PLATFORM-SPECIFIC STANDARDS FOR REXX

ERIC GIGUERE
UNIVERSITY OF WATERLOO

Platform-Specific Standards for REXX: Issues for Developers and Implementors

Eric Giguère

Computer Systems Group
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Internet: giguere@csg.uwaterloo.ca
BIX: giguere

Introduction

Standards are important to the growth and acceptance of a programming language. However, standards cannot — and should not — specify everything about language implementations. Many details — type sizes, calling conventions, file manipulation methods — will change with the machine architecture and operating system, and sometimes at the whim of the implementation team. This is why standards documents are littered with the phrase "implementation-defined behaviour".

REXX is no different from any other language in this respect. For example, REXX has the capability to send an arbitrary command to a host environment and to receive a return code in reply. The *hows* of registering the host environment, sending the command, and receiving the reply are all implementation-defined. Even the command strings and reply codes have different meanings on different systems.

REXX is different from other languages, however, in that it explicitly provides the means for *integration* and *expandability*. A typical use for REXX is to tie together two applications from different vendors. The REXX implementation should make this integration as painless and seamless as possible, for both the application developers and the end users. And on a given machine/OS pair it shouldn't matter which REXX implementation a user uses — each version of REXX should be "plug compatible" with the other. Similarly, different applications should support similar REXX interfaces and command sets.

This paper discusses the idea of *platform-specific* standards as they apply both to REXX implementors and to developers of REXX-compatible products. The emphasis is on co-operation and the adaptation of existing standards to meet the needs of the REXX community.

Many of the examples in this paper will center around ARexx, an implementation of REXX for the Commodore Amiga. A brief description of the Amiga and ARexx can be found in the appendix. OS/2 REXX and CMS REXX are also mentioned.

1. The Host Interface

There are four basic problems in using or implementing the REXX host command interface.

Finding hosts: The use and syntax of the `ADDRESS` instruction is well-documented, but not what the host address actually means, or how REXX locates the host.

On the Amiga, host addresses correspond to named message ports. When an ARexx-compatible application starts execution it opens a message port to which commands may be sent. This *ARexx port* is a public resource that other applications, including ARexx, can search for by name.[1]

OS/2 and CMS require applications to register *subcommand handlers* with the REXX system. Each handler is a library/program entry point.

Naming hosts: When multiple copies of the same application are running, which host does the user actually want to talk to, and how do they specify it?

Naming rules are not enforced under ARexx. The system ports list is prioritized, but a friendly application should not duplicate or override any name already in the list. Commodore's developer guidelines advocate letting the user set whatever port name they desire. If the user doesn't assign a specific port name, then the application should use its own name, stripped of non-alphanumeric characters and converted to uppercase. Applications that support multiple projects/documents should append a *slot number* to the name, as in `EDIT.01`, `EDIT.02`, and so on (in other words, an ARexx port can be allocated for each document). The actual slot numbers are searched for and assigned dynamically. Slot numbers should also be used when two or more copies of an application are executing simultaneously.

Under OS/2 REXX a user can append the handler's *library name* to the host address, as in `ADDRESS 'Edit.Qedit'`, to differentiate between applications using the same host address.

Sending commands: Some form of inter-application communication is necessary for sending host commands and receiving the return codes in reply. On systems without inter-process communication the interface is obviously much harder to implement.

ARexx simply sends the command as a message to the appropriate port. The message structure is fixed and includes fields for strings, results, and action codes. When the application receives a message at its port, it retrieves the message and parses the command string. The ARexx program that sent the message is suspended until a reply arrives (each ARexx program is a separate task).

OS/2 and CMS call the handler function that was registered with the REXX system.

Starting REXX macros: How does an application access and start the REXX interpreter when it wishes to invoke a REXX macro program?

An Amiga application merely sends a message (using the same message structure described above) to the ARexx *resident process*. The resident process spawns the ARexx program as a new task. The application has the option of waiting for the program to finish execution or of continuing on with its work. The application must always be ready to process incoming ARexx command messages as well.

OS/2 and CMS applications call a system function to start a REXX program.

[1] Note that unlike OS/2 REXX, there is no formal registration to be made. ARexx will search for and locate the appropriate message port each time it has a message to send.

Note that all macros that are started by an application should have the application's port/handler set as their default host address. This minimizes the naming problems discussed above.

2. Command Standards

REXX performs minimal processing for commands: it merely evaluates an expression and sends the resulting string to a host application for processing. It's the application's responsibility to parse this string and act on its contents. This is as it should be: it is not REXX's mandate to assign meaning to these strings.

It would be nice, however, if REXX-aware applications used similar command sets.[2] A simple text editor macro such as:

```
/* Load a file, search for a string */

'open' arg(1)
'search' arg(2)
```

should be simple to adapt for another text editor. This makes it possible for developers to include fairly complete sets of REXX macros with their applications without having to explicitly support every programming tool on the market. Command standards make application integration much simpler.

As an example of what can happen without a set of guidelines, consider the text editors available for the Amiga. These days all Amiga text editors (and most other applications) are ARexx-aware. Unfortunately, their ARexx command sets are completely incompatible. While there are many interesting macros available for various purposes (automatic tracking of compiler errors, for example), the macros have to be rewritten from scratch for each text editor. Telecommunication utilities are in the same boat.

To address this situation, Commodore has just released a set of ARexx command guidelines to developers as part of the *Amiga User Interface Style Guide*. The guidelines list suggested commands (names and options) for common operations. Hopefully new applications will include these commands in their command set and ease the confusion that prevails right now.

3. Returning Command Results

It isn't enough to send commands to an application — a REXX program must be able to receive and process data from those commands as well, if only to know whether the command failed or succeeded.

"Vanilla" REXX only defines the concept of a *return code* when it comes to command results. When a command has been executed, the special variable `RC` will hold a numeric value which the program can then use as basis for further action:

```
/* Run a program */
address command
'run rxtools:rxtools'
```

[2] Commands to the underlying operating system being an obvious exception.

```
if( rc ~= 0 )then do
    say "Could not start RxTools."
    exit 1
end
```

The value in RC is (of course) implementation-defined. A general convention is that non-zero values indicate warning or error conditions.

ARexx implements an extension that allows applications to return *result strings* as well as return codes. Result strings must be requested before sending a command by using the OPTIONS RESULTS instruction. After a command has been sent, the special variable RESULT may have been set to a value if no error occurred (RC is 0):

```
/* Ask user for a string */
options results
address 'rexx_ced'
'getstring "Please enter search pattern:"'
if( result ~= "" & result ~= "RESULT" )then do
   ..... /* do stuff */
end
```

Note that RC doesn't actually need to be checked since if an error occurred the value of RESULT will be DROPped automatically.

The OPTIONS RESULTS extension is a useful one because it allows commands to act like function calls. Another way of passing back data is to use RVI/VPI, as discussed below.

Using a clipboard is also an effective way of passing information between applications. A 'PASTE' command could be sent to an application to place its data into the clipboard. A set of functions (either built-in or part of a function package) could then be used from within a REXX program to manipulate this data. The paste operation could even be performed manually by the user for applications that aren't REXX-aware.

4. Function Packages

A simple but effective way of extending the REXX language is to allow developers to create their own *function packages*. REXX programs can then call the functions in a package as if they were built-in functions. Unlike external functions, the functions in these packages are probably not written in REXX.

An obvious use for function packages is to extend REXX to include user interface support. At least two such packages (one freeware, one commercial) already exist on the Amiga. Another ARexx function package provides transcendental mathematics functions.[3] ARexx even allows applications to act as function packages (*function hosts*) as well as accepting command messages.

The user can run into trouble using function packages, however, through *namespace pollution*. Sooner

[3]For serious mathematics an ARexx-aware application program such as Maple is recommended instead.

or later one function package is going to use a name already used by another function package. Which function will get called? Can function packages override built-in functions?

5. RVI/VPI

The last important issue has to do with the sharing of data between REXX and application programs. Under ARexx this capability is known as the *REXX Variables Interface* (RVI), while OS/2 and CMS refer to it as the *Variable Pool Interface* (VPI).

RVI can only be used by REXX-aware applications. RVI allows these applications to set and examine the value of REXX variables. These alterations can only happen when a REXX program has sent a host command and is waiting for a reply. During the processing of the command the host application can use RVI to modify the REXX program's symbol tables.

RVI is a simple way to pass back complex information to a REXX program. For example, on CMS the XEDIT `'EXTRACT'` command can be used to copy state information into a stem variable in the calling program.

When processing a command, a host application must be careful to ensure that the command came from a REXX program and not some other application before using the RVI routines. Some method must be built into the host addressing to differentiate between REXX and non-REXX callers.

And of course, user documentation is very important. If an application can change a variable, the user should be made aware of the fact. Preferably, the user will have control over which variables are changed. Commodore's command standards, for example, include `'VAR'` and `'STEM'` options to allow the user to specify variables for command results.

Conclusions

This paper doesn't pretend to present any startling conclusions, but only some observations and some simple advice for any implementor: check out current REXX implementations, especially the ones on the platform you're developing for. If possible, offer a similar set of capabilities and interfaces, at least as an option. Consider the interfaces other macro languages — BASIC, for example — offer, especially if REXX is not the predominant macro language for your system. You can't expect every application to be REXX-aware, so it certainly helps if they can still use REXX even on the most rudimentary level.

A. ARexx: A Sample Platform

The Commodore Amiga is a microcomputer based on the Motorola 680x0 architecture. The base operating system, *Exec*, is a message-based, preemptive multitasking system. File I/O and command shells are provided by *AmigaDOS*, while graphics and user interface support are provided by *Intuition*.

ARexx is an implementation of REXX 3.5 with some Amiga-specific extensions. The ARexx interpreter is stored as a *shared library* and is about 33K in size. A *resident process* of about 3K runs as a

background task. The resident process is the master ARexx control program: it launches new ARexx programs and keeps track of global resources.

To start an ARexx program, a message is sent to the resident process. It spawns a new process which invokes the interpreter. Each ARexx program runs as a separate task and performs its own resource tracking.

Messages are at the heart of ARexx program interaction. ARexx defines its own message protocol as an extension of the Exec message structure. (The Amiga's memory space is shared between all tasks. Message ports are really just linked lists.) An ARexx message is defined as follows:

```
struct RexxMsg
  {
   struct Message  rm_Node;      /* Exec message structure       */
   APTR            rm_TaskBlock; /* global structure (private)   */
   APTR            rm_LibBase;   /* library base (private)       */
   LONG            rm_Action;    /* command (action) code        */
   LONG            rm_Result1;   /* primary result (return code) */
   LONG            rm_Result2;   /* secondary result             */
   STRPTR          rm_Args[16];  /* argument block (ARG0-ARG15)  */

   /* Extension fields (not modified by ARexx) */

   struct MsgPort *rm_PassPort;  /* forwarding port              */
   STRPTR          rm_CommAddr;  /* host address (port name)     */
   STRPTR          rm_FileExt;   /* file extension               */
   LONG            rm_Stdin;     /* input stream (filehandle)    */
   LONG            rm_Stdout;    /* output stream (filehandle)   */
   LONG            rm_avail;     /* future expansion             */
  };
```

The message structure includes fields for setting various *action codes* (whether a message is a host command or function call) and *modifier flags* (is a result string required? how many arguments are being passed?), a return code, a result string, and the arguments for the command or function call. Arguments are always passed as strings.

To send a host command, an ARexx program allocates a `RexxMsg` structure, fills in the appropriate values, and sends the message to the host's port. The host will receive the message, parse the command string, execute the command, set a return code (and if requested, a result string) and reply to the message. The ARexx program remains blocked until the reply message arrives.

Function calls are also handled with messages. If a function call cannot be resolved by an internal or built-in function, ARexx will search a prioritized *library list* maintained by the resident process. Each entry in the library list is either a *function library* or a *function host*. A function library is a shared library with a public entry point, while a function host is an application program with a public message port. ARexx will query each library/host in turn (by calling the library's entry point directly or by sending a message to the host) until the desired function is found. If this fails, a search is made for an external function.

References

[Amiga 91] Commodore-Amiga, Inc. *Amiga Programmer's Guide to ARexx*, 1991 (forthcoming).

[Cowlishaw 90] M. F. Cowlishaw. *The REXX Language: A Practical Approach to Programming, 2nd edition*, Prentice-Hall, 1990.

[Giguère 91] Eric Giguère. "Rexx: Not Just a Wonder Dog", *Computer Language*, Vol. 8 No. 3, March 1991.

[Hawes 87] William S. Hawes. *ARexx User's Reference Manual*, Wishful Thinking Development Corporation, 1987.

[IBM 87] International Business Machines Corporation. *SAA Common Programming Interface/Procedures Language Reference*, SC26-4358-0, 1987.

[IBM 88a] International Business Machines Corporation. *VM/SP System Product Interpreter Reference*, SC24-5239-03, 1988.

[IBM 88b] International Business Machines Corporation. *VM/SP System Product Interpreter User's Guide*, SC24-5238-04, 1988.

[IBM 89a] International Business Machines Corporation. *OS/2 1.2 Procedures Language 2/REXX*, 1989.

[IBM 89b] International Business Machines Corporation. *OS/2 1.2 Procedures Language 2/REXX Programming Reference*, 1989.

[Watts 90] Keith Watts. "REXX Language I/O and Environment Challenges", *Proceedings of the 1990 REXX Symposium*, SLAC Report 368, 1990.