ISSUES IN THE SPECIFICATION OF REXX

CHARLES DANEY
QUERCUS SYSTEMS

Issues in the specification of REXX

Charles Daney
Quercus Systems
P. O. Box 2157
Saratoga, CA 95070
(408) 867-REXX

REXX is, for the most part, clearly and thoroughly specified in *The REXX Language*. This presentation deals with a few areas which have been found to be less completely specified.

File I/O facilities

- File "opening"

  Many operating systems require an explicit "open" operation before performing file I/O functions. This operation is important for specifying file sharing and processing options. REXX permits an open to be done in the STREAM() function, but does not specify any standard syntax for widely applicable options.

- File read/write pointers

  REXX specifies that independent read and write pointers shall be maintained for I/O positioning. But the specification isn't sufficiently clear and emphatic, with the result that significant implementations, such as IBM's OS/2 REXX, do not maintain independent pointers. This can cause serious and difficult to detect problems when applications are ported to different platforms.

- File "closing"

  Although LINEOUT() and CHAROUT() can be used to close a file, their use is not intuitive for input files. In addition, the specific syntax used makes it impossible to close the default input stream.

- Ambiguities in LINES() and CHARS()

  These functions permit an application to determine whether any data remains to be read in an input stream. It is recognized that a given file system may have difficulty implementing both precisely. But there is no way for an application to determine when the results may be inexact. The behavior of these functions on "transient" streams needs to be clarified to distinguish whether a value of 0 means no data currently available, or whether an "end of file" indication has been received. (This is an issue, for instance, with "pipes".) A related problem is to specify when CHARIN() may return with fewer than the requested number of characters (as opposed to when it should wait).

- Lack of STREAM() standardization

  The STREAM() function was recently added to the language. While it does provide a means of performing certain I/O operations like opening, closing, seeking, and obtaining file information, it uses an un-REXX-like command syntax. In addition, because the exact command syntax isn't specified, STREAM() is useless for writing portable applications.

- Lack of file maintenance functions

  There should be standard functions for common file maintenance operations, such as "create", "delete", "rename", and the like. Most systems where REXX is currently implemented also support the hierarchical file directory concept, and REXX needs analogous standard maintenance functions for directories as well.

- Implementation capability determination

  While it is clear that implementations of REXX in different environments cannot be expected to support fully equivalent file I/O capabilities, there is no means for an application to determine in a portable way what is supported. The capabilities which are likely to be of interest include line or character orientation of files, read or write protection, end-of-file detection, and ability to perform random access.

- I/O error handling with NOTREADY

  The recently-added NOTREADY condition behaves differently from all other conditions (except HALT), in that it can occur multiple times within a single clause. It is not specified whether the condition will be raised independently for each occurrence, nor in what order in relation to other possible conditions.

- Independence of file operations in REXX programs

  Because there is no explicit open operation in REXX, it is unclear how operations on the same file by separate REXX programs should be handled. If a file is "opened" in one program, it is also "open" in another program called from the first? This affects the handling of file status information such as the read/write pointers. Some environments (e. g. OS/2 and Unix) allow child processes to "inherit" open files but also permit independent access to the same files. Which way should REXX go? And at what point can a file be assumed to be closed automatically if an explicit close isn't done?

Miscellaneous language issues

- External data queue

  The concept of an external data queue is firmly embedded in the REXX definition, yet system support for it varies greatly across operating systems. It is not very clear how the queue should interact with a keyboard input stream, particularly for programs run by REXX. Important queue facilities, such as the ability to have independently named queues or separate "buffers" within a queue are outside the scope of the language definition, making it impossible for applications to use such features and remain portable. The problems are similar to I/O portability problems.

- Recommended function return values

  There are two incompatible ways of representing "success" and "failure" in the values returned by functions which are called primarily for their side effects. 1 to represent success and 0 to represent failure makes sense by analogy with the representation of "true" and "false". Yet 0 for success and 1 (or non-zero) makes sense by analogy with command return codes. REXX should have a recommended representation in order to avoid widespread confusion.

- Internal numeric precision of built-in functions

  REXX specifies that, with the exception of "mathematical" functions, NUMERIC DIGITS 9 is assumed for internal operations regardless of what prevails otherwise in the program. This is untenable for I/O functions like CHARIN()/CHAROUT() already, and may be so for other functions in the future (e. g. very long character strings).

- Byte ordering of D2C(), C2D(), etc.

  REXX takes no position on byte ordering issues, despite the differences existing between various CPU types. This causes problems primarily when REXX programs must access data from other sources at a byte level. The lack of a REXX standard for this means that a REXX program, even if it knows what type of CPU it is running on, cannot tell what byte order is in use. This problem is more acute for REXX than for other languages, because all REXX data is untyped and nominally treated as strings of characters. Most CPU types lay out character strings in ascending address order, regardless of how they represent numbers. But REXX has no way to tell the difference.

- Confusing terminology for condition handling

  There are two separate events in the handling of one condition which are not clearly distinguished. The first is the occurrence of the circumstances which define the condition (e. g. an I/O error for NOTREADY), and the second is the invocation of a handler (either default or user-defined). These two events can be separate, at least for NOTREADY and HALT. Consequently, it is not possible to speak precisely about how REXX actually processes such conditions. The term "trapped" is sometimes used to describe either event, and in addition the situation that a user-defined handler has been enabled for the condition. "Enabled" itself is used ambiguously to mean the event can occur, or that a condition handler has been defined.

- Labels allowed within IF, SELECT, and DO instructions

There is no clear prohibition of labels in inappropriate contexts, and most implementations seem to allow them. Yet there seems to be little legitimate use for labels within IF, SELECT, and DO instructions, as actual use would almost always lead to errors. Perhaps they should be prohibited. A similar question is whether duplicate labels within a program should cause a syntax error.

API capability guidelines

Although capabilities of the API (application programming interface) are normally outside of the scope of a language definition, the unique position of REXX as a "glue" language between applications and the operating system raises the API to a level of importance it would not otherwise have. There is substantial informal agreement on a minimal set of capabilities that need to be present, such as interfaces for the variable pool, command invocation, and function packages. (Though even in these cases, there is much more vagueness than precision.) But various other interfaces appear desirable, yet haven't even begun to be defined well.

- Service exits

A service exit allows the REXX language processor to make a call to a handler defined by an application in order to process commonly used services such as keyboard and screen I/O. At least one implementation that does provide service exits (IBM's OS/2 REXX) seems to provide exits for keyboard/screen I/O at some times but not others, defeating the purpose of allowing an application to control the keyboard and screen.

- Exits for raising conditions

When an application is called from REXX for either command or function handling, it should be able to raise REXX conditions which will be recognized upon return to REXX, the HALT condition most especially. It would be nice if applications could raise appropriate conditions not already defined by REXX. This would require that REXX allow the enablement of "non-standard" condition handlers. Clarification of condition handling terminology to cover such cases is also needed.

- Specification of search order for external routines

Each REXX implementation currently defines a search order for external routines, and the order necessarily is implementation dependent. There is, however, no recognized API for an application to change the search order, other than (perhaps) allowing for the execution of programs already loaded into memory.