Rexx File Utility Functions

Patrick TJ McPhee (ptjm@interlog.com)
DataMirror Corporation

20 April 2002

Contents

1 Introduction

This package defines a set of functions which allow interaction with running processes using pipes. The idea is to have some special kinds of files using an interface much like linein and lineout. This could actually be done using system-dependent parameters to the stream() function, however there is no portable way to extend the stream() interface for all possible interpreters. In the future, I plan to provide support for handling files compressed with gzip, and the tar and zip archive formats, which will be accessible through the same interface.

Although the library is ultimately meant to support a variety of special file formats, it isn't there yet, and I'm in a rush, so this documentation will mostly talk about reading and writing to processes.

The interface is modelled on the standard Rexx I/O functions. The library provides an open and a close routine, however it is not necessary to 'open' a file before reading from or writing to it. The open function exists primarily to allow two processes which have exactly the same command-line to run concurrently. Processes which are started using the read and write functions are identified by their command line, so the first write to 'sort -n' will start a new process, and the second write will go to the same process. The open function returns a handle which is independent of the command being run, which would allow two sorts to go on concurrently.

2 Function Descriptions

2.1 List of Routines

FileOpen	(file,how) → handle: starts a command running, and returns a handle;
FileClose	(file[,stream]) → 0 or failure: either closes all open streams to a process and waits for the process to exit, or closes the specified stream;
FileLineIn	(file[,[line][,[count]][,stream]]) → data: reads a line from either the standard output or standard error stream of a process;
FileCharIn	(file[,[line][,[count]][,stream]]) → data: reads *count* characters from either the standard output or standard error stream of a process;
FileLineOut	(file[,[string][,line]]) → 0 or count: writes a string and a new-line to the input stream of a process;
FileCharOut	(file[,[string][,line]]) → 0 or count: writes a string to the input stream of a process;
FileLines	(file[, stream]) → 0 or 1: tells whether there's anything left to be read from a stream;
FileChars	(file[, stream]) → 0 or 1: tells whether there's anything left to be read from a stream.

2.2 FileOpen

```
handle = FileOpen(file,how)
```

Open a file using method *how*. Currently, *how* must be 'pipe' and *file* is the command to execute. *handle* is a 10-digit number which should be passed in the *file* argument to the other functions.

You don't have to call FileOpen() for most operations. If you don't call FileOpen(), and pass the file name to the other functions, they will open the file if it is necessary and it makes sense to do so. The only time fileopen() is needed is if the exact same file needs to be opened more than one time concurrently. In this case, *handle* differentiates between the two files. When open methods other than 'pipe' are available, FileOpen() will be required to open files which are not of the default type (there may be a new function to change the default type from 'pipe' to whatever's most convenient for the program at hand. We'll see.)

For the ‘pipe’ open method, *file* is the command to execute. Under NT, any string which can be passed to CreateProcess() is acceptable. So far as I know, this means spaces delmit arguments, and double-quotes can be used to create arguments with spaces in them. ˆ can be used to escape quotes.

Under Unix, the library parses the command using essentially the same rules as the Bourne shell. Spaces delimit arguments, and either single- or double-quotes can be used to create arguments with spaces in them. `\` is the escape character, and it escapes either spaces or quotes. Within single-quotes, there is no escape character. Thus `sort file\ with\ spaces`, `sort "file with spaces"`, and `sort ’file with spaces’` all invoke sort with the single argument ‘file with spaces’. However, `sort "file with \"quote\""` refers to a file `file with "quote"` while `sort ’file with \"quote\"’` refers to a file `file with \"quote\"`. Don’t be too clever, and you can’t go wrong.

2.3 FileClose

```
rc = FileClose(file[,stream])
```

If *stream* is not specified, FileClose() closes all streams to the process specified by *file* then waits for the process to exit. *file* can be a command name or a handle returned by FileOpen(). In this case, FileClose() returns the process return code.

If *stream* is specified, it should be ‘in’, ‘out’, or ‘error’, and it closes the corresponding stream. Note that ‘in’ refers to the process’s input stream, which is what the rexx program might have been writing to. This seems a bit counter-intuitive at first, but I think it makes sense. Similarly, ‘out’ refers to the process’s standard output stream, and ‘error’ refers to its standard error stream. Many filters won’t start writing until you’ve fed them an awful lot of data, or you’ve closed their input stream.

2.4 FileLineIn

```
data = FileLineIn(file[,[line][,[count]][,stream]])
```

Reads a line from either the standard output or standard error stream of a process. *file* can be either a command or a handle returned by FileHandle(). See FileOpen() for information about the command format.

If *line* is specified, the running process is ended, and a new process is started. The following

```
 cmd = ’sort -n’
 call FileClose cmd
 data = FileLineIn(cmd)
```

is equivalent to

```
 cmd = ’sort -n’
 data = FileLineIn(cmd, 1)
```

If *count* is specified, it should be 0 or 1. 1 means to read a line, 0 means to evaluate the function for side-effects (starting or re-starting the command).

stream can be ‘out’ to read from the standard output, or ‘error’ to read from standard error. The default is ‘out’.

If there are no characters to read but the stream is still open, FileLineIn() will wait until characters are available. If you try to read and write to and from the same running process, you can deadlock yourself (*i.e.*, you can be waiting for the other process to write something at the same time it’s waiting for you to write something).

On error, FileLinIn() returns an empty string.

2.5 FileCharIn

```
data = FileCharIn(file[,[line][,[count]][,stream]])
```

This is just like FileLineIn(), except *count* is the number of characters to read.

2.6 FileLineOut

```
count = FileLineOut(file[,[string][,line]])
```

Writes a string and a new-line to the input stream of a process. *file* can be the command to run or a handle returned by FileOpen(). *string* is the string to write.

If *line* is specified, the process is re-started. See FileLineIn() to see what I mean by that.

If *string* is not specified, the function is evaluated for its side-effects (starting or re-starting the process).

2.7 FileCharOut

```
FileCharOut(file[,[string][,line]])
```

FileCharOut() is just the same as FileLineOut() except a new-line is not written to the output stream.

2.8 FileLines

```
rc = FileLines(file[,stream])
```

Returns 0 if there is nothing to be read from the input stream, or 1 if there is something to be read. *file* can be a command name or a handle returned by FileOpen(). *stream* can be ‘out’ for standard output (the default) or ‘error’ for standard error.

2.9 FileChars

```
rc = FileChars(file[,stream])
```

FileChars() is exactly the same as FileLines().