How to Write Portable Rexx

by Howard Fosdick © 2012

Rexx offers a number of advantages as a cross-platform language. It has a strong standard that every interpreter upholds, called *TRL-2*. It runs on nearly every operating system and platform. And it is an easy language to maintain, so that if you do have to make changes when porting Rexx, this takes less time and effort than it does in many other programming languages. This primer tells how to write portable Rexx code.

The Two Rexx Standards

First, understand the two Rexx standards and decide which you'll code to. Every Rexx interpreter conforms to the language definition set out in Michael Cowlishaw's definitive work, *The Rexx Language 2nd ed* – commonly referred to as *TRL-2*.

The American National Standards Institute codified the language further in their ANSI-1996 Rexx Standard. The ANSI standard sets out a few more required language elements as a superset of TRL-2. This document summarizes the differences between the TRL-2 and ANSI-1996 standards.

Writing Portable Rexx Code

To write portable Rexx code -- or to port existing Rexx code from one platform to another -- your first task is to identify what parts of Rexx programs may not be portable. These include --

- Operating system commands
- Commands to non-portable interfaces (eg: GUIs, databases, etc)
- Interpreter-specific extensions (eg: Regina Rexx extensions versus TSO-REXX language extensions)
- I/O incompatiblities (especially when using non-standard Rexx I/O, for example, the **EXECIO** command of mainframe Rexx)
- Operating systems and interface error codes may vary across platforms

This diagram summarizes what Rexx components are portable versus those that may not be, depending on the situation --

There are a number of ways around portability issues—

- Use packages like **RexxUtil** (aka RegUtil) that are specifically designed to render operating system functions portable.
- Code strictly to TRL-2 standards (since all Rexx interpreters meet these standards)
- Use TRL-2 standard I/O functions
- Minimize operating system commands (since these are often OS dependent)
- Ensure interfaces are portable across the systems you're dealing with
  (example— if you code to a database interface, make sure that database or
  its interface runs on all the platforms that concern you)
- Write a routine that informs your program which platform it is running on, then execute **IF** or **SELECT** statements to run platform-specific code
- Isolate platform-specific code in its own subroutines or functions
- Isolate error-handling to Rexx's exception-handlers, and manage platform-specifc errors there. Or isolate error-handling to your own platform-specific routines.

Writing a Routine to Discover the Environment

A cross-platform Rexx script needs a subroutine or function to determine the platform it runs on. This routine can then be invoked in the program when it needs this information to execute correct code for the platform.

The subroutine will (minimally) use the **parse source system** and **parse version** instructions. It may also issue **address** to determine the default command environment. Here's an example:

```
/* WHERE AM I:                                                    */
/*                                                                */
/*    This script learns about its environment and determines     */
```

```
/*    exactly which Windows or Linux OS it runs under.           */

parse version language level      date     month year .
parse source  system   invocation filename .

language = translate(language)       /* ensure using Regina Rexx */
if pos('REGINA',language) = 0 then
   say 'ERROR: Interpreter is not Regina:'  language

say 'Interpreter version/release date:' date month year
say 'Language standards level is:     ' level
say 'Version information from an OS command follows...'

/* determine operating system, get its version/release info      */

select
   when system = 'WIN32' then
       'ver'
   when system = 'LINUX' then
       'uname -a'
   otherwise
      say 'Unexpected SYSTEM:' system
end

if rc <> 0 then            /* write message if OS command failed */
   say 'Bad return code on OS Version command:' rc

/* Code from Rexx Programmers Reference */
```

Here's a similar routine written in Open Object Rexx (ooRexx):

```
/*********************************************************************/
/* WHICH OS
*/                                                                   /*
*/
/* Identifies the operating system by the command shell.             */
/*********************************************************************/
os = .operating_systems~new   /* create a new object                 */

os~write_command_shell        /* invoke the method to do work        */

exit 0                        /* always code EXIT instruction        */

::class operating_systems     /* class with 2 methods following it */

  ::method init               /* method INIT prompts for shell name*/
    expose shell              /* EXPOSE the shared variable          */
    say 'Enter the shell name:'  /* prompt for and read user input */
    parse pull shell .
    return
```

```
::method write_command_shell  /* this method determines OS       */
  expose shell
  select                       /* determine OS for this shell     */
    when shell = 'CMD'      then string = 'DOS or Windows 9x'
    when shell = 'COMMAND'  then string = 'Windows 2K/2003/XP'
    when shell = 'ksh'      then string = 'Korn Shell'
    when shell = 'csh'      then string = 'C Shell'
    otherwise string = 'unknown'
  end
  say 'OS is:' string         /* write out the OS determined     */
  return 0
```

Steve Ferg provides similar classic Rexx code in his discussion on porting Rexx scripts here.

Running Different Code on Different Platforms

Once you have a routine to determine which platform your script is running on, select the platform dependent code to run like this—

```
/*    This code runs a different routine depending on its  */
/*    host operating system.                               */

 my_platform = determine_platform_routine()

 select
    when  my_platform = 'WIN32'  then
    /* insert Windows code or a routine here */

    when my_platform = 'UNIX'  |  'LINUX'  then
    /* insert UNIX/LINUX code or routine here */

    otherwise
    say 'Unexpected, unrecognized system–error'

 end   /* Code from Rexx Programmers Reference */
```

Use **IF** statements or a **SELECT** statement to run code appropriate to the platform the script runs on. The platform-specific code could either be written in-line, or invoked through a subroutine or function. Use a subroutine if there's much code involved. OS-specific error handling within each subroutine is also a good idea.

How Different are Your Targets?

Part of your challenge is determined by the systems you're porting to. How similar or different are they?

For one site I wrote code that ran on Oracle/Sun Solaris, HP's HP/UX, and IBM's AIX. These are all Unix variants, so writing and testing cross-platform code was pretty easy.

Contrast this to another project, where my code had to run under both Windows and Linux. Not only are these two systems more dissimilar, but Windows systems have slightly different OS command sets depending on the OS version. Tracking the slight differences in Windows commands across Windows 95/98/ME/XP/Vista/7/8 is challenging!

Always see if you can limit your porting to the smallest number of operating systems and versions possible. For example, a dual-platform script is easier to write if it only supports Ubuntu 10.04 and Windows 7 than if it must run on all Linux distros and all Windows versions.

Conclusion

Rexx ports well across systems. It is highly standardized, runs on many different systems, and is easy to change if you do have to write platform-specific code. I hope that the basic principles in this article help.