type = "array"
anArray.name = "a construct"
anArray.0 = 10
anArray.1.type = "indirect string 256"
anArray.1.name = "some string"
/* no anArray.2.type required */

/* The array contains space for 10 pointers. Each
 * pointer points to a hidden allocated buffer of
 * 257 bytes. Each buffer is aligned to a processor
 * friendly address.
 */
```

# Container Like

- A Container's content can be taken from another container by using the "`LIKE`" keyword.

- Set the type field to
  - `CONTAINER LIKE <name of a stem or branch>`

# Container Like Example

```
aStem.calltype = "pascal as function" /* can't use
"with parameters" because of complex arguments */
aStem.0 = 5
aStem.1.type = "indirect string 256" /* used name */
aStem.2.type = "unsigned32"          /* arg count */
aStem.3.type = "indirect array"      /* arguments */
aStem.3.0 = 10
aStem.3.1.type = "container like RxString"
aStem.4.type = "indirect string 256" /* queuename */
aStem.5.type = "indirect container like RxString"
aStem.return.type = "unsigned32"

call RxFuncDefine "RxFuncDefine", "libgci.so",,
"RxFuncDefine", aStem
```

# Thrown Signals

Signals are thrown when

- wrong stem values are used when calling RxFuncDefine
- a buffer overrun occurs on input for strings
- a value overrun/underrun occurs on input of values
- ±INF or NaN occurs on output of values

`GCI_RC` is usually set. RxFuncErrMsg() returns `GCI_RC` within Regina.

F

# Outlook

- Callback support
- Increase number of supported systems
- Better math unit support while passing parameters