REXX/imc
A REXX Interpreter for UNIX

Ian Collier
Oxford University

Work in progress

REXX/imc

A Rexx interpreter for Unix

Ian Collier

available from `rexx.uwaterloo.ca`

in `/pub/freerexx/imc`

Abstract

Since 1989 I have been working on a Rexx interpreter for Unix in my spare time (what little I get). It was first released to the public in August 1992 and has had many improvements since then. In my presentation I will demonstrate the most recent enhancements and some of the language extensions that I have added to the interpreter, a few of which are connected with the work of the X3.J18 standardisation committee. I hope to show some of the ways in which REXX/imc can interface with its environment; this will include the use of Unix-specific built-in functions, the writing of external function libraries, and the application interface with programs such as THE (an editor based on KEDIT written by Mark Hessling). If time allows, I will take a brief look at the internals of the interpreter, showing the basic blocks of which it is built, and giving a short explanation of how it performs a task such as evaluating a Rexx expression.

Unfortunately, since my 'real' job is to write a D. Phil. thesis, I have not been able to enhance REXX/imc as much as I would have liked for this presentation. However, work is still in progress to turn REXX/imc into an efficient and fully integrated programming language on Unix.

History

- May 1989: Work begins!
- Jan 1991: Interpreter has most language constructs except the stack, but no I/O functions.
- May 1992: REXX/imc is not ready in time for the Symposium.
- Aug 1992: REXX/imc release 1.2 released.
- Sep 1992: Release 1.3
- May 1993: Release 1.4 announced at the Symposium.
- Jun 1993: Release 1.5, the first level 4.00 release of REXX/imc.
- Sep 1993: Release 1.5a, with some bug fixes.
- May 1994: Release 1.6 is presented at the Boston Symposium.
- 1994– : ?

Because REXX/imc is a spare-time project, work on it has been characterised by bursts of activity and long periods of slow development. Even though the interpreter was functional in 1991, it was not released until August 1992. In fact it is interesting to note that REXX/imc was already capable of running a program to calculate π to many decimal places by October 1989, although it had no functions.

The period between the 1992 Symposium and the initial release of REXX/imc was spent in implementing the file I/O functions and in documenting the source—even the few comments that are dotted around now were almost entirely absent before this period!

Many of the changes between versions of REXX/imc have been bug fixes—thanks to Anders Christensen who spent time running his trip tests on REXX/imc, and to everyone who reported a bug.

The main changes in release 1.5 were the addition of language level 4.00 features (`SIGNAL ON` with the `NAME` keyword, `CALL ON`, `CONDITION()`, `STREAM()` and so on), the command line flags, and the `OPTIONS` options.

The main changes in release 1.6 are the addition of an API and the improvement of function handling.

Things planned for the future include, but are not limited to, the following (not in any particular order): implementing speedups (in at least three areas: improving the variable table, improving the arithmetic and implementing a pre-parsing process), improving tracing, adding a Unix system call library, adding `OPTIONS` to control the language extensions and to move towards the language standard, adding extensions as proposed by the Rexx Extensions committee, adding a 'stems' library, completing the API, adding an API which can be called by other processes even after Rexx has started, and anything else which people suggest...

Files

librexx.so.1.6	204800
rexx	5712
rxmathfn.exec	6743
rxmathfn.rxfn	57344
rxmathfn.rxlib	57
rxque	8016
rxstack	6600
const.h	16140
functions.h	16423
globals.h	6165
rexxsaa.h	5678
calc.c	49157
globals.c	8683
interface.c	37875
main.c	4896
rexx.c	97258
rxfn.c	77118
rxmathfn.c	8061
rxque.c	8610
rxstack.c	6051
shell.c	8228
util.c	80214
rexx.info	33568
rexx.ref	155257
rexx.summary	12627
rexx.tech	33320

The file **librexx.so.1.6** is the main library file which contains all the routines necessary for an application to use the SAA API of REXX/imc. On the SunOS system, this is a dynamically loaded shared library, which means that an application which uses the library does not need to include a copy of the library within its object code, thus saving disk space. This can be seen from the fact that the program **rexx**, which is the interpreter itself, is only a 6K file! This program is merely an interface between the command line and the API library, and is compiled from the source file **main.c**.

The programs **rxque** and **rxstack** are for the Rexx stack, which will be discussed later.

The file **rxmathfn.rxlib** is a function dictionary for the REXX/imc mathematical functions, which are implemented in Rexx as **rxmathfn.exec** and in C as the object file **rxmathfn.rxfn**.

As shown opposite, REXX/imc comes with about 430K of source.

The four major documentation files shown opposite are **rexx.info**, which is my attempt at a tutorial for Rexx, **rexx.ref**, which is a complete reference on the language features of REXX/imc, **rexx.summary**, which is a 'reference card' on REXX/imc, and **rexx.tech**, which gives details to the application programmer or any programmer who is interested in the internals of REXX/imc. There are also several minor documentation files, not shown here, which give details about the current release, the change history, the installation instructions, etc.

Invocation

rexx [options] [program] [arguments]

where options are:

-<option> – any option from '**OPTIONS**';

-v – print version;

-s <string> – execute the string as a program;

-t <trace> – turn tracing on;

-i – enter interactive trace mode;

-x – run a Unix-executable Rexx program.

- The **OPTIONS** instruction's most useful option for using on the command line is the **tracefile=f** option, which redirects tracing output to a file.
- The **-t** option can be followed by any Rexx trace setting, which allows you to trace a program without altering it.
- The **-v** option can be used alone (in which case the interpreter does nothing except print its version) or with other options (in which case it prints its version and then runs a program).
- The **-x** option is usually used for programs which invoke themselves on Unix by having a '**#!**' or a shell instruction on the first line. REXX/imc will treat the first line of the program as a comment, and will not append anything to the program name.
- If no program name is given, or if the program name is '**-**', then the program will be read from the standard input.

Added Features

- `stem.(expression)`
- `stem.'string'`
- `SELECT expression`

  `WHEN value THEN instruction`

  `...`

  `END [SELECT]`
- `PROCEDURE HIDE`
- `PARSE VALUE x,y,z WITH p1,p2,p3`
- Any non-zero number is true
- `OPTIONS 'SETRC'` for setting `RC` after I/O operations
- `*,*` trace prefix for continued lines
- Extra tracing for `SIGNAL ON x` when `x` is an undefined label
- Features from CMS
  - `PARSE NUMERIC`
  - `JUSTIFY()`
  - `LINESIZE()`

Of these enhancements, one, namely the `*,*` trace prefix, is as a result of a decision of the X3.J18 standardisation committee, and one other, namely the compound variable with an expression as part of its tail, has been provisionally accepted by the extensions committee. More substantial enhancements based on meetings of these committees (such as date/time conversion functions) were planned but have been delayed.

The `PROCEDURE HIDE` instruction really means 'procedure expose everything-except-the-following', and its use is not strongly reccommended at present.

The `OPTIONS 'SETRC'` instruction makes all I/O (including `SAY` and `PARSE PULL` set the variable `RC` to indicate the success or otherwise of the operation, in order to allow this to be checked without calling `STREAM`. It also causes a `SIGNAL ON ERROR` if that is appropriate. This option was added in order to preserve backward compatibility with a previous version of REXX/imc which had neither `STREAM` nor `SIGNAL ON NOTREADY`.

The 'extra tracing' extension prints out a traceback including the `SIGNAL ON` instruction and the cause of the error whenever the target label for the trap is not found. For example, the program:

```
signal on novalue
call test
exit
test: say xyz
```

produces this traceback:

```
      +++ No-value error on XYZ
    1 +++   signal on novalue
    4 +++  say xyz
    2 +++ call test
Error 16 running test.exec, line 1: Label
not found
```


Features for Unix

- The TRL I/O functions
- Pre-defined streams: `stdin stdout stderr`
- The `STREAM` commands: `close fdopen fileno flush ftell open pclose popen`
- Functions: `CHDIR GETCWD SYSTEM USERID`
- Access to the Unix environment via the `VALUE` built-in function
- Access to Unix error messages via the `ERRORTEXT` built-in function
- Subcommand environments `UNIX` and `COMMAND`
- The stack daemon
- The function interface

- REXX/imc offers a variety of file access functions via the function call `STREAM(stream,'C',command)`. The `open` command allows any file to be attached to a stream in either read or read/write mode. The `popen` command starts a Unix command and attaches it to the named stream for reading or writing. The `fdopen` command allows Rexx to access any Unix file number as a stream. The file number of any Rexx stream is given by the `fileno` command. The `ftell` command gives the file pointer which was set by the last access on the named stream.
- The `SYSTEM` function runs a shell command and returns its output as a string.
- Environment variables may be examined and/or set using the `VALUE` function with a third argument of `'ENVIRONMENT'`. Note, however, that changes made to the environment will be lost when the Rexx interpreter finishes.
- The function call `ERRORTEXT(n+100)` gives the nth Unix error message, such as 'No such file or directory', which is message number 2.
- The subcommand environment `UNIX` passes each command to a Bourne shell. The `COMMAND` environment passes each command to a small built-in shell which tokenises and executes the command directly, which is usually much faster than invoking a shell for each command.

The REXX/imc Stack

- `rxque` is the stack daemon
  - it runs as a separate process
  - it is created and destroyed automatically by the interpreter
  - it may be run as a server for a whole session
- `rxstack` is a stack client
  - `rxstack [-fifo|-lifo]` copies standard input to the stack
  - `rxstack -string x` stacks one entry
  - `rxstack -print` copies stack contents to standard output
  - `rxstack -pop` copies one entry to standard output
  - `rxstack -num` prints the number of stacked entries
- REXX/imc is also a stack client
  - `queue x` stacks an entry in FIFO order
  - `push x` stacks an entry in LIFO order
  - `queued()` tells the number of stacked lines
  - On SunOS, REXX/imc can transfer stack contents to the keyboard buffer.

The program `rxque` forks off a stack daemon and prints out its process number and socket name in the form of two environment variables. The format of the output is as either a Bourne shell command or (with the flag `-csh`) a c-shell command. `rxque` may be given the name of a socket to create, in which case the output is just the process number.

The stack daemon is usually started by REXX/imc and killed with signal 15 when the Rexx program finishes. REXX/imc checks for the presence of a stack daemon by looking for environment variable RXSTACK. If a stack exists, then it uses that instead of creating one. Queued entries may then persist between programs:

```
% eval `rxque -csh`
% ls -al | rxstack
% rexx -s "say queued()"
45
% rexx -s "pull .; parse pull a; say a"
drwx------  5 imc       1024 May  2 16:00 .
% kill $RXSTACKPROC
```

On some systems, REXX/imc can be compiled with the preprocessor symbol `STUFF_STACK` defined. REXX/imc can then pretend to cause persistent changes to the environment:

```
% rexx -s "queue 'cd /tmp'"
cd /tmp
% % pwd
/tmp
```


Application Programming Interface

The following SAA API functions are implemented:

- RexxStart
- RexxVariablePool (except requests RXSHV_EXIT and RXSHV_PRIV)
- RexxRegisterSubcomExe
- RexxDeregisterSubcom
- RexxQuerySubcom
- RexxRegsiterExitExe

  with exits: RXCMDHST RXSIODTR RXSIOSAY RXSIOTRC RXSIOTRD RXINIEXT RXTEREXT
- RexxDeregisterExit
- RexxQueryExit
- RexxRegisterFunctionExe
- RexxDeregisterFunction
- RexxQueryFunction

More will be added later.

Release 1.6 of REXX/imc is the first to have an API. The functions have been modelled on those of OS/2. It should be possible to compile a Rexx-aware application—such as Mark Hessling's editor 'THE'—with REXX/imc without altering it (as long as it uses only the functions which are currently supported).

In order to use the API, an application includes the C header file `rexxsaa.h` supplied with REXX/imc, which will declare the functions opposite and the associated constants and datatypes. When the application is compiled, it is linked with the library file which is created when REXX/imc is compiled. This file will be either `librexx.a`, in which case the code from REXX/imc will be included in the application's object file (*static* linkage), or `librexx.so.1.6`, in which case only a reference to the library file will be included in the application's object file (*dynamic* linkage).

If linkage is dynamic, it will be possible to upgrade to a later release of REXX/imc without recompiling the application, just by copying the new library into the same directory as the old one.

External Functions

External functions or libraries for REXX/imc can be written

- in Rexx
- using the SAA API
- using REXX/imc hooks
- as a Unix program

Writing an external function in Rexx or with the SAA API is the same as for any other interpreter.

A function may be compiled and linked as a dynamically loaded object called `*.rxfn` with the `*` replaced by the function's name (by which it will be called by a Rexx program). When REXX/imc searches for external functions, it searches for such a file first. If the file is found, it is linked in and called as if it were built-in. The function must retrieve its arguments from the REXX/imc calculator stack and place the result (if any) there.

A `*.rxfn` file may contain several functions, all of which will be registered when the file is first loaded.

A function library using the SAA API may be compiled as a `*.rxfn` file in order to make a library which is portable but which can be called by an already-running program. To do this, the library is augmented by an initialisation function which takes no parameters and returns no result, but which uses the SAA API to register all the other functions in the library. Before calling any of the functions, the Rexx programmer must call the initialisation function.

If a function cannot be found, then a Unix program having the same name as the function is searched for. The program can be in any language supported by Unix, such as C, perl or shell script. It will be 'exec'ed with the arguments in `argv[]` and the function name in `argv[0]`, and it should print out the result (if any) on its standard output followed by a newline character.

Many functions can be aliased to one function library by supplying a text file called `*.rxlib` (where `*` is the basename of the function library) which lists the names of all the functions in the library. The library can be a `*.rxfn` file, a Rexx file or a Unix program. If it is Rexx, then it can find out which function is being called using `parse source`.

Interpreting a program

1. Read command line parameters (`main()`)
2. Load program from disk (`load()`)
3. Tokenise program (`tokenise()`)
4. Enter main loop (`interpreter()`)
   (a) Fetch the next token.
   (b) If `NOP` then do nothing
   (c) If `SAY` then print an expression
   (d) If `RETURN` then return an expression
   (e) If `IF` then read and test an expression
   ...
   (f) If program has ended then return, else go to (a).
5. Clean up and finish.

Tokenising a program means (in the case of REXX/imc):

- rejecting invalid characters and unmatched quotes
- removing comments, null clauses and excess blanks
- Concatenating lines which are continued with a ','
- translating un-quoted text to upper case
- recognising keywords (like `NOP`, `SAY`, `IF` and so on)
- organising the program as a list of clauses (each end-of-line, ';', or `THEN` starts a new clause. In addition, labels, `THEN`, `ELSE` and `OTHERWISE` are clauses by themselves)
- making a label table

Keywords are recognised based on what has appeared since the start of the current clause. For example, `THEN` is only allowed when the current clause started with `IF`. Keywords are stored as negative character codes (defined in `const.h`). This makes them easy to recognise: during the main loop, instead of asking, "Are the next three characters 'say'?" we can ask, "Is the next character equal to the constant `SAY` (which is -128)?" It also makes it clearer for the expression evaluator when to stop; the code `WHILE` (-88) is obviously not part of an expression, whereas the word `while` could be a variable name.

The tokenised list of clauses is stored in an array `prog[]`, which also gives other information such as the line number and address of the clause within the source.

The main loop is relatively trivial; it is executing the individual instructions such as `DO` and evaluating the expressions which is the difficult part...

Internal data structures

- the source (`source`)
- the tokenised program (`prog`)
- the label table (`labelptr`)
- the calculator stack (`cstackptr`)
- the program stack (`pstackptr`)
- the signal stack (`sgstack`)
- the variable table (`vartab`) and pointer list (`varstk`)
- the work space (`workptr`)

- The source and tokenised program are each kept in a linear stretch of memory, pointers to which are held in the arrays `source` and `prog` respectively. The label table is stored in a linear stretch of memory which is organised as a kind of linked list.
- The calculator stack is a space to store a list of intermediate values during calculations.
- The program stack records information about the control structures that are currently open (such as `DO` groups and function calls). It stores the variable name, step and limit and/or the `FOR` counter of a `DO` instruction, and it stores all the saved state which must be restored on return from a function call.
- The signal stack holds information about which conditions are currently trapped or delayed, and it also holds the data for the `CONDITION` function. It has one entry for each `INTERPRET` or function call currently active.
- The variable table is a linear stretch of memory which is divided into sections by `varstk`. Each section contains the variables for an active `PROCEDURE` or external function call (apart from the workspace, this is the only one of the above structures which persists across external function calls). Within each section the variables are stored in a tree structure. Exposed variables contain a pointer to another section where the 'real' copy of the variable is to be found.
- The work space is a temporary area for all sorts of calculations. It is cleared after interpreting each instruction.

Example: DO

1. Store the current clause number on the stack.
2. Fetch next token. If clause has ended then finish.
3. Flag the stack entry as 'repetitive'.
4. If the token is `FOREVER`, skip past it.
5. Otherwise, try and fetch a symbol and '='. If found then:
   (a) Store the symbol name on the stack.
   (b) Fetch an expression and assign it to the symbol.
   (c) Search for `TO`, `BY` and `FOR` expressions and store them on the stack.
   (d) If the limit is already passed then `LEAVE`.
6. If that failed, try to evaluate an expression and store it on the stack.
7. Store the pointer to any `WHILE` or `UNTIL` on the stack.
8. If `WHILE` is found and the following expression is false then `LEAVE`.

`DO` and `END` have been chosen to illustrate how the program stack works.

Most of the work of `DO` is to find out what sort of `DO` clause this is and to set up an entry in the program stack which describes the `DO` clause. The information needed is:

- where to come back to
- whether there is a symbol and if so, what are its name, and its step and limit values
- whether there is a counter or `FOR` value, and if so, how many iterations are left
- where the `WHILE` or `UNTIL` can be found, if any

`DO` also has to check to make sure the loop is to be executed at least once.

Example: END

1. Fetch the top stack entry. If none exists, complain.
2. If the entry is not from `DO` or `SELECT`, complain.
3. If the entry is not flagged 'repetitive' then
   (a) Delete the top stack entry
   (b) finish.
4. Fetch the pointer to any `WHILE` or `UNTIL`. If `UNTIL` is found and the expression following it is true, go to 3(a).
5. If a symbol name is stored, add the step to it and compare with the limit. If the limit is passed, go to 3(a).
6. Decrement any `FOR` counter. If it is zero, go to 3(a).
7. Fetch the pointer to any `WHILE` or `UNTIL`. If `WHILE` is found and the expression following it is false, go to 3(a).
8. Fetch the stored clause number and jump to the following clause.

Even though the `END` instruction contains no information (although it might contain a symbol name, details of which have been skipped here), it can be interpreted because the information is all on the program stack. Interpreting the stacked data is relatively straightforward.

Example: expressions

There is a stack of values and a stack of operations.

1. Stack an 'end marker' operation with priority 0.
2. Search for a value:
   - If the next token is a unary operation, stack it and repeat 2.
   - If it is '(' then evaluate the expression inside, check for ')' and go to 3.
   - If it is a quote, collect a string.
   - Collect a symbol name.
   - If the token after the string or symbol is '(' then call a function, otherwise stack its value.
3. Search for the 'current' operator:
   - If the next token is a keyword, ')', ',' or the end of the clause then the operator is an end marker.
   - Otherwise, if it is not a binary operator then the operator is an implicit concatenation.
4. Perform operations:
   - If the top stacked operator and the current operator are both end markers, then finish.
   - If not, and the top stacked operator has a priority no less than that of the current operator, perform the stacked operator and go to 4.
   - Otherwise stack the current operator and go to 2.

The function which performs the above algorithm is called **scanning**.

This is a variant of a well-known algorithm to turn an expression in infix notation into one in reverse polish notation (sometimes described by analogy with a railway track with a siding, the siding being the operation stack). REXX/imc evaluates the reverse polish expression as it is created. The calculator stack is the stack which reverse polish notation requires.

The unary operations each operate on the top value on the calculator stack, replacing it with the result. The binary operations each operate on the top two values, replacing them with the result. It is clear that at step 4 of the above algorithm it is always true that the number of values on the calculator stack is one more than the number of stacked binary operations. Since each stacked binary operation reduces the size of the calculator stack by one item, this means that when the stacked operations have all been performed there is precisely one element left on the calculator stack. This is the result.

Arguments to functions and expressions within parentheses are evaluated by calling **scanning** recursively.