SLAC-PUB-5622
August, 1991
(M)

Pipelines Programming Paradigms: Prefab Plumbing*

Chuck Boeheim

Stanford Linear Accelerator Center
Stanford University, Stanford, CA 94309

ABSTRACT

Mastery of CMS Pipelines is a process of learning increasingly sophisticated tools and techniques that can be applied to your problem. This paper presents a compilation of techniques that can be used as a reference for solving similar problems.

---

* Work supported by Department of Energy contract DE-AC03-76SF00515

Presented at SHARE 77
Chicago, IL, August 19-23, 1991

This document and/or portions of the material and data furnished herewith, was developed under sponsorship of the U.S. Government. Neither the U.S. nor the U.S.D.O.E. nor the Leland Stanford Junior University nor their employees makes any warranty, express or implied, or assumes any liability or responsibility for accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use will not infringe privately-owned rights. Mention of any product, its manufacturer, or suppliers shall not, nor is it intended to, imply approval, disapproval, or fitness for any particular use. The U.S. and the University at all times retain the right to use and disseminate same for any purpose whatsoever.

For reprints of this paper, write:

Stanford Linear Accelerator Center
Publications Dept, Mail Stop 68
P.O. Box 4349
Stanford, CA 94309

Copyright (C) 1991 The Board of Trustees of The Leland Stanford Junior University. All Rights Reserved.

Permission is granted to SHARE Inc. to publish this presentation paper in the SHARE Proceedings

Contents

Section 1: Introduction 1.1

Section 2: Coding Paradigms 2.1

- Naming Pipes 2.1
- Portrait style 2.1
- FMTP XEDIT 2.3
- SC XEDIT 2.4
- Multi-stream pipes 2.4

Section 3: Testing Paradigms 3.1

- Console 3.1
- Disk 3.1
- Special debugging stages 3.2
- PIPEDEMO 3.2

Section 4: Pipeline Paradigms 4.1

- Locating alternatives - AND 4.1
- Locating alternatives - OR 4.1
- Locate ignoring case 4.2
- Search file for match 4.3
- Constructing host commands 4.3
- Read a CMS file into a REXX stem 4.4
- Dump REXX variables to a file for debugging 4.5
- Save and restore REXX variables 4.6
- Load CP SET values into REXX stem 4.7
- Restore CP settings from REXX stem 4.8
- Compute length of records 4.9
- Using LOOKUP 4.10
- Using DELAY for monitoring 4.10

Section 5: CallPipe Paradigms 5.1

- Two pipeline connections 5.1
- One pipeline connection 5.1
- No pipeline connections 5.2
- Multiple pipeline connections - multiple streams 5.2
- Pass stream through unchanged 5.3

Section 6: Filter Paradigms 6.1

NULL 6.1
Non-Delay 6.1
Summary 6.2
Circumventing the REXX 500-character limit 6.3
EXEC & Filter in same file 6.3
Using INTERPRET to apply a REXX function to each record 6.4
Read contents of a list of files into the pipeline 6.5

Section 7: A Simple Service VM 7.1

GETCMD REXX 7.1
DELAYCMD REXX 7.2
Main pipeline 7.3

Section 1

Introduction

The basic concepts of CMS Pipelines can be learned in a few hours; after that, attaining mastery is a process of learning increasingly sophisticated tools and techniques that can be applied to your problems. There are over 100 builtin drivers and filters, and large collections of stages written by users available on tools disks and bulletin boards.

To help the apprentice plumber learn the Pipelines techniques that can be applied to various classes of data manipulation problems, this guide catalogs common programming paradigms or idioms that can be copied into pipelines with little or no change.

These sections of prefabricated plumbing were collected from various sources. Where they are not either standard well-known constructs or taken from the author's own toolkit, the first contributor of the pipeline is acknowledged. Each pipeline is presented as it would appear in an exec with address command in effect, ready to be inserted into an exec for use. Contributions may have been altered to fit the style of the other examples. This is not to say that any coding style is better than any other; it is simply to lend consistency to the examples to make them easier to follow.

Section 2

Coding Paradigms

A. Naming Pipes

CMS Pipelines has a sometimes underused option called NAME. It may seem an unnecessary bit of icing to name all of your pipelines until receive an error message such as the first one from some exec that calls other execs as subroutines, and each contains dozens of pipelines. (This particular error message means that you named a stage that didn't exist. You may not have had a disk accessed that contained the needed stage.) If you name your pipelines, that name will appear in the error messages, and you will have an easier time locating the errant plumbing. To encourage this style, all examples in this guide are shown with names.

```
PIPSCB027E Entry point SWAP not found.
PIPSCA003I ... Issued from stage 3 of pipeline 1.
PIPSCA001I ... Running "swap".
Ready(-0027); T=0.01/0.01 16:24:23
```

```
PIPSCB027E Entry point SWAP not found.
PIPSCA004I ... Issued from stage 3 of pipeline 1 name "swap".
PIPSCA001I ... Running "swap".
Ready(-0027); T=0.01/0.01 16:24:50
```

B. Portrait style

Portrait style is a way of writing pipelines in REXX execs that makes them easier to understand. One stage of the pipe is written per line, with a comment explaining what that stage does. This is a good habit for you to get into early, even for pipes that you don't expect anyone else to be reading. Chances are that you'll end up reading your own pipes weeks or months later, and that you'll be grateful that you took the time to document their workings.

One further argument for the portrait style is that you will find that you tend to cut sections of pipelines out of one exec to paste into new execs that you write. If you write your pipelines in portrait style, you'll find it far easier to select just the stages you need.

Below is a sample of a portrait style pipeline. The final comma on each line but the last is the REXX continuation character. REXX will read those lines as a single command, removing the commas and joining the lines with a single blank between them. (I happen to like to line up the commas in one column to make it easy to visually spot when I have left one out. Missing commas can make your exec fail in sometimes non-obvious ways.)

Another matter of personal preference is whether the pipe separator character goes on the left of each line or the right. Some choose the left because the pipe separators line up nicely, and again it is harder to accidentally leave one out. However, there is a pitfall in this technique that makes some choose the right. The pitfall is that when REXX concatenates the strings on each line, it inserts a blank between them. An extra blank on the right side of a separator character never makes a difference, but an extra blank on the left side is potentially meaningful data to the preceding stage.

Consider the pipeline 'PIPE < paradigm script * | find :h| console' to display all the GML heading tags in this script file. The argument of FIND begins after exactly one blank and runs up to the next pipe separator character. If you write it as '| find :h |' it will find no tags, because ':h' is never followed by a blank in the source. The two ways around the problem are to write the separators on the right, or to use the REXX concatenate operator in cases were the blank would matter (primarily FIND, NFIND, TOLABEL, and related filters).

```
+--------------------------------------------------------------------+
|                                                                    |
| /* Left-sided portrait display sample.                        */   |
|                                                                    |
|   'PIPE (name LEFTY)'            ,                                 |
|     '  < paradigm script a'      , /* Read the input file.     */  |
|     '| find :h' ||               , /* Find headers.            */  |
|     '| console'                    /* Display on console.      */  |
|                                                                    |
+--------------------------------------------------------------------+
```

```
/* Right-sided portrait display sample.                        */

  'PIPE (name RIGHTY)'              ,
    '< paradigm script a |'         , /* Read the input file.      */
    'find :h|'                      , /* Find headers.             */
    'console'                         /* Display on console.       */
```

C. FMTP XEDIT

An XEDIT macro (which prefers the right-sided style) named FMTP is distributed with CMS Pipelines to help you format your pipes in portrait style. With a pipe positioned at the current line and one or more lines of stages, FMTP will reformat your pipe for you. You can also assign FMTP to a PF key and use the cursor to indicate which pipe to format.

```
=====
===== 'PIPE (name RIGHTY) < paradigm script a | find :h| console'
=====

====> fmtp
```

```
=====
===== 'PIPE (name RIGHTY)|',
=====    '< paradigm script a |',
=====    'find :h|',
=====    'console'
=====

====>
```

D. SC XEDIT

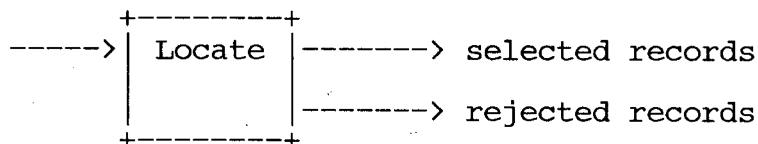
Unfortunately, FMTP won't write your comments for you, but once you add your comments another tool can make them look pretty. The SC XEDIT macro aligns REXX comments on a specified set of lines.

```
+---------------------------------------------------------------------------+
|                                                                           |
|  =====                                                                    |
|  ===== 'PIPE (name RIGHTY)|',                                             |
|  =====    '< paradigm script a |',  /* Read the input file. */           |
|  =====    'find :h|',  /* Find headers. */                                |
|  =====    'console'  /* Display on console. */                            |
|  =====                                                                    |
|                                                                           |
|  ====> sc 4                                                               |
|                                                                           |
+---------------------------------------------------------------------------+
```

```
+---------------------------------------------------------------------------+
|                                                                           |
|  =====                                                                    |
|  ===== 'PIPE (name RIGHTY)|',                                             |
|  =====    '< paradigm script a |',    /* Read the input file.      */    |
|  =====    'find :h|',                 /* Find headers.             */    |
|  =====    'console'                   /* Display on console.       */    |
|  =====                                                                    |
|                                                                           |
|  ====>                                                                    |
|                                                                           |
|                                                                           |
+---------------------------------------------------------------------------+
```

E. Multi-stream pipes

Multi-stream pipes are conceptually simple, but their notation can be tricky. In a multi-stream pipe, you can define alternate streams with different processing for selected records. One simple example is the LOCATE filter: it passes selected records along the main pipeline, and in addition can pass the rejected records to a secondary pipeline for further processing.



What makes the notation difficult is representing this two-dimensional idea in a one-dimensional pipeline. First we need an END character, which delimits the alternate linear sections of pipeline, and then we need labels within those sections, to show where they connect.

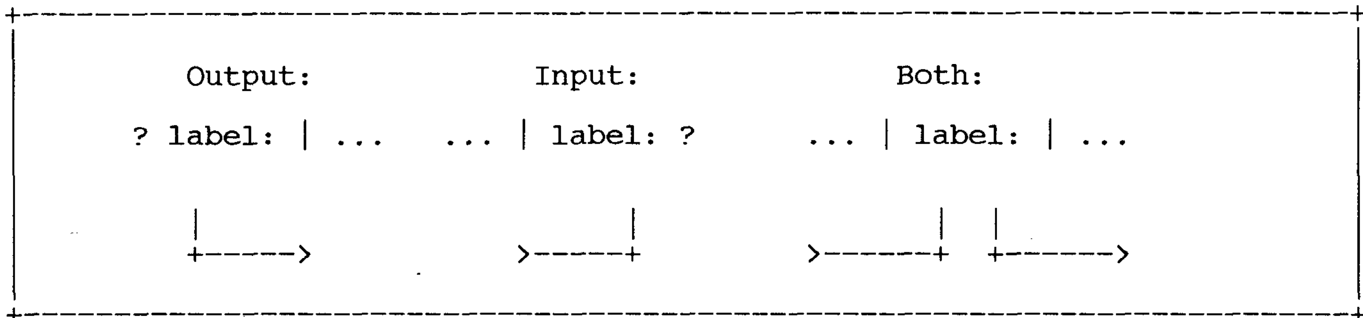
```
'PIPE (name MULTI end ?)' ,           /* Define END character   */
  ...                                 /* Some record source.    */
  '| tee: locate /something/'   ,     /* Produce two streams.   */
  '| > selected records a'      ,     /* Process first stream.  */
  '? tee:'                      ,     /* Begin second stream.   */
  '| > rejected records a'            /* Process second stream. */
```

The end character separates the linear sections of the pipeline from each other. These sections are independent pipelines and have nothing to do with each other until you define connections between them with labels. (If you didn't define any connections between them, they would happily run as two self-contained parallel pipelines.)

Labels are one to eight character strings, followed by a colon. There are two ways for a label to appear: 1) as a prefix to a stage name (as on the LOCATE stage above); this is the DEFINITION of the label; or 2) as a stage by itself (as in the second appearance of 'tee' above); this is a REFERENCE of the label. Two simple rules about specifying labels: the first appearance of a label must be a definition, and there may be any number of following references.

The definition of a label only tells you where connections to other pipelines can be made; it doesn't tell you how many there are or whether they are for input or output to the stage. The number of connections is determined by the number of references, and the type of connection is determined by the placement of the references. Further, you have to check the description of the stage in the reference manual or help file to find out what kind of connections are used by that stage; it is certainly possible to define connections that will never be used!

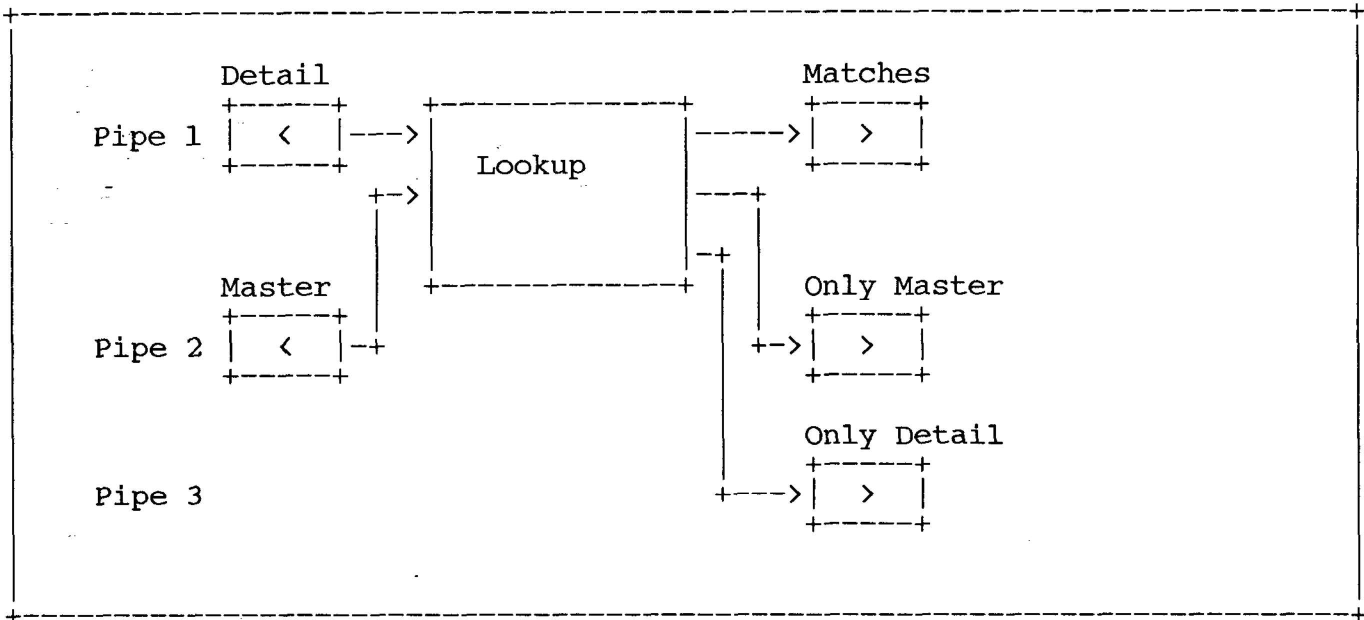
All references to a label are relative to the original definition. If a reference has more stages to its right, it defines an output from the original stage (and input to the stages to its right). If a reference has more stages to its left, it defines an input to the original stage (and output from the stages to its left). If a reference has stages on both sides, it defines both an input and an output.



The streams intersecting at a labeled stage are numbered. The primary stream (flowing through the pipe connectors from left to right), is numbered zero. The alternate input and output streams are each numbered starting at one.

LOOKUP is a good example to study to understand multi-stream pipes. LOOKUP reads two input files, and writes matched records to the primary stream and unmatched records to two additional streams depending on which input file they came from. So in addition to the primary input and output it needs a second input defined and two additional outputs defined.

Start by writing the LOOKUP stage and giving it a label. This defines where the potential connections may be made. Then in a second pipeline (after an end character), write a reference to that label to the right of the source of the second file. To the left of that label, write the destination of the first set of unmatched records. You have just defined input stream 1 and output stream 1 to the lookup stage. Finally, in a third pipeline, begin with another reference to the label, followed by the destination of the second set of unmatched records. You have just defined output stream 2. That's all there is to it.




```
'PIPE (name LOOKDEMO end ?)',
   '  < detail records *'   , /* Read Detail file.          */
   '| lkup: lookup detail'  , /* Match the files.           */
   '| > matches file a'     , /* Write matches.             */
   '?'                      , /* End of first pipe.         */
   '  < master records *'   , /* Read Master file.          */
   '| lkup:'                , /* Route through lookup.      */
   '| > master only a       , /* Lookup's secondary output*/
   '?'                      , /* End of second pipe.        */
   '  lkup:'                ,
   '| > detail only a'        /* Lookup's tertiary output.*/
```

Section 3

Testing Paradigms

A. Console

When you are first learning CMS Pipelines, or when you write a pipeline that doesn't do what you expect, you can use CONSOLE stages at strategic points in your pipeline to display the records that flow past that point in the pipe. The key here is that CONSOLE both displays records on the console, and passes them along unaltered to the following stage, if any.

```
/* Demonstrate use of CONSOLE stage for debugging.          */

  'PIPE (name CONSDEMO)',
    ...                    /* Some source for records.      */
    '| console'          , /* Display unaltered records.    */
    '| split'            , /* Split into words.             */
    '| strip'            , /* Remove extra blanks.          */
    '| console'          , /* Display changes to this point.*/
    ...                    /* Continue processing.          */
```

B. Disk

The DISK stage can be used similarly to CONSOLE to collect the records passing each point in the pipeline. DISK has the advantage that it can collect larger quantities of data, and can segregate the output by directing it to different files. On the other hand, it has the disadvantage that the relative timing of the different outputs is not apparent.

```
/* Demonstrate use of DISK stage for debugging.              */

  'PIPE (name DISKDEMO)',
    ...                       /* Some source for records.     */
    '| disk set1 recs a',     /* Display unaltered records.   */
    '| split'           ,     /* Split into words.            */
    '| strip'           ,     /* Remove extra blanks.         */
    '| disk set2 recs a',     /* Display changes to this point.*/
    ...                       /* Continue processing.         */
```

C. Special debugging stages

A handy REXX filter to keep around is one like the following TATTLE: it reports on the records flowing through it, like CONSOLE does, but prefixes the data with the stage number, to allow you to identify from which portion of the pipeline the output came.

```
/* TATTLE REXX:  Report on records passing through.    */

  'stagenum'
  stagenum = rc

  Signal On Error

  Do Forever                             /* Do until EOF     */
     'readto record'                     /* Read from pipe   */
     say 'Stage' stagenum':' record      /* Report record.   */
     'output' record                     /* Write to pipe    */
  End

Error: Exit RC*(RC<>12)                  /* RC = 0 if EOF    */
```

D. PIPEDEMO

One of the most powerful tools for debugging or demonstrating Pipelines is the PIPEDEMO EXEC, written by the author of this paper (so you know that this is an unbiased evaluation!) PIPEDEMO displays a full-screen animation of the data flowing through your pipeline. Your stages are shown in portrait format (one stage per line) on the left side of the

display, and the data that was most recently written by that stage appears on the right side. The cursor moves to show the currently executing stage, and the data display is updated every time a new record is written.

PIPEDEMO can show you how data really flows through your pipeline, and will demonstrate the order in which the various stages perform their tasks. It can illustrate multiple-stream pipelines, asynchronous pipelines, and most CALLPIPE subroutines.

PIPEDEMO is not part of CMS Pipelines. It is available on the VMSHARE electronic conference as NOTE PIPEDEMO. It is also available on the LISTSERV at AWIIMC12. Send the following line to LISTSERV at AWIIMC12 via the TELL command, or as the only line in a mail item:

```
GET PIPEDEMO PACKAGE
```

The file will come as a KNAPSACK archive. If you do not have the DEKNAP EXEC, also send the LISTSERV the command:

```
GET CMSPIP-L PACKAGE
```

If neither of those distribution methods is available to you, PIPEDEMO can also be found on the 1991 VMWorkshop Tools Tape, which is distributed by the University of Waterloo for a nominal duplication fee. For information, contact

Jack Hughes, MC2053 University of Waterloo Department of Computing Services Waterloo, Ontario Canada N2L 3G1 (519) 888-4621

```
+----------------------------------------------------------------------+
|                                                                      |
|   cp QUERY NAME                    -> SLDTEST  -  DSC , JBTAP    -   |
|   | split after ,                  -> SLDTEST  -  DSC ,              |
|   | strip leading                  -> SLDTEST  -  DSC ,              |
|   | a: nlocate / DSC/              -> TINEKE   -  122E,              |
|   | spec 1.8 1                     -> TINEKE                         |
|   | sort                           ->                                |
|   | > conn users a                 ->                                |
|   | count lines                    ->                                |
|   | spec 1-* 1 / conn. users./ next->                                |
|   | pause                          ->                                |
|  \a:                               ->_TVM      -  DSC                |
|   | spec 1.8 1                     -> TVM                            |
|   | sort                           ->                                |
|   | > disconn users a              ->                                |
|   | count lines                    ->                                |
|   | spec 1-* 1 / dsc. users./ next ->                                |
|   | pause                          ->                                |
|                                                                      |
|                                                                      |
|  ENTER=Resume    PF1=Help  PF2=Step PF3=Quit  PF4=Delay              |
|                                                                      |
+----------------------------------------------------------------------+
```

Section 4

Pipeline Paradigms

A. Locating alternatives - AND

The LOCATE filter is useful for selecting records when the string is one contiguous sequence of characters. But when you need to select records that have certain strings in more than one field in the record, you must use more than one LOCATE filter. Each LOCATE discards the records that do NOT contain the specified string, so any record that passes through all of the LOCATE filters must contain ALL of the specified strings.

```
/* Fragment to locate records that contain ALL of several   */
/* alternatives.                                            */

 'PIPE (name ANDDEMO)' ,
   ...                     /* Some source for records.      */
   '| locate /first/'  ,   /* Select records that have      */
   '| locate /second/' ,   /*   all of the required strings,*/
   '| locate /third/'  ,   /*   discard all others.         */
   ...                     /* Continue processing.          */
```

B. Locating alternatives - OR

Just as frequently, one must select records that have any one of several strings. The solution for this is to re-examine the stream of rejected records, which LOCATE directs to its secondary output, for additional alternatives.

```
/* Fragment to locate records that contain ANY of several    */
/* alternatives.                                             */

'PIPE (name ORDEMO end ?)' ,
   ...
   '| a: locate /first/'  ,  /* Locate first alternative    */
   '| z: faninany'        ,  /* Merge all alternatives      */
   ...
   '? a: '                ,  /* Non-matches go here.        */
   '| b: locate /second/' ,  /* Locate second alternative   */
   '| z: '                ,  /* Send to primary stream.     */
   '? b: '                ,  /* Non-matches go here.        */
   '| c: locate /third/'  ,  /* Locate third alternative    */
   '| z: '                   /* Send to primary stream.     */
```

C. Locate ignoring case

The LOCATE filter performs an exact match -- letters must be the same case as the target to match. If you wish to ignore case, you can simply translate all records and the search string to upper case. However, if you also want to preserve the original mixed-case record, the following little trick duplicates the data in the records, translates one of the copies to upper case, then discards that copy after the comparison. This example handles up to 256 bytes of data, but can handle more if necessary. The portion copied can be restricted to just the columns that contain the target, if that is less than the entire record.

```
/* Fragment to locate records ignoring case.                 */

  s = translate(searchstring)

  'PIPE (name IGNCASE)'      ,
    ...
    '| specs 1.256 1 1-* 257', /* Make a duplicate of the   */
    '| xlate 1.256 upper'    , /*   data columns, make it   */
    '| locate 1.256 /'s'/'   , /*   upper case, match on    */
    '| specs 257-* 1'        , /*   the copy, then delete.  */
    ...
```

D. Search file for match

A frequent task is to validate an input against a list -- for instance checking a userid against a list of privileged users. This simple pipeline sets a REXX variable to boolean true or false if the search value is found in the list.

```
/* Find input value in list.                                    */

  'PIPE (name VALIDATE)' ,
    '  < user list *'      , /* Or load the list from a stem. */
    '| strip'              , /* Remove leading blanks.        */
    '| xlate upper'        , /* Upper case for match.         */
    '| find' value         , /* Search list for matches.      */
    '| take 1'             , /* Take only first match.        */
    '| var match'          , /* Optionally save matched line. */
    '| count lines'        , /* 1=found, 0=not found.         */
    '| var valid'            /* Set boolean variable.         */
```

E. Constructing host commands

A most useful technique is constructing host commands from records in the pipeline and issuing them via the CP and CMS device drivers. Each of these drivers can take a single command as an argument (e.g., cms listfile * * a), however their real power lies in their ability to read their commands from the input stream. The following exec queries the users linked to a specified virtual disk, and constructs messages telling them to reaccess their disk.

```
/* REACCMSG EXEC                                            */
/* Notify users to re-ACCESS a changed disk                 */

   Parse Arg vaddr message

   'PIPE (name REACCMSG)'     ,
     '  cp q links' vaddr     ,    /* Issue CP command         */
     '| split at ,'           ,    /* Get one user per line    */
     '| strip'                ,    /* Remove leading blanks    */
     '| sort unique 1-8'      ,    /* Discard duplicates       */
     '| nfind' userid()       ,    /* Don't msg ourselves.     */
     '| spec /MSG/ 1'         ,    /* Make MSG commands        */
       'w 1 nextword'         ,    /* Fill in userid           */
       '/Please re-ACCESS your/ nextw',
       'w 2   nextword'       ,    /* Fill in virtual address */
       '/disk./ nextword '    ,
       '/'message'/ nextword',
     '| cp'                        /* Issue MSG commands       */
```

F. Read a CMS file into a REXX stem

This is one of the most basic ways to use CMS Pipelines to replace EXECIO. However, Pipelines can go far beyond EXECIO: you can insert filters to remove comment lines or columns, select only the records or columns that the exec needs, and otherwise reduce the amount of work the exec needs to do (and reduce the amount of REXX code you have to write!).

One word of warning: if you replace EXECIO in an exec that is using it to read or write one record at a time in a loop, and simply replace that with an identical pipeline to read or write one record, you will find that the exec is much less efficient. This is because EXECIO does not close the file (unless you tell it to), while Pipelines does. However, if you rethink the exec to do all the processing of the loop inside the pipeline, you will probably gain efficiency.

```
+--------------------------------------------------------------------------+
|                                                                          |
|  /* Fragment to load a file into a REXX stem variable.         */        |
|                                                                          |
|    'PIPE (name LOADFILE)' ,                                              |
|      '  <' fn ft fm        ,     /* Read disk file.            */        |
|      '| nfind *'           ,     /* Optionally remove comments.*/        |
|      '| stem record.'            /* Put into RECORD.           */        |
|                                                                          |
|    /* RECORD.0 now contains the number of records read.       */        |
|    /* RECORD.n (0 < n <= RECORD.0) now contains record n.     */        |
|                                                                          |
|                                                                          |
+--------------------------------------------------------------------------+
```

G. Dump REXX variables to a file for debugging

The REXXVARS device driver can find all active REXX variables at the time that it is called. This makes it very useful for error routines in your execs. You can dump all variables to the console or a file, to make debugging easier. The following is an ON ERROR routine for an exec to dump all variables to a file.

```
Error:    /* Unexpected nonzero return code.                      */

  /* Save the return code                                       */

  rt = rc

  /* Clear any odd ADDRESS and empty the stack.                  */

  address command
  'DROPBUF 0'

  /* Find out our name.                                          */

  parse source . . $fn $ft $fm .

  $dft = left($ft,4,'X')'VARS'

  /* Report the error.                                           */

  say right(sigl,6) '+++' sourceline(sigl)
  say 'Error in line' sigl 'of' $fn $ft $fm 'rc =' rt

  /* Dump the variables to a file with our name.                 */

  'PIPE (name DUMPVARS)'     ,
     '  rexxvars'            ,  /* Get all REXX variables.      */
     '| drop 1'              ,  /* Get rid of header line.      */
     '| spec 3-* 1 / / next',   /* Get rid of id chars.         */
     '| join 1 /= /'         ,  /* Join var names & values.     */
     '| sort'                ,  /* Sort by variable name.       */
     '| >' $fn $dft 'a'         /* Write to file.               */

  say 'Variables have been dumped to' $fn $dft 'A'

  exit rt
```

H. Save and restore REXX variables

Sometimes an exec must save its current state to load another time. This may arise for an exec that must invoke XEDIT for processing, and so must stack a call to itself and call XEDIT. Or a service virtual machine may need to save its state across restarts. These two fragments illustrate saving all REXX variables to a file and reloading them.

```
/* Save REXX variables in CMS file.                              */

  'PIPE (name SAVEVARS)'    ,
     '  rexxvars'           ,  /* Get all REXX variables.        */
     '| drop 1'             ,  /* Get rid of header line.        */
     '| spec /=/ 1 3-* next',  /* Insert delimiters.             */
     '| join 1'             ,  /* Join var names & values.       */
     '| > rexx vars a'         /* Write to file.                 */

/* Restore REXX variables from CMS file.                         */

  'PIPE (name RESTVARS)'    ,
     '  < test vars a'      ,  /* Read variables from file.      */
     '| varload'               /* Create rexx variables.         */
```

I. Load CP SET values into REXX stem

This example illustrates several nice properties of VARLOAD. This exec returns the values of the individual settings reported by the CP QUERY SET command. The SPLIT stage divides up the CP response into individual records for each setting, each of which will have one word for the setting name, and one or more words for the value. The SPEC stage constructs variable names for these settings, using the supplied stem and the setting name. (So CPSET.MSG may contain values such as ON, OFF, or IUCV.) VARLOAD then creates the variables, but the argument '1' tells it to create the variables not in this exec, but in the exec that called this one. This technique allows you to write procedures that pass back information this way, much the way the XEDIT EXTRACT command does.

```
/* CPQSET EXEC: Load CP SET values into REXX stem.          */
/* Jim Colten, University of Minnesota                      */

   Address Command

   Arg stem .

   If stem = '' then stem = 'CPVAR.'

   'PIPE (name CPQSET)'   ,
     '  cp QUERY SET'     ,       /* Get Q SET output          */
     '| split ,'          ,       /* Split into settings.      */
     '| spec /='stem'/ 1',        /* Build up stem name,       */
     '   word 1 next'     ,       /*   delimiters, and         */
     '   /=/ next'        ,       /*   value for VARLOAD.      */
     '   word 2-* nextw'  ,
     '| varload 1'                /* Create vars for caller.   */
```

J. Restore CP settings from REXX stem

This example is the inverse of the previous one. It gathers a set of variables from the calling exec, changes them into CP commands, and issues them. The STEM stage is useful for retrieving a REXX stem if the subscripts are all numeric and sequential, and if the stem.0 element has been set to the number of valid stems. This illustrates a technique for selecting an arbitrary set of variables from the environment: the REXXVARS stage will locate and output every variable in the environment; selection stages can then be applied to filter out only those variables of interest.

```
/* CPRESET EXEC: Restore CP variables from REXX stem.          */
/* Jim Colten, University of Minnesota                          */

  Address Command

  Arg stem .

  If stem = '' then stem = 'CPVAR.'

  sl = length(stem)

  'PIPE (name CPRESET)'            ,
     '   rexxvars 1'               , /* ALL variable info      */
     '|  drop 1'                   , /* Drop source line.      */
     '|  spec 3-* 1'               , /* Join name & value,     */
     '     read 3-* nextword'      , /*   removing type col.   */
     '|  find' stem           ||   , /* Only our stem.         */
     '|  nfind' stem'0'            , /* Discard a counter      */
     '|  nlocate /ECMODE/'         , /* SET ECMODE is BAD!     */
     '|  spec /CP SET / 1'         , /* Make CP command,       */
     '     sl+1'-* next'           , /*   removing stem name.  */
     '|  cp'                       , /* Let CP do reSET.       */
     '|  console'                    /* Show any CP output.    */
```

K. Compute length of records

This pipeline fragment will put all records in one REXX stem variable, and put the length of each record in the corresponding position in another stem variable. If you insert a SPLIT filter earlier in the pipeline, you can obtain a list of the words in the input with their lengths.

```
/* SIZE EXEC: Fragment to compute record lengths.              */

   'PIPE (name SIZE)'          ,
    ...
    '| stem record.'           , /* Put all records in stem. */
    '| specs 1-* v2c 1.2'      , /* Make variable-length char*/
    '| specs 1.2 c2d 1 left'   , /*   strings, keep only len.*/
    '| stem size.'               /* Put lengths in stem.     */
```

L. Using LOOKUP

LOOKUP is a very handy stage for matching files. It reads two input files and compares them, and writes out the records that match and the records that don't match. The trick to understanding the following example is realizing that the second occurrence of the label 'lkup:' defines both an input stream and an output stream.

```
'PIPE (name LOOKDEMO end ?)',
  '  < detail records *'   , /* Read Detail file.          */
  '| lkup: lookup detail'  , /* Match the files.           */
  '| > matches file a'     , /* Write matches.             */
  '?'                      , /* End of first pipe.         */
  '  < master records *'   , /* Read Master file.          */
  '| lkup:'                , /* Route through lookup.      */
  '| > master only a       , /* Lookup's secondary output*/
  '?'                      , /* End of second pipe.        */
  '  lkup:'                ,
  '| > detail only a'        /* Lookup's tertiary output.*/
```

M. Using DELAY for monitoring

The DELAY stage can be used to do a task at intervals. This exec issues a system monitoring command every five minutes and logs the response to a disk file.

```
/* MONITOR EXEC:  Gather statistics                          */
/* Melinda Varian, Princeton                                 */

 'PIPE (name MONITOR)'            ,
   '  literal +5:00'              ,  /* Interval to wait.    */
   '| dup *'                      ,  /* Repeat wait.         */
   '| delay'                      ,  /* Do wait.             */
   '| spec /QUERY TIME/ 1 write'  ,  /* Change output into   */
         '/QUERY USERS/ 1'        ,  /*   two CP commands.   */
   '| cp'                         ,  /* Issue commands.      */
   '| diskslow users file a'         /* Record & close file.*/
```

Section 5

CallPipe Paradigms

A. Two pipeline connections

Many typical subroutine pipelines are a packaging of a sequence of stages, which replace the calling stage with that sequence. As such, they have a pipeline connector at each end, one to accept records from the pipeline, and one to direct records back. This simple example packages the two steps of creating fixed-length records.

```
/* FIXED REXX: Make fixed length records.                   */

 arg length .

 'callpipe (name FIXED)',     /* Add subroutine pipeline.  */
   '  *:'               ,     /* Get records from main pipe*/
   '| chop' length      ,     /* Chop off the long ones.   */
   '| pad'  length      ,     /* Pad out the short ones.   */
   '| *:'                     /* Put back into main pipe.  */
```

B. One pipeline connection

Some subroutine pipelines only insert data into a pipeline, or only take data from the pipeline; in these cases the pipeline has only a single connector to the main pipeline. You may need this if the stage is intended to only be a sink or source for records, which would only appear as the first or last stage of a pipeline. Alternately, it may be part of a REXX filter that reads or writes the pipeline with READTO and OUTPUT, and uses the subroutine pipeline for the other connection (see section 'Read contents of a list of files into the pipeline' for an example).

```
/* GETNAMES REXX: Search NAMES file for matches.        */

 parse arg criterion

 'callpipe (name GETNAMES)',
   '  command NAMEFIND' criterion ':USERID :NAME (TYPE *' ,
   '| spec 1-* 1.8'          ,   /* Combine userid and     */
   '     read 1-* 10'         ,   /*   name in 1 record.    */
   '| *:'                         /* Write to pipe          */
```

C. No pipeline connections

A pipeline segment created with CALLPIPE need not have any intersections with the main pipeline. This can be useful to simply take advantage of pipeline builtin stages to process data within the REXX stage.

```
/* How many reader files do we have?                    */

 'callpipe (name QFILES)',
   '  cp QUERY FILES'    ,  /* Query number of files.   */
   '| spec word 2 1'     ,  /* Take only number.        */
   '| change /NO/0/'     ,  /* Make always numeric.     */
   '| var nfiles'           /* Put in REXX variable.    */
```

D. Multiple pipeline connections - multiple streams

A subroutine pipeline can define multiple input and output streams, which may be connected to the calling pipeline through the use of labels on the calling stage. The stream number is supplied on the connector, with zero being the number of the primary stream, one the number of the secondary, and so on. The following example writes the list of currently connected users to the primary stream, and the list of disconnected users to the secondary stream. Following this is a sample of how you might call it, sorting each output stream separately and labeling them. However, you are not required to connect the secondary output stream; if it is not connected, it is simply not used and the records are discarded.

```
/* QNAMES REXX: write connected users to primary stream,  */
/* disconnected users to secondary stream.                */

  'callpipe (name QNAMES end ?)'  ,
   'cp 80000 QUERY NAMES'  ,     /* Query all users.       */
    '| split at ,'         ,     /* One user per line.     */
    '| strip'              ,     /* No extra blanks.       */
    '| nlocate /LOGN/'     ,     /* Get rid of LOGNxxx.    */
    '| dsc: nlocate / DSC/' ,    /* Keep connected users.  */
    '| spec 1-8 1'         ,     /* Keep only the userid.  */
    '| *..0:'              ,     /* Send to primary stream*/
    '? dsc:'               ,     /* Disconn users here.    */
    '| spec 1-8 1'         ,     /* Keep only the userid.  */
    '| *..1:'                    /* Send to secondary.     */


/* LISTUSER EXEC: List connected & disconnected users.    */

  'PIPE (name LISTUSER end ?)',
    '  x: qnames'            ,   /* Generate streams.      */
    '| sort'                 ,   /* Sort first stream.     */
    '| join 7 / /'           ,   /* Put 8 per line.        */
    '| literal Connected:'   ,   /* Label the display.     */
    '| console'              ,   /* Put on console.        */
    '? x:'                   ,   /* Second stream here.    */
    '| sort'                 ,   /* Sort second stream.    */
    '| join 7 / /'           ,   /* Put 8 per line.        */
    '| literal Disconnected:' ,  /* Label the display.     */
    '| console'                  /* Put on console.        */
```

E. Pass stream through unchanged

Sometimes you want to want to pass records through a subroutine unchanged, but your processing must make temporary changes. Rather than attempting to preserve and restore the records, this technique passes through the main stream unchanged, and duplicates records to a secondary stream that can be altered. This example would be used in an XEDIT macro to insert records in the file after the current line.

```
/* XI REXX:   Xedit insert                                    */
/* John Hartmann 30 Jan 1989 18:35:31                         */

  signal on novalue

  /* If XEDIT's linend is turned on, translate that           */
  /* character to blanks, because it can't be input.          */

  address xedit 'extract /linend/'
  If linend.1='ON'
     then xlate='|xlate *-*' c2x(linend.2) 'space'
     else xlate=''

  'callpipe (name XI end ?)' ,
  '*:'               ,  /* Accept input from calling pipe.    */
  '|f: fanout'       ,  /* Duplicate to secondary stream.     */
  '|*:'              ,  /* Pass records back to pipeline.     */
  '?f:'              ,
     xlate           ,  /* Translate if necessary.            */
  '| change //i /'   ,  /* Convert to Insert commands.        */
  '| subcom xedit'      /* Pass to XEDIT.                     */
```

Section 6

Filter Paradigms

A. NULL

This is the basic 'do nothing' pipeline filter, which everyone should keep around as a starting point for writing any new filters. The expression on the last line is itself an idiom, which returns zero if RC is equal to either zero or twelve, and otherwise returns the value of RC. Use 'signal on error' to end the loop if you don't issue many host commands that may have valid non-zero return codes, otherwise be sure you test the return code from both READTO and OUTPUT to leave the loop.

```
/* NULL REXX:                    Dummy pipeline filter   */

  Signal On Error

  Do Forever                        /* Do until EOF      */
     'readto record'                /* Read from pipe    */
     'output' record                /* Write to pipe     */
  End

Error: Exit RC*(RC<>12)             /* RC = 0 if EOF     */
```

B. Non-Delay

```
/* NONDELAY REXX:                   Dummy pipeline filter  */

  Signal On Error

  'peekto record'                     /* Prime the loop.   */
  Do Forever                          /* Do until EOF      */
     'output' record                  /* Write to pipe     */
     'readto record'                  /* Release record.   */
     'peekto record'                  /* Get next record.  */
  End

Error: Exit RC*(RC<>12)               /* RC = 0 if EOF     */
```

C. Summary

The final basic type of filter reads records continuously, but does not write any records until end-of-file is reached on the input, and then writes a single record summarizing the input records. The builtin filter COUNT works in this fashion.

```
/* SUMMARY REXX:                    Dummy pipeline filter  */

  count = 0

  Do Forever                          /* Do until EOF      */
     'readto record'                  /* Read from pipe    */
     if rc <> 0 then leave            /* Leave on EOF.     */
     /* Summarize the records here.                        */
     count = count + 1                /* For example.      */
  End

  'output' count                      /* Write summary.    */

Error: Exit RC*(RC<>12)               /* RC = 0 if EOF     */
```

D. Circumventing the REXX 500-character limit

REXX has a limit of 500 characters per clause, which can prevent you from entering a complex pipeline as a single string. If you store portions of the pipeline as variables, you can circumvent that limit.

```
part1 = 'stage1' ,
        '| stage2' ,
        '| stage3'

part2 = '| stage4' ,
        '| stage5' ,
        '| stage6'

'PIPE' part1 part2
```

E. EXEC & Filter in same file

Sometimes to keep the number of files down, one wishes to package both a pipeline and a filter called by the pipeline in the same exec. There are two techniques for doing this, both of which follow. The first uses the REXX stage to call a file with filetype of EXEC as a filter, the second EXECLOADs the exec with a filetype of REXX before calling it.

```
/* First demonstration of pipeline and filter in a single file */

  parse source . . $fn $ft . . how .

  If how='?' then do
     'output Hello world! (From a dual-path REXX.)'
  end
  else do
    'PIPE (name HELLO)'          ,
      '  rexx (' $fn $ft ')'     ,
      '| console'
  end

  exit RC

/* Second demonstration of pipeline and filter in a single file */

  parse source . . $fn $ft . . how .

  If $ft = 'REXX' then do
     'output Hello world! (From a dual-path REXX.)'
  end
  else do
    'EXECLOAD' $fn $ft '*' $fn 'REXX'

    'PIPE (name HELLO)'          ,
         $fn                     ,
      '| console'

     'EXECDROP' $fn 'REXX'
  end

  exit RC
```

F. Using INTERPRET to apply a REXX function to each record

This useful stage can apply any REXX function to each record passing through it. For instance, pipe ... | applyfn reverse(record) | ...

```
/* APPLYFN REXX:  Apply a REXX function to each record      */
/* John Lynn, Mobil                                         */

  Arg function
  Signal On Error

  /* The entire loop following is interpreted with the      */
  /* function passed as an argument.                        */

  Interpret "
    Do forever;
       'readto record';
       'output' "function";
    End"

  Error:  Return RC*(RC<>12)
```

G. Read contents of a list of files into the pipeline

This stage reads a list of file names from its input, and writes the contents of those files to its output. For instance, one way (though not the best way) to find out many lines of execs you have written:

```
pipe cms listfile * exec a | readfile | count lines | console
```

This example also illustrates that you can combine READTO and CALLPIPE in a single stage, that CALLPIPE can be called multiple times from one stage, that you can substitute another source of records for those flowing into the stage.

READFILE is built in to later releases of Pipelines as the GETFILES stage.

```
+--------------------------------------------------------------------------+
|                                                                          |
|  /* READFILE REXX:     Send contents of files into the pipe        */    |
|  /*                    after processing with the filter            */    |
|  /*                    passed as an argument                       */    |
|                                                                          |
|  /* Input: filenames;  Output: (modified) contents of the files    */    |
|                                                                          |
|  Signal On Error                                                         |
|                                                                          |
|    Do Forever                         /* Do until get EOF          */    |
|       'readto record'                /* Get next input record     */    |
|       Parse Var record fn ft fm .    /* Break out file name       */    |
|                                                                          |
|       'callpipe'   ,                 /* Invoke pipeline           */    |
|         '  <' fn ft fm ,             /* Put file into stream.     */    |
|         '| *:'                       /* Connect into main pipe    */    |
|    End                                                                   |
|                                                                          |
|  Error: Exit RC*(RC<>12)              /* RC = 0 if EOF             */    |
|                                                                          |
+--------------------------------------------------------------------------+
```

Section 7

A Simple Service VM

This description of the implementation of a service machine using Pipes was written by Melinda Varian of Princeton University:

Several people have asked me to post a small service machine implemented in Pipes, to show a way of using 'delay', 'starmsg', 'immcmd', and PIPMOD STOP.

A. GETCMD REXX

The guts of the server are hidden in a single subroutine called GETCMD, which receives all classes of input, combines it with the FANINANY stage and routes it to the main pipeline.

Commands typed on the virtual machine console are trapped by the 'immcmd' stages. If a SHUTDOWN command is typed, then a PIPMOD STOP command is given to CMS to stop the pipeline. If a CP command is typed, then it is given to CMS to pass to CP. If a command intended for the server (prefaced by 'CMD') is typed, then it is sent through the 'faninany' stage and the connector into the calling pipeline.

Commands SMSGed from other virtual machines are received by the 'starmsg' stage and are also fanned in and sent through the connector to the calling pipeline.

Messages generated by the arrival of spool files are trapped by the same 'starmsg' (all kinds of messages have been set to IUCV), so they also go to the calling pipeline.

A 'literal' stage fires once when the pipeline is first invoked. This sends a 'primer' line into the calling pipeline to get it to start processing.

Timer interrupts are generated by a subroutine called DELAYCMD, which feeds a line into GETCMD each time it has done something for which the main (calling) pipeline should be wakened. (GETCMD fans those lines in and sends them through the connector to the caller, just as it does with everything else.)

```
/*  GETCMD REXX                                                    */

Signal On Syntax

primer = '00000004*       RDR FILE **** FROM'

'callpipe (end ?)'                     ,
   '| immcmd CMD'                      ,  /* Immediate commands   */
   '| spec /00000004*/ 1.16 1-* next', ,  /* As if SMSG from self */
   '| f: faninany'                     ,  /* Join all commands    */
   '| *:',                             ,  /* Pass to caller       */
'?'                                    ,
   '  starmsg'                         ,  /* Listen for SMSGs     */
   '| f:'                              ,  /* Pass to caller       */
'?'                                    ,
   '  literal' primer 'startup'        ,  /* Prime the pipeline   */
   '| f:'                              ,  /* Pass to caller       */
'?'                                    ,
   '  delaycmd'                        ,  /* Repetitive cmds.     */
   '| f:'                              ,  /* Pass to caller       */
'?'                                    ,
   '  immcmd CP'                       ,  /* Immediate CP command */
   '| spec /CP/ 1 1-* 4'               ,  /* Build command        */
   '| subcom cms'                      ,  /* Pass to CMS          */
'?'                                    ,
   '  immcmd SHUTDOWN'                 ,  /* SHUTDOWN command     */
   '| spec /PIPMOD STOP/'              ,  /* Build PIPMOD STOP    */
   '| subcom cms'                         /* Pass to CMS          */
```

B. DELAYCMD REXX

Here's a portion of DELAYCMD:

```
'callpipe (end ?)'                              ,
   '  literal +24:00:00'                        , /* Once per day             */
   '| dup *'                                    , /* Repeat interval.         */
   '| literal 23:00:00'                         , /* At 11pm daily            */
   '| delay'                                    , /* Do the wait.             */
   '| spec /CHANGE RDR CLASS K NOHOLD/',
   '| cp'                                       , /* Release Class K files    */
   '| nlocate /NO/'                             , /* Forget it if no files    */
   '| spec /'primer'/ 1 '                       ,
   '    / class k delay/ next'                  , /* Fake file arrival        */
   '| f: faninany'                              , /* Join all messages        */
   '| *:',                                      , /* Pass to GETCMD           */
'?'                                             ,
   '  literal +240:00'                          , /* Four-hour interval       */
   '| dup *'                                    ,
   '| delay'                                    ,
   '| spec /CHANGE RDR CLASS J NOHOLD/',
   '| cp'                                       , /* Release Class J files    */
   '| nlocate /NO/'                             , /* Forget it if no files    */
   '| spec /'primer'/ 1 '                       ,
   '     / class j delay/ next'                 , /* Fake file arrival        */
   '| f:'                                       , /* Pass to GETCMD           */
'?'                                             ,
   . . .
```

C. Main pipeline

Thus, the main pipeline starts out by invoking GETCMD; all its input comes from GETCMD, arriving in whatever order the various events (reader files, console commands, timer-driven events, SMSGed commands) occur and whenever they occur. The rest of the pipeline is simply an extensive decoding network that looks at the single record received from GETCMD for each event and decides what action to take in each case.

```
'pipe (end ?)',
  'getcmd |',
  . . .
```