REXX I/O ON VM

GARY BRODOCK
IBM

REXX I/O on VM

Gary Brodock

IBM G09/16-4
P.O. Box 8009
Endicott, NY 13760
607-752-5134

5/18/93

Contents REXX I/O

GKB

Introduction REXX I/O

- REXX Language statements for input and output
- The VM support follows the defined REXX I/O model
  - Seven new builtin functions
  - A new variant on the PARSE instruction
- Additional related support
  - A new NOTREADY condition
  - SIGNAL and CALL enhancements
- The VM I/O model

GKB

Introduction /Notes REXX I/O

The definition for input and output is published in "The REXX Language" by Mike Cowlishaw. It was implemented in the OS/2 operating system in 1990 and this definition is what will be followed for VM. The goal is to provide input and output capability through REXX language statements on the VM platform.

I/O is defined through the use of seven new builtin functions, three for character based operations, three for line based operations, and one for general stream functions. In addition, a new variant for PARSE, PARSE LINEIN, is added to read a line from the default input stream and parse the contents according to the template.

One new condition (in addition to ERROR, HALT, SYNTAX, and NOVALUE) is provided to handle error conditions while doing I/O. This is the NOTREADY condition and it is supported by the SIGNAL and CALL instructions for error handling.

GKB

The General I/O Model

REXX I/O

- CMS is a line based system
- Line based operations are easy
- Character based operations need special support
  - Need to buffer data
  - Perhaps need to indicate line ends
- Two I/O pointers, one buffer
  - Can cause extra I/O when doing both input and output in the same data stream

GKB

The General I/O Model /Notes

REXX I/O

All CMS functions read and write lines of data, not characters. Since the line functions read and write complete lines, they fit into the CMS model very well. When character operations are requested, special processing must be done to properly interface with the line oriented operating system. On input, a data stream line is read and placed in a buffer. Then, characters are given to the user as requested. If more characters are requested than are available, another data line is read and processing continues. On output, the characters supplied by the user are placed in the buffer and written as a line to the data stream only under certain conditions. These conditions are: either I/O pointer is changed, a line end character is given by the user and the data stream is in TEXT mode (explained later), the stream is closed, or an entire buffer of data is given. Otherwise, the data just remains in the buffer and gets added to on the next output operation.

TEXT mode is specified on the STREAM function when opening a data stream. The meaning of this option is that line end characters are significant when doing character based I/O. On input, a line end character is appended to the data at the end of each line read in. On output, the character string to be written is scanned for line end characters and appropriate lines are written as they are found. If any character can be in the data stream, then the user should open the stream with the BINARY option, signifying that line end checking should not be done.

Since there are two I/O pointers and only one buffer, doing both character based input and output on the same stream can cause some flip-flop of the buffered data. This could cause performance degradation.

GKB

Line functions

REXX I/O

- LINEIN(<name><,<line><,count>>)

  returns 0 or 1 lines from the input stream
  - name - the stream name
    line - the line number to read
    count - 0 or 1, the number of lines to read
- PARSE LINEIN template
- LINEOUT(<name><,<string><,line>>)

  returns the number of lines not written (0 or 1)
  - name - the stream name
    string - the line to be written
    line - the line number to write
- LINES(<name>)

  returns the number of lines remaining in the input stream
  - name - the stream name

GKB

Line functions /Notes

REXX I/O

The three line functions are used when the I/O is to be performed a line at a time. LINEIN will read COUNT (either 0 or 1) lines from the input stream, NAME. This will by default read the line at the current read position unless another line is specified by the LINE parameter. If zero is specified for the COUNT, then the current read position is set to LINE and nothing is read from the stream. If part of a line has already been read with the CHARIN function and LINE is not specified on the command, LINEIN will return only the remainder of the line.

The new variant of PARSE, "PARSE LINEIN template", can be used to read a line from the default input stream and parse it according to the supplied template. This is a shorter form of "PARSE VALUE LINEIN() WITH template"

LINEOUT will write the line contained in the STRING parameter to the stream NAME. This write will by default start at the current write position unless a new position is specified by the LINE parameter. If there are characters in the buffer from a previous CHAROUT and a new line is not specified on the LINEOUT call, the STRING is appended to the characters already in the buffer and then the entire line is written. If STRING and LINE are both omitted, the stream will be closed.

The LINES function will return the number of completed lines remaining in the input stream. This count may include a partial line if the stream has been read with the CHARIN function.

GKB

The Character functions

- CHARIN(<name><,<start><,length>>)

  returns LENGTH characters from the input stream

  - name - the stream name
    start - starting character number (only 1 is valid)
    length - number of characters to read

- CHAROUT(<name><,<string><,start>>)

  returns the number of characters not written

  - name - the stream name
    string - the string of characters to write
    start - starting character number (only 1 is valid)

- CHARS(<name>)

  returns 0 if no more characters, 1 if there are

  - name - the stream name

The Character functions /Notes

The three character based functions are used when the I/O is to be done as a string of characters and not whole lines. The CHARIN function will read in a specified number of characters, LENGTH, from a stream, NAME. Optionally, a start position of 1 can be specified to reset the read position to the beginning of the stream. CHAROUT will write a string of characters, STRING, to a stream, NAME. Again, optionally, a start position of 1 can be specified to start writing at the beginning of the stream. A read and a write position is maintained for each persistent stream and if not reset, reading and writing will start from these positions. The CHARS function can be used to indicate if there are more characters in the input stream NAME.

If the name of the stream is omitted, characters are read or written to the default stream. If the stream was opened with the TEXT option, then a LINEEND character is appended to the input characters at the end of each line. For output operations on a stream opened with the TEXT option, the string is scanned for LINEEND characters and lines are written as appropriate. The LINEEND character is not written to the stream in this case. For streams opened with the BINARY option, no indication of line ends is given on input and output records are written when the buffer is filled.

The STREAM function

- STREAM(name<,action<,stream_command>>)

  - returns a string describing the state of, or the result of an operation upon, the stream

  - action - Command, State, or Description

    Command - perform the stream_command given on the named stream

    State - return a string indicating the state of the named stream

    Description - same as State, with additional information: return code and reason code from the last I/O

  - stream_command - specific command to be performed on the named stream

The STREAM function /Notes

The STREAM function is used to get the status of a stream or to perform an operation on a stream. Users can request the State, request the Description, or issue a Command. State will return only the current state of the stream. Description will return the state and also the return and reason codes from the last I/O done on the stream. If Command is specified for ACTION, then a STREAM_COMMAND must be given. The various commands are described on a later foil.

Stream states — REXX I/O

- ERROR - an I/O has caused an error condition
- NOTREADY - an I/O has made the stream not ready and I/O to that stream could raise the NOTREADY condition
- READY - the stream is ready for I/O
- UNKNOWN - the stream is closed or has not been opened yet

Stream states /Notes — REXX I/O

There are four states that a stream can be in. ERROR means that the stream was subject to an erroneous operation, such as a disk problem when writing to a minidisk file. NOTREADY is similar to ERROR, only it usually means that recovery is easier. This could be the case when the user tries to set the read or write pointer to a nonexistent line in the stream or to read past the end of the stream. Recovery would be to reset the pointer to a valid place and then continue the input or output. READY specifies that the stream is ready for I/O operations but does not guarantee that an operation will succeed. UNKNOWN specifies that REXX does not know the state of the stream, such as when the stream does not exist or it has not been opened yet.

Stream_commands — REXX I/O

- OPEN <options>
  - READ/WRITE/NEW/REPLACE
  - LRECL nnnn
  - TEXT/BINARY
  - LINEEND xx
- CLOSE
- LINEPOS offset type
  - offset is a whole number optionally preceded by =, <, +, or -
  - type is READ or WRITE
- QUERY option
  - DATETIME
  - EXISTS
  - FORMAT
  - INFO
  - LINEPOS READ
  - LINEPOS WRITE
  - SIZE

Stream_commands /Notes — REXX I/O

OPEN: is used to open a stream, with additional options to tell the characteristics that are desired.

READ: specifies that the stream will be opened for read only and it must exist.

WRITE: specifies that the stream will be opened for read/write and it will be created if it doesn't exist.

NEW: specifies the stream will be opened for read/write and it must not exist already.

REPLACE: specifies that the stream will be opened for read/write and will be created if it doesn't exist or be replaced (old version thrown away) if it does.

LRECL: indicates the size of the buffer that will be used for input or output. Most of the time, for an existing stream, the current lrecl will be used. However, for new streams or certain existing streams, this is not known and users will have to specify it or take the default of 1024.

TEXT: specifies that line end characters are significant when doing character based operations. This means that a LINEEND character is appended to the input string at the end of each line as an indication that the line is complete. On character output operations, lines are written when *the LINEEND character is encountered in the string.*

BINARY: means that all character codes may be present in the data stream and no indication of LINEEND characters will be provided or searched for. Line based operations are not affected by this option.

LINEEND: specifies the character to be used to indicate line ends. This can be specified as one or two hexadecimal digits with the default being 15.

CLOSE: is used to write out any data left in the buffer due to a character output operation and close the stream.

LINEPOS: is used to change the read or the write line pointer to the beginning of a specified line. The specification can be just a number, such as 10, meaning to move the pointer to line 10. The same move can be made using =10. If you want to move to an offset from the end of the stream, use the <. <0 means point just past the end of the stream, <1 means point at the last record, etc. Relative offsets from the current position are done with the + and - prefixes.

QUERY: the remainder of the commands are various querys to obtain information on a stream. DATETIME gives the date and time that the stream was last modified. EXISTS returns the fully qualified name if the stream exists, FORMAT returns the record format and the logical record *length of the stream, INFO returns format data, size data, and date/time,* LINEPOS READ/WRITE return the current position of the read or write pointer, and SIZE returns the number of lines in the stream.

VM Specific Stream names — REXX I/O

- Reader file - nnnn RDRFILE CMSOBJECTS.
- Punch - VIRTUAL PUNCH CMSOBJECTS.
- Printer - VIRTUAL PRINTER CMSOBJECTS.
- SFS file - filename filetype dirname
- Minidisk or accessed SFS file - filename filetype filemode
  - filemode is optional for an input file, or can be *
  - filemode must be specified for an output file
- Program stack - PROGRAM STACK CMSOBJECTS.
- Default stream - no name specified or name is null
- Or may use the unique ID returned on the OPEN command

GKB

VM Specific Stream names /Notes — REXX I/O

Case is insignificant when specifying the names for the reader, punch, printer or program stack. When specifying a minidisk file, an accessed SFS directory file or an SFS file, the case is significant (thus allowing you to process files with mixed case names). As a point of clarification, a little background is necessary on the names we have choosen. We are working on a standard form of I/O that accesses a variety of data streams. In creating this standard I/O model, we wanted to have names that would totally describe the data stream. That is, if you know the data stream name, you know its characteristics. The stream names used by REXX I/O are part of this model.

Reader - nnnn is the spool file number that you want to process. Specifying an asterisk for nnnn means to use the first file in the reader. Normal rules apply as to reader class and the class of spool files.

SFS file - the directory does not have to be accessed, all you have to do is specify the directory name - "dirname". Wild card characters are not permitted.

minidisk or accessed SFS file - You can omit the file mode if you are working with an input file. If you are working with an output file, you must give the file mode. Wild card characters are not permitted.

Program stack - The default for this is FIFO stacking for output. You can add FIFO or LIFO to the name to explicitly use FIFO or LIFO stacking. The name for LIFO stacking would be "PROGRAM STACK CMSOBJECTS.LIFO". The name for FIFO stacking would be "PROGRAM STACK CMSOBJECTS.FIFO".

default stream - This is the terminal input buffer for input and the users display for output. On input, if there are no lines in the terminal input buffer, a VM READ results. You can omit the name on the I/O functions (except STREAM) to specify the default stream. You can also use a null name on all functions to specify it.

When you use the STREAM function to open a data stream, the returned string on a successful open contains the string READY: followed by a unique ID. This ID can be later used on all I/O function calls in place of the name, and it will speed up processing. An example of obtaining and using the unique ID is as follows:

```
/* show use of the unique ID    GKB 5/93 */
parse value stream('TEST FILE A1','c','open read') with ok id
if ok ¬= 'READY:' then signal open_error
say linein(id)  /* will read and display a line from TEST FILE A1 */
```

GKB

Additional information — REXX I/O

- All I/O is done by calling CSL routines
  - these routines pass back a return code and a reason code on every call
- The STREAM(name,'D') command can be used to get these codes when errors occur
- NOTREADY traps should be used to handle error conditions
  - both SIGNAL ON and CALL ON are supported
  - the CONDITION function can be used to get important information

GKB

Additional information /Notes — REXX I/O

REXX uses calls to CSL routines to do the actual I/O. A return code and a reason code is always passed back from the CSL routine and this information is kept in the data stream control block. Using the STREAM function with a request for a "Description" will return these codes to the user.

It is also a very good idea to have a NOTREADY trap set up in your program to handle the error conditions as they occur. SIGNAL ON NOTREADY and CALL ON NOTREADY are both supported. While in the NOTREADY processing routine, the CONDITION function can be used to retrieve the error string passed back from the I/O routine. This string contains the return and reason codes from the error condition and will show exactly why the error occurred.

GKB

Examples — REXX I/O

```
/* This routine copies the stream or */
/* file named by the first argument  */
/* to the stream or file named by    */
/* the second, as lines.             */

parse arg inname, outname

do while lines(inname)>0
  call lineout outname, linein(inname)
end
```


Examples ... — REXX I/O

```
/* This routine collects characters  */
/* from the stream named by the       */
/* first argument until a line is    */
/* complete, and then places the     */
/* line on the external data queue.  */
/* The second argument is the single */
/* character that identifies the end */
/* of a line.                        */

parse arg inputname, lineendchar

buffer=''  /* initialize accumulator */

do forever
  nextchar=charin(inputname)
  if nextchar=lineendchar then leave
  buffer=buffer || nextchar
end
queue buffer  /* place on data queue */
```


Examples ... — REXX I/O

```
/* Read the first line of the input  */
/* file and get the number of lines  */
/* in the file.  Generate a random   */
/* number from 2 to the number of    */
/* lines in the file and then read   */
/* that line number.                 */

infile = 'SAYINGS SCRIPT A'
parse value stream(infile,'c','open read'),
          with ok handle
if ok ¬= 'READY:'
  then do
         'MSG Error in opening' infile
         'MSG Description string =',
                      stream(infile,'d')
         exit 100
       end

how_many = word(linein(handle,1),1)
num = random(2,how_many)
saying = linein(infile,num)
call lineout(infile)
```


Summary — REXX I/O

- Native REXX language Input/Output
- The general VM I/O model
- Three line based functions
  - a line based variant of the PARSE instruction
- Three character based functions
- A STREAM function for minipulation of a data stream
  - stream states
  - stream commands
- Stream names in VM
- Additional information