z/OS Piping in 10 Easy Stages
--- Draft ---

Combining the best of UNIX, CMS/TSO Pipelines and REXX in z/OS

Notes on Draft Version

The code here has been modified based on full-version testing. If you run it, you may find minor issues and/or bugs. I will attempt to apply the fixes here.

I’m interested in getting feedback on the technique, primarily, and its presentation, secondarily. Comments questions and/or suggestions are welcome.

Bug reports are welcome in any event.

Since my first attempt, It been 30 years coming. Sorry it took so long.

Contents

Introduction to Piping

Piping originated in UNIX, where hardware was typically constrained by the budgets of academic and research organizations. Piping provided productivity and efficiency via an operator (|) that connected the output of one program to the input of the next thru memory.

```
ls | wc -l
```

This saved disk input and output, which were very much slower than the processors of the day.

Some platforms have piping facilities, i.e. Windows, Linux, and z/VM. On EBCDIC based systems, where data is passed with pointers, piping of binary and/or volume data is practical.

z/OS currently lacks a built-in piping facility. (BatchPipes is optional and chargeable.) Here we present a rudimentary piping facility and a simple technique for writing filters.

What is REXX?

REXX is a full-functioned, all-purpose programming language that is small and easy to use. REXX borrows from the best of other programming languages, especially PL/I. You can hit the ground running.

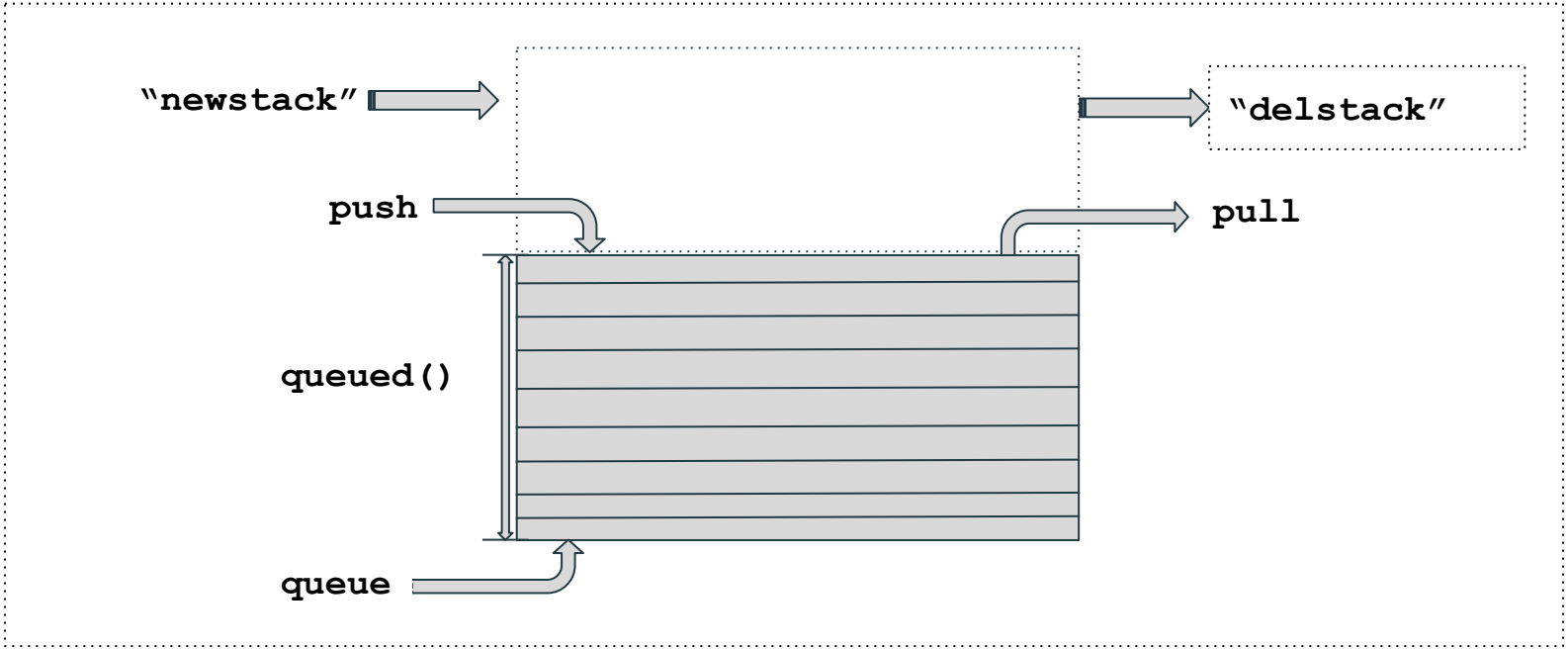
- Numeric and string constants are typical: 10, 3.14159, -0.333, and “The quick brown fox …”.
- Variables follow conventions, for example: Quantity, Price, InterestRate, SocSecNo, and Group10.
- Arithmetic, comparison, and logical operators are familiar: +, -, *, /, =, >, <, >=, <=, |, and &.
- Assignment of a value to a variable is well known: `Circumference = 3.14159 * Radius`
- REXX’s if-then-else follows the usual practice:

```
if Divisor > 0 then Quotient = Dividend / Divisor
else Quotient = “”
```

- Loops can have optional iterative and logical conditions; commentary is enclosed in /* and */:

```
do iCount = 1 to 10 while Item > 0
    Total = Total + Item /* Add to the Total. */
    end iCount
```

The REXX Stack



Piping With REXX

Here we present a simple way to add piping to z/OS using REXX: Each stage or filter takes it's input from the REXX data stack and places its output on the bottom of the stack stack.

A stage performs some specific transformation on the stack and/or it's records. The resulting (output) stack records are available to the next stage or program executed.

Instead of the argument syntax of UNIX/Linux or CMS/TSO Pipelines, we use that of REXX.

- REXX has a rich, well-defined, general set of operators and functions.
- Interpret lets us process operands with efficiency and minimal additional code.

Design Notes

1. The output of one program is passed to the next using the stack. Volume testing strongly suggests the stack is a chain of pointers.
2. Shown stages are single stream. Multiple stack passing is up to user.
3. Like UNIX/Linux, stages can be entered from the command line or invoked from a program. Commands without a pipe write to the terminal; see below slide
4. Unlike UNIX/Linux, records are passed by pointers. Unchanged data do not go thru the CPU.
5. Like CMS/TSO Pipelines, output does not go to the terminal by default. (Exception: #3 above.)
6. Unlike Pipelines, all records are delayed; stages are dispatched only once.
7. Operand expressions use REXX syntax, avoiding complexity and multiple selection stages.

TSO Stage - Write line mode output to stack.

```
/* REXX - Send TSO command output to the stack.                    */
  parse source Source 1 OpSys CallType ExecName ExecDD ExecDS Rest
  parse arg TSOCmd "|" NextStage
  call outtrap "CmdOut."             /* Capture line mode output.   */
  (TSOCmd)
  call outtrap "off"
  do iL = 1 to CmdOut.0
      queue CmdOut.iL
      end iL
  if CallType == "COMMAND" then
      exit zPipe("stack |" NextStage, Source) /* Start stack manager. */
  exit 0
```

zPipe - Manages the stack, handles errors.

```
/* REXX - Provide transition between stages and manager the stack.    */
  signal on syntax
  parse source OpSys CallType ExecName ExecDD ExecDSN UsedName Rest
  if ExecDSN <> "?" then "altlib act appl(exec) dataset('"ExecDSN"')"
  call Table.Put "Alias.", "< StackIO <", "> Stack >", ">> StackIO >>",,
    "terminal LineMode", "term LineMode", "call RXCall"
  parse arg PipeSpec, ClrSrce
  do StageNum = 1 by 1 while PipeSpec <> "" & result >= 0
      parse var PipeSpec StgPgm StgArg "|" PipeSpec
      StgPgmUC = translate(StgPgm)
      if symbol("Alias.StgPgmUC") == "VAR" then
          StgPgm = Alias.StgPgmUC
      if StgPgmUC \== "STACK" | StageNum > 1 & PipeSpec <> "" then do
          if StageNum = 1 then "newstack" /* Protect caller's stack.  */
          if word(ClrSrce, 2) == "COMMAND" & PipeSpec = "" then
              PipeSpec = "terminal"       /* Write to terminal.       */
          interpret "call" StgPgm StgArg  /* Invoke stage as SUBROUTINE. */
          end
      End StageNum
  if StgPgmUC <> "STACK" then "delstack"  /* Restore caller's stack.   */
  exit result
```

zPipe (cont.)

```
Table.Put:
  parse arg TblName .
  do iArg = 2 to arg()
       parse value arg(iArg) with Tail Val
       call value TblName || translate(Tail), Val
       end iArg
  return

syntax:
  say errortext(rc) "in stage" StageNum":" StgPgm StgArg
  say "zPipe deleting" queued() "records so TSO doesn't execute then,"
  "delstack"
  exit rc
```

TERMINAL stage (Alias of LineMode)

```
/* REXX - Send stack contents to the screen.                          */
  parse source Source 1 OpSys CallType ExecName ExecDD ExecDS Rest
  parse arg "|" NextStage
  do queued()                          /* Unindexed repetitive loop. */
      parse pull Rcd                   /* Get record from the stack. */
      say Rcd                          /* Display it.                */
      queue Rcd                        /* For next stage.            */
      end
  if CallType == "COMMAND" then
      exit zPipe("stack |" NextStage, Source) /* Start stack manager. */
  exit rc
```

PICKIF - A Generic Selection Stage.

```
/* REXX - Write records to the stack for which TFExpr is true.        */
  signal on syntax
  parse source Source 1 OpSys CallType ExecName ExecDD ExecDS Rest
  parse arg TFExpr "|" NextStage
  parse value "| ||" with Or SCat
  do queued()
      parse pull Rcd 1 W1 W2 W3 W4 W5 W6 W7 W8 W9 W10 Rest
      interpret "Bool =" TFExpr /* REXX does the heavy lifting here.  */
      if Bool then
          queue Rcd
      end
  if CallType == "COMMAND" then
      exit zPipe("stack |" NextStage, Source) /* Start stack manager. */
  exit 0

syntax:
  say errortext(rc) "in PickIf" TFExpr
  exit rc
```

zPipe in Action

```
     1 >>> "%query"
* ZPipe - Stack piping tool.  Ver. 01.0 beta.

     2 >>> "%literal testing | terminal"
testing

     3 >>> "%zPipe tso status REXXMAN | PickIf wordpos(executing, Rcd) > 0 | terminal"
IKJ56211I JOB REXXMAN(TSU09714) EXECUTING
```

SPEX will let us reformat records.

```
/* REXX - Write Expr evaluation of records to stack.                  */
  signal on syntaX
  parse source Source 1 OpSys CallType ExecName ExecDD ExecDS Rest
  parse value "| ||" with Or SCat
  parse arg Expr "|" NextStage
  do queued()
      parse pull Rcd 1 W1 W2 W3 W4 W5 W6 W7 W8 W9 W10 Rest
      interpret "queue" Expr /* REXX does the heavy lifting here.     */
      end
  if CallType == "COMMAND" then
      exit zPipe("stack |" NextStage, Source) /* Start stack manager. */
  exit rc
syntax:
  say errortext(rc) "in" Expr
  exit rc
```

RXCALL lets functions be filters, even yours.

```
/* REXX - Write result of function call to stack.                    */
  signal on syntax
  parse source Source 1 OpSys CallType ExecName ExecDD ExecDS Rest
  parse value "| ||" with Or SCat
  parse arg Funct Args "|" NextStage
  do queued()
      parse pull Rcd 1 W1 W2 W3 W4 W5 W6 W7 W8 W9 W10 Rest
      interpret "call" Funct "Rcd," Args /* REXX does heavy lifting here. */
      if symbol("result") == "VAR" then  /* Did it return a value?        */
          queue result                   /* If so, write to stack bottom. */
      end
  if CallType == "COMMAND" then
      exit zPipe("stack |" NextStage, Source) /* Start stack manager. */
  exit rc

syntax:
  say errortext(rc) "in" arg(1)
  exit rc
```

More zPipe in Action

```
     4 >>> "%tso st REXXMAN | call word 3  | term"
REXXMAN(TSU09714)

     5 >>> "%zPipe tso st rexxman | spex translate(W3, '  ', '()') | term"
REXXMAN TSU09714
```

STACKIO reads and write datasets.

```
/* REXX - Read and write records to/from the stack.        */
  signal on error
  parse source Source 1 OpSys CallType ExecName ExecDD ExecDS Rest
  RC = 20
  parse arg Action DataSet AllocOpts
  parse value sysdsn(Dataset) with ResW1 ResW2 ResW3 1 result
  if result <> "OK" & ResW2 <> "NOT" then signal error
  ActNo = wordpos(Action, "< > >>")             /* <:1, >:2, >>:3 */
  if ActNo = 0 then signal error
  if ActNo > 1 & result <> "OK" then /* Provide defaults in case new. */
      AllocOpts = "unit(sysda) cyl space(5 5) recfm(v b) lrecl(5004) blksize(0)" AllocOpts
  parse value subword("shr diskr old diskw mod diskw", ActNo * 2 - 1, 2),
    with Disp Funct
  "alloc reuse dd(StackIO)" Disp "dsname("DataSet") AllocOpts
  "execio *" Funct "StackIO (finis"
  IORC = rc
  "free dd(StackIO)"
  rc = IORC

error:
  if result <> "OK" & (ActNo = 1 | ResW3 <> "FOUND") then say result":" DataSet
  else if ActNo < 1 then say "StackIO: First arg word must be <, >, or >> ."
  exit -rc
```

SPLIT - Divide records into words/strings.

```
/* REXX - Divide records into words or strings.                    */
  parse source Source 1 OpSys CallType ExecName ExecDD ExecDS Rest
  parse arg Str "|" NextStage
  WordParse = (Str = "")
  if \WordParse then interpret "Dlm =" Str /* Support hex, bit, etc. */
  do queued()
      parse pull Rcd
      do while Rcd \== ""
          if WordParse then
              parse var Rcd Word Rcd
          else
              parse var Rcd Word (Dlm) Rcd
          queue Word
          end
      end
  if CallType == "COMMAND" then
      exit zPipe("stack |" NextStage, Source) /* Start stack manager. */
  exit 0
```

JOIN - Concatenate records.

```
/* REXX - Concatenate records.                                  */
  parse source Source 1 OpSys CallType ExecName ExecDD ExecDS Rest
  parse arg N Str "|" NextStage
  if N = "" then N = 1
  if Str = "" then Str = ""  /* In case extra blanks. */
  else interpret "Str =" Str /* Support hex, bit, etc */
  do queued()
      parse pull Rcd
      do N
          parse pull NextRcd
          Rcd = Rcd || Str || NextRcd
          end
      end
  if CallType == "COMMAND" then
      exit zPipe("stack |" NextStage, Source) /* Start stack manager. */
  exit rc
```

Final zPipe Action

```
     6 >>> "%zPipe < zpipe.exec(IsEQ) | call left 72 | stack"

     7 >>> "%zPipe stack | split '/*' | term | PickIf  pos('*/', Rcd) = 0 | term"

 REXX - Case independent compare.     **           ISEQ EXEC HS2112 */
IsEq:
           ISEQ EXEC HS2201 */
  arg Str1, Str2
           ISEQ EXEC HS2112 */
  return Str1 = Str2
           ISEQ EXEC HS2112 */

IsEq:
  arg Str1, Str2
  return Str1 = Str2
```

GetFiles - Read Named Data Sets

```
/* REXX - Load stack named data sets into stack.                  */
  parse arg "|" NextStage
  do queued() until rc < 0
      parse pull DSName
      "%zPipe <" DSName "| stack"
      end

  exit rc
```

UNIX/Linux Behaviour

UNIX/Linux users expect to see command output at the terminal. When a filter is invoked as a command, theaction is preserved by the following.

```
parse source OpSys CallType ExecName ExecDD ExecDS UsedName Rest
…
do queued()
   …
    end
…
if CallType == "COMMAND" then
    exit zPipe("stack |" NextStage, Source) /* Start stack manager. */
…
```

Advantages of the Approach

- Full compatibility with CMS/TSO Pipelines was too challenging.
- Because dispatching does not take place at every record transfer, performance could approach that of TSO Pipelines.
- Single stream piping is sufficient for many uses and serves as an introduction to TSO Pipelines. Multi-stream piping is an advanced skill that builds on single stream piping and introduces new concepts.
- Demonstrations using the code here and/or user written stage may convince your management to install BatchPipes, a.k.a. TSO Pipelines, and/or make use of this technique.
- Programs written to pass data thru the stack integrate well with other programs.
- Writing user filters in TSO Pipelines is an advanced skill with a different command set.
- REXX syntax is consistently used across various types of filters.
- Stages can be invoked from programs without zPipe as functions or subroutines. Place your input inthe stack and get the result(s) from the stack.

Performance

Passing data from program to program using the stack is efficient; the stack is a chain of descriptors to data already in the REXX data heap. Adding or removing stack elements does not move the data itself.

Except for very short records, performance should be better than similar UNIX/Linux commands.

Using REXX **`interpret`** to process arguments means less code, more capabilities and better performance than could be done without **`interpret`**.

Installation and Usage Notes

1. Copy and paste the code above into a library allocated/altlibed to your session, or use explicit invocation the first time.
2. Use the bolded stage names, in the title of each slide, as variable.
3. Any REXX program that uses the stack can be a filter.
4. Any REXX function can be invoked by the **`RXCALL`** stage. For non-builtins, the code will fetched, tokenized, and interpreted repeatedly for each and every record.
5. Self-escaping pipe characters are not implemented. Use the **`Or`** and **`SCat`** variables provided.

Writing Your Own Stages

Stages or filters perform some transformation on the stack. Typically this means selecting records and/or modifying them in some way.

Things to be aware of, in general, when writing your own stages:

- Start with **`parse arg Options "|" NextStage`**
- Loop using **`do queued()`**. (**`do while queue() > 0`** will loop infinitely in most cases.)
  - Get input from the stack with **`parse pull`**.
  - Change, discard, etc. the record.
  - Put any output into the bottom of the stack stack with **`queue`**.
- End with **`if CallType == "COMMAND" then exit zPipe(...).`** This allows the next stage to see the output. TSO will not try to execute the records.

Note

The plan is to create a CBT file distribution (#1035), initially with about 20 modules, mostly stages. Over all, there are about 100 modules total in the fill planned distribution.

You can download a version of this document as a PDF, the source library, and the CEXEC library, using these links. The libraries were unloaded by XMIT using OUTDSN().

PDF document:
https://mail.google.com/mail/u/0?ui=2&ik=abeb8f3322&attid=0.1&permmsgid=msg-a:r8560464330329904716&th=1856e78b34538ecb&view=att&disp=inline&realattid=f_lcdoa3k90

EXEC Library: https://drive.google.com/file/d/1uwq3xeeDSuLzXMyfg_5sRIxft0kbPmiX/view?usp=share_link

CEXEC Library:
https://drive.google.com/file/d/1ZKbHJYAbnzwZ4aN3yOXAbydQdz6oQb1_/view?usp=share_link

Coding Notes

In my code I use these conventions. I have use a version since the 1970's; you may adopt them.

- Names, not part of the language, i.e. variables and labels, are capitalized at each word.
- Names with specific ranges have lower case prefixes. iArg indicates an integer.
- Labels start in column 1.
- Statements start in column 3 and are indented 4 positions for each level of nesting. (:<4 and :>4 are assigned to PF keys 10 and 11).
- Continuations are indented 2 positions.
- **`end`** statements follow execution pathing and are indented 4 positions from their matching statement. This will be familiar to ISPF Panel and Python coders.
- Assignment of logical expression are enclosed in redundant parenthesis.
- Usually I put singleton instructions after **`then`**, **`else`**, and **`otherwise`** on a new line. I did not always do so here due to space limitations.
- Ending comment delimiters (*/) line up in cols. 71-72.

Summary

The above represents a starter implementation of piping for z/OS. The result is a very small subset of TSO Pipelines/BatchPipesWorks, but enough to give your and your management a taste.

The technique has these features:

- User stages and filters can written entirely in REXX.
- The TSO host command environment is the only one required, moderating the learning curve.
- Like CMS/TSO Pipelines data does not move but is passed using pointers.
- Like CMS/TSO Pipelines, both EBCDIC text and binary data can be processed equally well.
- Like UNIX/Linux, COMMAND last stage output defaults to the terminal.

To integrate your own programs into this mechanism, read input from the stack and write output to the stack, using **`queue`**, as described within.