by Howard Fosdick

Input and Output in Rexx

Overview

Input/output, or *I/O*, is how a program interacts with its environment. Input may come from what a user types in, an input file, or another program. Program output might be written to the display, to an output file, or to a communication mechanism such as a pipe. These are just a few of the possibilities.

Rexx provides a simple-to-use, high-level I/O interface. At the same time, Rexx aims for standardization and portability across platforms. Unfortunately, this latter goal is difficult to achieve — I/O is inherently *platform-dependent*, because it relies upon the file systems and drivers the operating system provides for data management. These vary by operating system.

This chapter describes the Rexx I/O model at a conceptual level. Then it explores examples and how to code I/O. The last part of the chapter discusses some of the problems that any programming language confronts when trying to standardize I/O across platforms, some of the trade-offs involved, and how this tension has been resolved in Rexx and its many implementations.

Rexx provides an I/O model that is easy to use and as portable as possible. Section II explores the I/O extensions that many versions of Rexx offer for more sophisticated (but less portable) I/O. Chapter 15 illustrates database I/O and how to interface scripts to popular database management systems such as SQL Server, Oracle, DB2, and MySQL.

The Conceptual I/O Model

Rexx views both input and output as *streams* — a sequence of *characters*, or *bytes*. The characters in the stream have a sequence, or order. For example, when a Rexx script reads an input stream, the characters in that stream are presented to the script in the order in which they occur in the stream.

A stream may be either *transient* or *persistent*. A transient stream could be the characters a user enters through the keyboard. They are read; then they are gone. A persistent stream has a degree

[This tutorial was excerpted from the *Rexx Programmer's Guide*, Wiley, © 2005]

of permanency. Characters in a file, for example, are stored on disk until someone deletes the file containing them. Files are persistent.

For persistent streams only, Rexx maintains two separate, independent *positions:* a *read position* and a *write position*. The type of access to the persistent stream or file determines which of these positions make logical sense. For example, for a file that a script reads, the read position is important. For a file that it writes, the write position is important.

The read and write positions for any one file may be manipulated by a script independently of one another. They might be set or altered *explicitly*. Normally, they are altered *implicitly* as the natural result of read or write operations.

Programs can process streams in either of two *modes*: character by character or line by line. Rexx provides a set of functions to perform I/O in either manner. These are typically referred to as *character-oriented I/O* and *line-oriented I/O*. Figure 5-1 summarizes these two basic I/O modes.

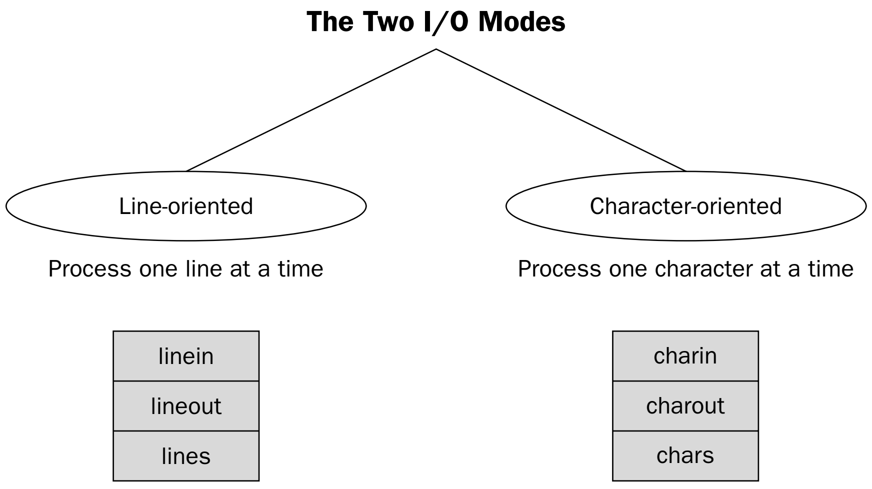


Figure 5-1

A stream is typically processed in either one of the two I/O modes or the other. However, it is possible to intermix character- and line- oriented processing on a single stream.

Like many programming languages, Rexx recognizes the concept of *standard input* and *standard output*. The former is the default location from which input is read, and the latter is the default location to which output is written. These defaults are applied when no specific name is encoded in a Rexx statement as the target for an I/O operation. Standard input is normally the keyboard, and standard output is the display screen. Standard Rexx does not include the concept of a *standard error stream*.

As with variables, Rexx files are defined by their first use. They are not normally predefined or "declared." In standard Rexx, one does not explicitly "open" files for use as in most programming languages. Files do not normally need to be closed; they are closed automatically when a script ends. For most situations, this high level of automation makes Rexx I/O easy to use and convenient. For complex programs with many files, a situation in which memory is limited, or when a file needs to be closed and reopened, Rexx provides a way to explicitly close files.

Line-Oriented Standard I/O

With this conceptual background on how input/output works in Rexx, we can describe standard Rexx I/O. Let's start with I/O that considers the stream to consist of *lines*, or line-oriented I/O. Here the three basic functions for standard line I/O:

- ❑ `linein` — Reads one line from an input stream. By default this reads the line from default standard input (usually the keyboard).
- ❑ `lineout` — Writes a line to an output stream. By default this writes to standard output (usually the display screen). Returns `0` if the line was successfully written or `1` otherwise.
- ❑ `lines` — Returns either `1` or the number of lines left to read in an input stream (which could be `0`).

This sample script reads all lines in an input file, and writes those containing the phrase `PAYMENT OVERDUE` to an output file. (A form of this simple script actually found a number of lost invoices and saved a small construction company tens of thousands of dollars!):

```
/*  FIND PAYMENTS:                                               */
/*                                                               */
/*  Reads accounts lines one by one, writes overdue payments     */
/*  (containing the phrase PAYMENT OVERDUE) to an output file.   */

parse arg filein fileout               /* get 2 filenames         */

do while lines(filein) > 0             /* do while a line to read */
  input_line = linein(filein)          /* read an input line      */
  if pos('PAYMENT OVERDUE',input_line) >= 1 then        /* $ Due?  */
     call lineout fileout,input_line   /* write line if $ overdue */
end
```

To run this program, enter the names of its two arguments (the input and output files) on the command line:

```
regina   find_payments.rexx   invoices_in.txt   lost_payments_list_out.txt
```

In this code, the `parse arg` instruction is to `arg` as `parse pull` is to `pull`. In other words, it performs the exact same function as its counterpart but does not translate input to uppercase. `arg` and `parse arg` both read input arguments, but `arg` automatically translates the input string to uppercase, whereas `parse arg` does not. This statement reads the two input arguments without automatically translating them to uppercase:

```
parse arg filein fileout               /* get 2 filenames        */
```

This statement:

```
do while lines(filein) > 0
```

shows how Rexx programmers often perform a read loop. The `lines` function returns a positive number if there are lines to read in the input file referred to. It returns `0` if there are none, so this is an easy way to test for the end of file. The `do` loop, then, executes repeatedly until the end of the input file is encountered.

The next program statement reads the next input line into the variable `input_line`. It reads one *line* or record, however the operating system defines a *line*:

```
input_line = linein(filein)           /* read an input line      */
```

The `if` statement uses the string function `pos`, which returns the position of the given string if it exists in the string `input_line`. Otherwise, it returns `0`. So, if the character string `PAYMENT OVERDUE` occurs in the line read in, the next line invokes the `lineout` function to write a line to the output file:

```
if pos('PAYMENT OVERDUE',input_line) >= 1 then        /* $ Due?  */
   call lineout fileout,input_line    /* write line if $ overdue */
```

There are two ways to code the `lineout` function:

```
call  lineout  fileout,input_line
```

or

```
feedback  = lineout(fileout,input_line)
```

The recommended approach uses the `call` instruction to run the `lineout` function, which automatically sets its return string in the special variable `result`. If the variable `result` is set to `0`, the line was successfully written, and if it is set to `1`, a failure occurred. The sample script opts for clarity of illustration over robustness and does not check `result` to verify the success of the write.

The second approach codes `lineout` as a function call, which returns a result, which is then assigned to a variable. Here we've assigned the function return code to the variable `feedback`. You'll sometimes see programmers use the variable `rc` to capture the return code, because `rc` is the Rexx *special variable* that refers to return codes:

```
rc  = lineout(fileout,input_line)
```

Now, here's something to be aware of. This coding *will not work*, because the return string from the `lineout` function has nowhere to go:

```
lineout(fileout,input_line)      /*  Do NOT do this, it will fail!  */
```

What happens here? Recall that the return code from a function is placed right into the code as a replacement for the coding of the function. So after this function executes, it will be converted to this if successful:

```
0
```

A standard rule in Rexx is that whenever the interpreter encounters something that is not Rexx code (such as instructions, expressions to resolve, or functions), Rexx passes that code to the operating system for execution. So, Rexx passes `0` to the operating system as if it were an operating system command! This causes an error, since `0` is not a valid operating system command.

We'll discuss this in more detail in Chapter 14, when we discuss how to issue operating system commands from within Rexx scripts. For now, all you have to remember is that you should either `call` a function or make sure that your code properly handles the function's returned result.

The `lines` function works slightly differently in different Rexx implementations. It always returns `0` if there are no more lines to read. But in some Rexx interpreters it returns `1` if there are more lines to read, while in others it returns the actual number of lines left to read. The latter produces a more useful result but could cause Rexx to perform heavy I/O to determine this value.

The ANSI standard clarified this situation in 1996. Today ANSI-standard Rexx has two options:

❑ `lines(file_name,C)` —Count. Returns the number of lines left to read.

❑ `lines(file_name,N)` —Normal. Returns 1 if there are lines left to read.

For backward compatibility, the second case is the default. A true ANSI-standard Rexx will return `1` if you encode the `lines` function without specifying the optional parameter, and there are one or more lines left to read in the file. However, some Rexx implementations will return the actual number of lines left to read instead of following the ANSI specification.

Standard Rexx does not permit explicitly opening files, but how about closing them? Rexx closes files automatically when a script ends. For most programs, this is sufficient. The exception is the case where a program opens many files and uses an exceptional amount of memory or system resources that it needs to free when it is done processing files. Another example is the situation in which a program needs to close and then reopen a file. This could happen, for example, if a program needed to sequentially process the same file twice.

How a file is closed or how its buffers are flushed is implementation-dependent. Most Rexx interpreters close a file by encoding a `lineout` function without any parameters beyond the filename. Just perform a write operation that writes no data:

```
call  lineout  'c:\output_file'     /* flushes the buffers and closes the file -
                                       in most Rexx implementations           */
```

The `stream` function is another way to close files in many implementations. `stream` allows you to either:

Check the state of a file

or

Issue various commands on that file

The status check is ANSI standard, but the specific commands one can issue to control a file are left to the choice of the various Rexx implementations. Here's how to issue an ANSI-standard status check on a file:

```
status_string = stream(file_name)      /* No options defaults to a STATUS check */
```

or

```
status_string = stream(file_name,'S')  /* 'S' option requests return of
                                          file STATUS                */
```

The status values returned are those shown in the following table:

Stream Status	Meaning
READY	File is good for use.
NOTREADY	An I/O operation attempt will fail.
ERROR	File has been subjected to an invalid operation.
UNKNOWN	File status is unknown.

The commands you can issue through the `stream` function are completely dependent on which Rexx interpreter you use. Regina Rexx allows you to open the file for reading, writing, appending, creating, or updating; to close or flush the file, and to get its status or other file information. Regina's `stream` function also allows scripts to manually move the file pointers, as would be useful in directly accessing parts of a file.

The file pointers may be moved in several ways. All Rexx scripts that perform input and/or output do this *implicitly*, as the result of normal read and write operations. Scripts can also move the file pointers *explicitly* . . . but these operations are implementation-specific. Some Rexx interpreters, such as Regina, enable this via `stream` function commands, while others provide C-language-style `seek` and `tell` functions that go beyond the Rexx standard. Read your Rexx's documentation to see what your interpreter supports. Part II goes into how specific Rexx interpreters provide this feature and offers sample scripts.

The `lineout`, `charout`, `linein`, and `charin` functions provide the most standardized way to explicitly control file positions, but care is advised. Most scripts just perform standard read and write operations and let Rexx itself manage the file read and write positions. Later in this chapter we discuss alternatives for those cases where you require advanced file I/O.

Character-Oriented Standard I/O

The previous section looked at line-oriented I/O, where Rexx reads or writes a line of data at a time. Recall from the introduction that Rexx also supports *character-oriented I/O*, input and output by individual characters. Here the three basic functions for standard character I/O:

❑ `charin` — Returns one or more characters read from an input stream. By default this reads one character from default standard input (usually the keyboard).

- `charout` — Writes zero or more characters to an output stream. By default this writes to standard output (usually the display screen). Returns `0` if all characters were successfully written. Or, it returns the number of characters remaining after a failed write.
- `output` (usually the display screen) — Returns `0` if all characters were successfully written. Or, it returns the number of characters remaining after a failed write.
- `chars` — Returns either `1` or the number of characters left to read in an input stream (which could be `0`).

This sample program demonstrates character-oriented input and output. It reads characters or *bytes*, one by one, from a file. It writes them out in hexadecimal form by using the `charout` function. The script is a general-purpose "character to hexadecimal" translator. Here is its code:

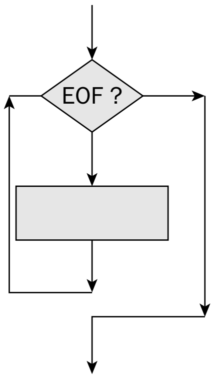
```
/*  TRANSLATE CHARS:                                               */
/*                                                                 */
/*   Reads characters one by one, shows what they are in hex format  */

parse arg filein fileout .     /* get input & output filenames       */
out_string = ''                  /* initialize output string to null */

do j=1 while chars(filein) > 0       /* do while a character to read */
   out_string = '  ' c2x(charin(filein))      /* convert it to hex   */
   call charout ,out_string                   /* write to display    */
   call charout fileout,out_string            /* write to a file too */
end
```

The script illustrates the use of the `chars` function to determine when the input file contains no more data to process:

```
do j=1 while chars(filein) > 0       /* do while a character to read */
```

This character-oriented `chars` function is used in a manner similar to the line-oriented `lines` function to identify the *end-of-file* condition. Figure 5-2 below summarizes common ways to test for the end of a file.

Testing for End of File



Common end of file tests –

- The "lines" function
- The "chars" function

Less common end of file tests –

- Scan for a known value
  (eg, user enters a null line to the script,
  or a value like "END" or "EXIT")
- The "stream" function
- SIGNAL ON NOTREADY error condition trap

Figure 5-2

The script uses the conversion function `c2x` to convert each input character into its hexadecimal equivalent. This displays the byte code for these characters:

```
out_string = '  ' c2x(charin(filein))      /* convert it to hex   */
```

This script illustrates the `charout` function twice. The first time it includes a comma to replace the output filename, so the character is written to the default output device (the display screen). The second `charout` function includes an output filename and writes characters out to that file:

```
call charout ,out_string                   /* write to display    */
call charout fileout,out_string            /* write to a file too */
```

Let's take a look at some sample output from this script. Assume that the input file to this script consists of two lines containing this information:

```
line1
line2
```

The hexadecimal equivalent of each character in the character string `line1` is as follows:

```
     l   i   n   e   1
     6C  69  6E  65  31
```

With this information, we can interpreter the script's output. The script output appears as shown, when run under Linux, Unix, Windows, DOS, and the MacOS. Linux, Unix, and BSD terminate each line with a line feed character (x'`0A`'). This character is also referred to as the *newline* character or sometimes as the *linefeed*. Windows ends each line with the pair of characters for carriage return and line feed (x'`0D0A`'). DOS does the same as Windows, while the Macintosh uses only the carriage return to mark the end of line:

```
     Linux:   6C  69  6E  65  31  0A  6C  69  6E  65  32  0A
     Unix:    6C  69  6E  65  31  0A  6C  69  6E  65  32  0A
     Windows: 6C  69  6E  65  31  0D  0A  6C  69  6E  65  32  0D  0A
     DOS:     6C  69  6E  65  31  0D  0A  6C  69  6E  65  32  0D  0A  1A
     MacOS:   6C  69  6E  65  31  0D  6C  69  6E  65  32  0D
```

Some operating systems mark the end of the file by a special *end-of-file character*. This byte occurs once at the very end of the file. DOS is an example. It writes its end-of-file character Control-Z or x'`1A`' at the very end of the file. Windows operating systems may optionally contain this character as the last in the file (for compatibility reasons) but one rarely sees this anymore.

This example shows two things. First, what Rexx calls *character I/O* is really "byte-oriented" I/O. Bytes are read one by one, regardless of their meaning to underlying operating system and how it may use *special characters* in its concept of a file system. Rexx character I/O reads every byte in the file, including the end-of-line or other special characters.

Second, character I/O yields platform-dependent results. This is because different operating systems manage their files in different ways. Some embed *special characters* to denote line end, others don't, and the characters they use vary. Character I/O reads these special characters without interpreting their meanings. Line-oriented I/O strips them out. If you want only to read lines of data or I/O records in your script, use line-oriented I/O. If you need to read *all* the bytes in the file, use character I/O.

Character I/O is easy to understand and to use. But it is often platform-dependent. If you're concerned about code portability, be sure to reference the operating system manuals and code to handle all situations. Or, stick to line-oriented I/O, which is inherently more portable.

Conversational I/O

A user interaction with a script is termed a *conversation* or *dialogue*. The interactive process is called *conversational I/O*. When writing a Rexx script that interacts with a user, one normally assumes that the user sees program output on a display screen and enters input through the keyboard. These are the default input and output streams for Rexx.

To output information to the user, code the `say` instruction. As we've seen, the operand on `say` can be any expression (such as a list of literals and variables to concatenate). `say` is equivalent to this call to `lineout`, except that `say` does not set the special variable `result`:

```
        call  lineout  , [expression]
```

The comma indicates that the instruction targets *standard output*, normally the user's display screen.

Use `pull` to read a string from the user and automatically translate it to uppercase, or use `parse pull` to read a string without the uppercase translation. Both instructions read user input into a *template*, or list of variables. Discard any unwanted input beyond the variable list by encoding a period (sometimes referred to as the *placeholder variable*).

This statement reads a single input string and assigns the first three words of that string to the three variables. If the user enters anything more than three words, Rexx discards it because we've encoded the period placeholder variable at the end of the line:

```
   parse  pull  input_1  input_2  input_3  .
```

Redirected I/O

I/O *redirection* means you can write a program using conversational I/O, but then redirect the input and/or output to other sources. Without changing your program, you could alter its input from the keyboard to an input file. The `pull` or `parse` instructions in the program would not have to be changed to make this work. Similarly, you could redirect a script's `say` instructions to write output to a file instead of the display screen, without changing your program code.

Here is how to redirect I/O. Just run the script using the redirection symbols shown in this table:

Redirection Symbol	Meaning
>	Redirects output to a new file. Creates a new file or overwrites an existing file if one exists with that filename.
>>	Appends (adds on to) an existing file. Creates a new output file if one does not already exist having the filename.
<	Redirects input from the specified file

How's how to invoke the Four-Letter Words program of Chapter 3 with input from a file instead of the keyboard:

```
regina  four_letter_words.rexx  <four_letter_words.input
```

The file `four_letter_words.input` consists of one word per line (so it conforms to the program's expectation that it will read one word in response to each prompt it gives). Here's how to give the script input from a file and redirect its output to a file named `output.txt` as well:

```
regina  four_letter_words.rexx  <four_letter_words.input  >output.txt
```

Redirected I/O is a very powerful concept and a useful testing tool. You can write programs and change their input source or output destination *without changing the script*!

But redirection is operating-system-specific. Operating systems that support redirected I/O include those in the Linux, Unix, BSD, Windows, and DOS families.

A warning about Windows — members in the Windows family of operating systems do not handle I/O redirection consistently. Different versions of Windows handle I/O redirection in slightly different ways. This has long been an issue for programmers who want their programs to run across many Windows versions. This is not a Rexx issue, but rather an inconsistency in the behavior of Windows operating systems. If you rely on redirection under Windows, you will have to test your scripts on each version of the operating system they run on to ferret out any Windows inconsistencies.

I/O Issues

I/O is operating system dependent and thus presents a difficult issue for any programming language. The reason is the inherent tension between an I/O model that is easy to use, easy to understand, and portable — versus the desire to take advantage of operating-system-specific features for file system manipulation.

Rexx always promotes ease of use and portability. Fitting with this philosophy, simplicity trumps OS-specific features and maximizing I/O performance. So, the ANSI standard Rexx I/O model is simple and portable. It does not take advantage of OS-specific I/O features or optimize I/O by platform.

Standard Rexx recognizes the trade-off between I/O portability and OS-specific I/O features by including functions such as `stream` and the `options` instruction, which are open ended and permit operands beyond the ANSI standard. This allows Rexx interpreters to add I/O extensions within the context of the ANSI standard that go beyond the standard to leverage OS-specific features.

The second section of the book describes the I/O extensions that different Rexx interpreters provide to leverage OS- specific I/O features. *Remember that all Rexx interpreters, whatever addtional I/O extensions they offer, still provide the standard Rexx line-oriented and character-oriented I/O described in this chapter.*

This chapter assumes the user interface to consist of a screen display and keyboard, and that disk I/O means manipulating data residing in files. Of course, many programs require more advanced I/O and different forms of user interfaces. Upcoming chapters cover these topics. Chapters 15 and 16, for example, describe and illustrate both database I/O and screen I/O using various GUI packages. Chapter 17 discusses Web interfaces for Rexx scripts. Section II illustrates the I/O extensions in many Rexx interpreters that provide more sophisticated file processing.

Summary

This chapter provides an overview of the Rexx I/O model and how it is implemented in standard functions for line- and character-oriented I/O. We discussed conversational I/O and how to redirect I/O under operating systems that support it. Redirection is a powerful debugging tool and provides great flexibility, because the source of input and target for output for scripts can be altered *without changing* the scripts themselves. The flexibility that redirection provides is very useful during script testing and debugging.

Two I/O related topics will be covered in upcoming chapters. The *external data queue* or *stack* is an area of memory that can be used to support I/O operations. The second important topic is I/O error handling. Both are covered in future chapters.

Upcoming chapters also cover I/O through interface packages, such as databases, GUI screen handlers, Web server interfaces, and similar tools.

Test Your Understanding

1. What are the two basic kinds of standard Rexx input/output? Why would you use one approach versus the other? Which is most portable across various operating systems?
2. What kinds of file control commands can you issue through the `stream` function? Do these vary by Rexx implementation? What file statuses does the `stream` function return?
3. Describe the two ways in which you can invoke an I/O function like `linein` or `charout`. How do you capture the return code from I/O functions? What happens if you fail to?
4. Do you need to close a file after using it? Under what conditions might this be appropriate? How is it done?
5. If you require very powerful or sophisticated I/O, what options does Rexx offer?