PLUNGING INTO *PIPES*

Melinda Varian

Office of Computing and Information Technology
Princeton University
87 Prospect Avenue
Princeton, NJ 08544 USA
—.—
BITNET: MAINT@PUCC
Internet: maint@pucc.princeton.edu
Telephone: (609) 258-6016

REXX Symposium
May 5, 1992

I. INTRODUCTION

CMS Pipelines[1] is the most significant enhancement to CMS since REXX. It introduces into CMS the powerful data flow model of programming that was popularized by UNIX[2] pipes. UNIX pipes were built to work with a byte-oriented file system, but *CMS Pipelines* has so successfully met the challenge of making the pipeline concept work well with a record-oriented file system that *CMS Pipelines* is now being used in MVS, GCS, and MUSIC, as well as in CMS.

There are two primary reasons for discussing *CMS Pipelines* at a REXX Symposium. First, REXX and *CMS Pipelines* work so well together that the example of their synergy may inspire advances in other REXX environments. When *CMS Pipelines* was first being developed, the author of REXX, Mike Cowlishaw, graciously made a critical change to REXX to facilitate the implementation of *Pipes*. The author of *CMS Pipelines*, John Hartmann, has himself said that there would be very little point to *CMS Pipelines* without REXX. Although *CMS Pipelines* does run at the command level (or even with EXEC 2), its real power comes when it is used in conjunction with REXX. Conversely, *CMS Pipelines* magnifies the power of REXX, and that is the second reason for discussing it here. *CMS Pipelines* brings to REXX many of the capabilities that are the subjects of user group requirements for REXX enhancements, and it also augments REXX in other important ways, such as by giving it device independence. I will try today to give you a glimpse of the ways in which REXX and *Pipes* complement one another and of the reasons why CMS REXX users are so excited about *Pipes*.

---

1 *CMS Pipelines* is part of CMS 8 in VM/ESA 1.1. Customers who are not yet running CMS 8 can order *CMS Pipelines* as a program offering, 5785-RAC, except in the United States, where *CMS Pipelines* is a Programming RPQ (P81059, 5799-DKF). The PRPQ, which includes Mike Cowlishaw’s LEXX editor, is in Higher Education Software Consortium Group I-A1.

2 UNIX is a trademark of AT&T Bell Laboratories.

The Pipeline Concept

A pipeline is simply a series of programs through which data flow, just as water flows through the sections of a water pipe. In a pipeline, a complex task is performed by processing data through several simple programs in an appropriate sequence.

The programs that are hooked together to form a pipeline are called “stages”. Each stage in a pipeline reads data from the pipeline, processes them in some way, and writes the transformed data back to the pipeline. Those data are then automatically presented as input to the next stage in the pipeline. The individual programs in the pipeline are independent of one another; they need not know or care which other programs are in the pipeline. They are also device independent; each of them does its own job without concern for where the data came from or for where they are going. The output of any program can be connected to the input of any other program; thus, the programs used to perform one task can be hooked together in a different order to perform a different task. Whenever a new pipeline stage is written, it can immediately be used in conjunction with any previously existing stage.

Pipeline programming involves applying “pipethink” to break a problem into a number of small steps, each of which can then be performed by a simple program. Wherever possible, a pipeline programmer uses existing programs as the stages in a pipeline. Traditionally, programs that run in pipelines are small and have one very well-defined function, but they should also be as general-purpose as possible, to allow re-use. Because they are so small and well-defined, it is possible to make them very reliable. In other words, programs that run in pipelines should be “little gems”. *CMS Pipelines* comes with a very rich collection of such gems, well over a hundred built-in programs. *CMS Pipelines* users typically find that most of their applications can be written using only the built-in programs, but if they have a need that is not addressed by a built-in program, they can easily craft their own little gems, preferably in REXX.

II. A *CMS PIPELINES* PRIMER

The Pipe Command

The pipeline concept has not been integrated into command parsing in CMS, as it has in UNIX. Instead, *CMS Pipelines* adds the new CMS command, PIPE:

```
pipe  pipeline-specification
```

The argument to PIPE is a “pipeline specification”. A pipeline specification is a string listing the stages to be run. The stages are separated by the “stage separator character”, which is usually a vertical bar (“|”):

```
pipe  stage-1  |  stage-2  |  stage-3  |  stage-4
```

When CMS sees this PIPE command (whether in an EXEC or typed on the command line), it passes control to the PIPE module, which interprets the argument string as a pipeline containing four stages. The pipeline parser locates the four programs and checks for correct syntax in the invocations of any that are built-in programs. If all the stages are specified correctly, the pipeline is executed; otherwise, the pipeline parser issues useful error messages and exits.

Device Drivers

In UNIX, a program can do I/O to a device in exactly the same way it does I/O to a file. Under the covers, the system has “device drivers” to make this work. Because CMS does not provide such device transparency, *CMS Pipelines* has its own device drivers, pipeline stages that connect the pipeline to host interfaces, thus allowing other pipeline stages to be completely independent of host interfaces.

CMS Pipelines provides a large number of device drivers. A very simple pipeline might contain only device drivers. We may as well be traditional and start with this one:

```
pipe literal Hello, World! | console
```

Here, the device driver literal inserts a record containing the phrase “Hello, World! ” into the pipeline. The device driver console then receives that record and displays it on the console.

This pipeline reads lines from the console and writes them to the punch:

```
pipe console | punch
```

(It continues reading from the console and writing to the punch until it reaches end-of-file, *i.e.*, until it receives a null line as input.)

As the use of console in these two examples shows, some device drivers can be used for either reading or writing. If they are the first stage in the pipeline, they *read* from the host interface. If they come later in the pipeline, they *write* to the host interface. This pipeline performs a simple echo operation:

```
pipe console | console
```

It just reads lines from the console and writes them back to the console. A similar pipeline performs a more useful task; it copies a file from one tape to another:

```
pipe tape | tape tap2 wtm
```

The first tape stage knows to read, because it can sense that it is the first stage in the pipeline; the second tape stage knows to write, because it can sense that it is *not* the first stage in the pipeline. tap2 and wtm are arguments to the second tape stage. When the pipeline dispatcher invokes the second tape stage, it passes along those arguments, which tape recognizes as instructions to use the CMS device TAP2 and to write a tapemark at the end of the data.

There are several device drivers to read and write CMS files. Some of them will look familiar to you if you know UNIX, but may look rather strange if you do not:

- The < (“disk read”) device driver reads a CMS file and inserts the records from the file into the pipeline. Thus, this pipeline copies a file from disk to tape:

  ```
  pipe < fn ft fm | tape
  ```

- > (“disk replace”) writes records from the pipeline to the CMS file specified by its arguments, replacing any existing file of the same name, so this pipeline copies a file from tape to disk:

```
pipe  tape  |  >  fn ft fm
```

- `>>` ("disk append") is the same as `>`, except that it appends an existing file of the specified name, if any, rather than replacing it. Thus, this pipeline also copies a file from tape to disk, but if the named file already exists, it is appended, not replaced:

```
pipe  tape  |  >>  fn ft fm
```

(Note that although <, >, and >> look like the UNIX redirection operators, they are actually the names of programs; like other CMS program names, they must be delimited by a blank.)

An output device driver is not necessarily the last stage of a pipeline. Output device drivers write the records they receive from the pipeline to their host interface, but they also pass those records back to the pipeline, which then presents them as input to the following stage, if there is one. For example, this pipeline reads a CMS file and writes the records to a CMS file, to the console, to the punch, and to a tape:

```
pipe  <  fn ft fm  |  >  outfn outft outfm  |  console  |  punch  |  tape wtm
```

If you wanted to include that PIPE command in a REXX EXEC, you would need to keep in mind that the entire command is a string, only portions of which should have variables substituted. Thus, in an EXEC you would write that PIPE command something like this:

```
'PIPE  <'  infn inft infm  '|  >'  outfn outft outfm  '|  console  |  punch  |  tape wtm'
```

That is, you would quote the parts that are not variable, while allowing REXX to substitute the correct values for the variable fields, the filenames.

As PIPE commands grow longer, using the linear form in EXECs becomes somewhat awkward. Most experienced "plumbers" prefer to put longer pipelines into "portrait format", with one stage per line, thus:

```
'PIPE (name DRIVERS)',
   '<' infn inft infm '|',
   '>' outfn outft outfm '|',
   'console |',
   'punch |',
   'tape wtm'
```

You can use the FMTP XEDIT macro, which comes with *CMS Pipelines*, to reformat a PIPE command into portrait format. Note the commas at the ends of the lines; those are REXX continuation characters. This pipeline specification will still be a single string once REXX has interpreted it.

Note also the "global option" name in parentheses immediately following the PIPE command. This gives the pipeline a name by which it can be referenced in a traceback, should an error occur while the pipe is running. (There are a number of other global options, but this is the only one we will meet in this session.)

Once you have the pipeline in portrait format, you can key in comments on each line and then invoke the SC XEDIT macro, which comes with *CMS Pipelines*, to line them up nicely for you:

```
'PIPE (name DRIVERS)',              /* Name for tracing */
   '<' infn inft infm '|',          /* Read CMS file    */
   '>' outfn outft outfm '|',       /* Copy to CMS file */
   'console |',                     /* And to console   */
   'punch |',                       /* And to punch     */
   'tape wtm'                       /* And to tape      */
```

You will notice that all the device drivers observe the rule that a program that runs in a pipeline should be able to connect to any other program. Although the device drivers are specialized on the side that connects to the host, they are standard on the side that connects to the pipeline.

There are four very useful device drivers to connect a pipeline to the REXX environment:

- var, which reads a REXX variable into the pipeline or sets a variable to the contents of the first record in the pipeline;
- stem, which retrieves or sets the values in a REXX stemmed array;
- rexxvars, which retrieves the names and values of REXX variables; and
- varload, which sets the values of the REXX variables whose names and values are defined by the records in the pipeline.

All four of these stages allow you to specify which REXX environment is to be accessed. If you do not specify the environment, then the variables you set or retrieve are from the EXEC that contains your PIPE command. But you may instead specify that the variables are to be set in or retrieved from the EXEC that called the EXEC that contains your PIPE command or another EXEC further up the chain, to any depth. For example, this pipeline:

```
'PIPE  stem parms.  1 | stem parms.'
```

retrieves the stemmed array “parms” from the environment one level back (that is, from the EXEC that called this EXEC) and stores it in the stemmed array “parms” in this EXEC. (If these two stages are reversed, then the array is copied in the opposite direction.)

rexxvars retrieves the names and values of all exposed REXX variables from the specified REXX environment and writes them into the pipeline, starting with the source string:

```
'PIPE rexxvars 1 | var source1'    /* Get caller's source. */
'PIPE rexxvars 2 | var source2'    /* And his caller's.    */

Parse Var source1 . . . fn1 .
Parse Var source2 . . . fn2 .

Say 'I was called from' fn1', which was called from' fn2'.'
```

In this example, rexxvars is used twice, once to retrieve the variables from the EXEC that called this one and once to retrieve the variables from the EXEC that called that one. In each case, a var stage is then used to store the first record produced by rexxvars (the source string) in a variable in this EXEC, where it can be used like any other REXX variable.

Another very useful group of stages issue host commands and route the responses into the pipeline. Among these "host command processors" are:

- cp, which issues CP commands;
- cms, which issues CMS commands with full command resolution through the CMS subcommand environment, just as REXX does for the Address CMS instruction; and
- command, which issues CMS commands using a program call with an extended parameter list, just as REXX does for the Address Command instruction.

Each of these stages issues its argument string as a command and then reads any records from its input stream and issues those as commands, too. The command responses are captured, and each response line becomes a record in the pipeline. For example, in this pipeline:

```
'PIPE cp query dasd | stem dasd.'
```

the cp stage issues a CP QUERY DASD command and writes the response into the pipeline, where the stem stage receives it and writes it into the stemmed array "DASD", setting "DASD.0" to the count of the lines in the response.

There are a great variety of other device drivers, for example:

- xedit, which writes records from an XEDIT session to the pipeline or *vice versa;*
- stack, which reads or writes the CMS program stack;
- sql and ispf, which interface to SQL and ISPF;
- qsam, which reads MVS files (and writes them under MVS);
- storage, which reads or writes virtual machine storage; and
- subcom, which sends commands to a subcommand environment.

The list of device drivers goes on and on, and it continues to grow.

Other Built-in Programs

Pipelines built only of device drivers do not really show the power of *CMS Pipelines* (although they may be quite useful, especially as they often out-perform the equivalent native CMS commands). There are dozens of other *CMS Pipelines* built-in programs. Most of these are "filters", programs that can be put into a pipeline to perform some transformation on the records flowing through the pipeline.

Using Pipeline Filters: A simple pipeline consisting of a couple of device drivers wrapped around a few filter stages provides an instant enhancement to the CMS command set. Once you have had some practice, you will find yourself typing lots of little "throwaway" pipes right on the command line.

Many *CMS Pipelines* filters are self-explanatory (especially as many of them behave just like the XEDIT subcommand of the same name). For example, this pipeline displays the DIRECTORY statement from a CP directory:

```
pipe < user direct | find DIRECTORY | console
```

The find filter selects records using the same logic as the XEDIT FIND subcommand.

This pipeline displays all the occurrences of the string “GCS” in the *CMS Pipelines* help library:

```
pipe < pipeline helpin | unpack | locate /GCS/ | console
```

The unpack filter checks whether its input is a packed file and, if it is, does the same unpack operation that the CMS COPYFILE and XEDIT commands do. The locate filter selects records using the same logic as the XEDIT LOCATE subcommand.

This pipeline tells you how many words there are in one of your CMS files:

```
pipe < plunge script a | count words | console
```

A slightly more elaborate pipeline tells you how many *different* words there are in that same file:

```
pipe < plunge script a | split | sort unique | count lines | console
```

split writes one output record for every blank-delimited word in its input; sort unique then sorts those one-word records and discards the duplicates, passing the unique records on to count lines to count. count writes a single record containing the count to its output stream. console reads that record and displays it on the console.

This pipeline writes a CMS file containing fixed-format, 80-byte records to a tape, blocking it in a format suitable to be read by other systems:

```
pipe < gqopt fortran a | block 16000 | tape
```

This pipeline writes a list of the commands used with “SMART” (RTM) to a CMS file:

```
pipe literal next| vmc smart help| strip trailing | > smart commands a
```

literal writes a record containing the word “next”. The vmc device driver sends a help command to the SMART service machine via VMCF and writes the response to the pipeline. It then reads the single record from its input and sends a next command to the SMART service machine, again writing the response to the pipeline. strip trailing removes trailing blanks from the records that pass through it, thus turning the blank lines in the response from SMART into null records. > reads records from its input, discards those that are null, and writes the others to the file SMART COMMANDS A.

And here is a pipeline I especially like; it would be typed on the XEDIT command line:

```
pipe cms query search | change //INPUT / | subcom xedit
```

In this pipeline, the cms device driver issues the CMS QUERY SEARCH command and routes the response into the pipeline; the change filter (which works like the XEDIT CHANGE subcommand) changes each line of the response into an XEDIT INPUT subcommand; and then subcom sends each line to XEDIT, which executes it as a command. This is a very easy way to incorporate the response from a command into the text of a file you are editing.

The Specs Filter: Now, let's look at one of the less obvious filters, specs. specs selects pieces of an input record and puts them into an output record. It is very useful and not really as complex as it looks at first. Its syntax was derived from the syntax for the SPECS option of the CMS COPYFILE command, but it has long since expanded far beyond the capabilities of that option:

- The basic syntax of specs is:

  specs input-location output-location

  with as many input/output pairs as you need.

- The input location may be a column range, such as "10-14". "10.5" means the same thing as "10-14". "1-*" means the whole record. "words 1-4" means the first four blank-delimited words. The input may also be a literal field, expressed as a delimited string, such as "/MSG/", or it may be "number", to get a record number.

- The output location may be a starting column number, or "next", which means the next column, or "nextword", which leaves one blank before the output field.

- A conversion routine, such as "c2d", may be specified between the input location and the output location. The specs conversion routines are similar to the REXX conversion functions and are applied to the value from the input field before it is moved into the output field.

- A placement option, "left", "center", or "right", may be specified following the output location; for example, "number 76.4 right" puts a 4-digit record number right-aligned starting in column 76.

```
/* PIPEDS EXEC:                     Find lrecl of an OS dataset */

Parse Upper Arg dsname fm

'PIPE (name PIPEDS)',
   'command LISTDS' fm '( FORMAT |',    /* Issue LISTDS.   */
   'locate /' dsname '/ |',       /* Locate file we want.  */
   'specs  word 2  1 |',          /* Lrecl is second word. */
   'console'                      /* Display lrecl.        */
```

PIPEDS EXEC is a simple example of using specs. PIPEDS displays the logical record length of an OS dataset. The command stage issues a CMS LISTDS command with the FORMAT option and routes the response into the pipeline, where locate selects the line that describes the specified dataset, *e.g.*:

```
U        6447  PO    02/25/80 RES342 B  SYS5.SNOBOL
```

specs selects only the second word of that line, the logical record length ("6447"), and moves it to column 1 of its output record, which console then reads and displays.

pipe < cms exec a | specs 1-27 1 8-27 nextword | > cms exec a

This is another simple example of using specs. The arguments to specs here are two pairs of input-output specifications. The first input-output pair (“1-27 1”) copies the data from columns 1-27 of the input record to columns 1-27 of the output record. The second input-output pair (“8-27 nextword”) copies the data from columns 8-27 of the input record to columns 29-48 of the output record; that is, a blank is left between the first output field and the second output field. So, this pipeline would be used to duplicate the filenames in a CMS EXEC created by the EXEC option of the CMS LISTFILE command. (This pipeline is almost 500 times as fast as the XEDIT macro I used to use to do this same thing.)

Augmenting REXX: People often start in gradually using *CMS Pipelines* in EXECs, first just taking advantage of the built-in programs that supply function that is missing or awkward in REXX. Here is a function that has been implemented a zillion times in REXX or Assembler:

'PIPE stem bananas. | sort | stem bunch.'

That sorts the values in the stemmed array “bananas” and puts them into the array “bunch”.

Here is an example of using specs to augment REXX (which has no “c2f” function):

'PIPE var cpu2busy | specs 1-* c2f 1 | var cpu2busy'

The device driver var picks up the REXX variable “cpu2busy”, which contains a floating-point number stored in the System/370 internal representation (*e.g.*, '4419B600'x), and writes it to the pipeline. specs reads the record passed from var and converts it to the external representation of the floating-point number (6.582E+03), and then the second var stage stores the new representation back into the same REXX variable, allowing it to be used in arithmetic operations.

Another function *CMS Pipelines* brings to REXX programmers is an easy way to process all the variables that have a given stem. In the example below, rexxvars writes two records into the pipeline for each exposed variable. One record starts with “n ” and contains the variable’s name; the other starts with “v ” and contains its value. The find stage selects only the name records for variables with the stem “THINGS”. specs removes the “n ”, and stem puts the names of the “THINGS” variables into the stemmed array “vars”, where they can be accessed with a numeric index. (The buffer stage prevents the stem stage from creating new variables while rexxvars is still loading the existing variables.)

```
'PIPE',                       /* Discover stemmed variables:  */
   'rexxvars |',              /*   Get all variables.         */
   'find n THINGS.|',         /*   Select names of THINGS.    */
   'specs 3-* 1 |',           /*   Remove record type prefix. */
   'buffer |',                /*   Hold all records.          */
   'stem vars.'               /*   Names of THINGS into stem. */

Do i = 1 to vars.0
   Say vars.i '=' Value(vars.i)
End
```

rexxvars has many other uses; for example, you might wish to use it in a syntax error routine to dump all exposed variables to a file for debugging. The combination of rexxvars and varload provides such capabilities as saving the state of an EXEC and later restoring it.

varload uses the information in its input records to set REXX variables. The input to varload consists of records that contain a delimited string specifying a variable name, followed by the value to which the variable is to be set. The canonical example of using varload and rexxvars is a pair of EXECs written by Jim Colten, of the University of Minnesota, with contributions by Chuck Boeheim and Michael Friendly. The first one is called to save all CP settings:

```
/* CPQSET EXEC: Load CP SET values into REXX stem.          */

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

The cp stage issues a CP QUERY SET command and routes the response into the pipeline, where the split stage splits the records at the commas, thus producing one record for each CP setting. The specs stage converts these records into the format required by varload: a delimited string containing the name (in this case, of the form “=CPVAR.xxx=”), followed by the value. varload 1 receives these records and loads the specified variables into the caller’s environment. The companion EXEC performs the inverse operation:

```
/* CPRESET EXEC: Restore CP variables from REXX stem.       */

'PIPE (name CPRESET)'         ,
   '  rexxvars 1'             ,    /* Get caller's variables. */
   '| drop 1'                 ,    /* Drop source line.       */
   '| spec 3-* 1'             ,    /* Join name & value,      */
   '    read 3-* nextword'    ,    /*   removing type prefix. */
   '| find CPVAR.'         || ,    /* Only our stem.          */
   '| nfind CPVAR.0'          ,    /* Discard the counter.    */
   '| nlocate /ECMODE/'       ,    /* SET ECMODE is BAD!      */
   '| spec /CP SET/ 1'        ,    /* Make into CP command,   */
   '    7-* nextword'         ,    /*   removing stem name.   */
   '| cp'                     ,    /* Let CP do reSET.        */
   '| console'                     /* Display any messages.   */
```

Replacing EXECIO: EXECIO is usually the first thing to go when one learns *CMS Pipelines*. Anything that can be done with EXECIO can be done with *CMS Pipelines*, generally faster and always more straightforwardly. (And replacing EXECIO with a pipeline makes it easier to port an EXEC between CMS and MVS.) Let's look at a few EXECIO examples from various IBM manuals, along with the equivalent pipelines:

- These both read the first three records of a CMS file into the stemmed array "X" and set the value of "X.0" to 3:

```
'EXECIO  3  DISKR  MYFILE DATA *  1  ( STEM X.'

'PIPE  <  myfile data *  |  take 3  |  stem x.'
```

- These both issue a CP QUERY USER command in order to set a return code (without saving the response):

```
'EXECIO  0  CP  ( STRING QUERY USER GLORP'
+++ RC(1045) +++

'PIPE  cp query user glorp'
+++ RC(45) +++
```

- These both put a blank-delimited list of the user's virtual disk addresses into the REXX variable "used":

```
Signal Off Error
'MAKEBUF'
Signal On Error
theirs = Queued()
'EXECIO * CP ( STRING Q DASD'
used = ''
Do While Queued() > theirs
   Pull . cuu .
   used = used cuu
End
'DROPBUF'

'PIPE  cp q dasd  |  specs  word  2  1  |  join  *  /  /  |  var used'
```

The EXECIO case comes from the REXX *User's Guide*. Admittedly, it is rather old-fashioned code; nevertheless, its eleven lines make up an all too familiar example of manipulating the CMS stack. In the pipeline, the cp device driver issues the CP QUERY DASD command and routes the response into the pipeline. specs selects the second word from each input record and makes it the first (and only) word in an output record. join * joins all these records together into one record, inserting the delimited string in its argument (a blank) between the values from the individual input records. And var stores this single record into the variable "used".

Pipeline Programs: After a while, you will find yourself not just augmenting your EXECs with small pipes, but also writing EXECs that are predominantly pipes, such as REACCMSG EXEC:

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

Parse Arg vaddr .

'PIPE (name REACCMSG)',
   'cp q links' vaddr '|',          /* Issue CP QUERY LINKS    */
   'split at , |',                  /* Get one user per line   */
   'strip |',                       /* Remove leading blanks   */
   'sort unique 1-8 |',             /* Discard duplicates      */
   'specs /MSG/ 1',                 /* Make into MSG commands  */
      'word 1 nextword',            /* Fill in userid          */
      '/Please re-ACCESS your/ nextword',
      'word 2 nextword',            /* Fill in virtual address */
      '/disk./ nextword |',
   'cp'                             /* Issue MSG commands      */
```

REACCMSG is used to send a message to all the users linked to a particular CMS disk to let them know that they should re-ACCESS the disk because it has been changed. It uses built-in programs we have seen before, but in a slightly more sophisticated manner: split receives the response from the CP QUERY LINKS command:

```
PIPMAINT 320 R/O, MAINT    420 R/O, TDTRUE   113 R/O, Q0606    320 R/O
Q0606    113 R/O, SERGE    420 R/O
```

and splits those records into multiple records by breaking them up at the commas between items; strip removes the leading blanks; and sort unique sorts the records on the userid field in the first eight columns and discards any duplicates, so that each user will be sent only one message. This example shows a more elaborate use of specs than before, but it is not difficult to understand if you keep in mind that specs's arguments are always pairs of definitions for input and output. This specs stage has been written in portrait format with each input-output pair on a separate line. You will note that the input definitions in three of the five pairs here are for literals. The first input-output pair puts the literal “MSG” into columns 1-3 of the output record; the second pair puts the userid from the first word of the input record (“word 1”) into columns 5-12 of the output record; and so on. Then as each record flows from the specs stage to the cp stage, cp issues it as a CP MSG command.

The next example is a simple service machine that uses the starmsg device driver to connect to the CP *ACCOUNT system service, so that it can monitor attempts to LOGON to the system with an invalid password. Each time CP produces an accounting record, this starmsg stage receives that record via IUCV and writes it to the pipeline (prefacing it with an 8-byte header). The locate stage discards all but the “Type 4” records, which are the ones that CP produces when the limit of invalid LOGON passwords is reached. specs formats a message containing a literal and three fields from the accounting record, which console then displays. (Note the stage separators on the left side here. This is a widely used alternative portrait format.)

This pipeline runs until you stop it by using the haccount immediate command, which *CMS Pipelines* sets up for you when it establishes the connection to the *ACCOUNT system service. starmsg can also be used to connect to several other CP system services, including *MSG and *MSGALL.

```
/* HACKER EXEC:          Display Type 4 Accounting Records. */

'CP RECORDING ACCOUNT ON LIMIT 20'

'PIPE (name STARMSG)',
   '| starmsg *account',         /* Connect to *ACCOUNT.     */
   '| locate 88 /4/',            /* Only Type 4 records.     */
   '| specs',                    /* Format warning message:  */
   '     /Hacker afoot? / 1',    /*    literal,              */
   '     9.8 next',              /*    ACOUSER,              */
   '     37.4 nextword',         /*    ACOTERM@,             */
   '     79.8 nextword',         /*    ACOLUNAM.             */
   '| console'                   /* Display on console.      */

If Userid() <> 'OPERACCT'
Then 'CP RECORDING ACCOUNT OFF PURGE QID' Userid()
```

The next example may be a bit arcane, but it can be very useful; it reads a file containing textual material of arbitrary content and record length and produces a file containing the same text formatted as Assembler DC instructions for use, say, as messages:

```
/* MAKEDC EXEC:   Reformat text into Assembler DC statements */

Parse Arg fn ft fm .          /* File to be processed.       */

"PIPE (name MAKEDC)",
   "<" fn ft fm "|",          /* Read the file.              */
   "change /&/&&/ |",         /* Double the ampersands.      */
   "change /'/''/ |",         /* Double the single quotes.   */
   "specs",                   /* Reformat to DC statement:   */
   "   /DC/ 10",              /*    literal "DC" in col 10;  */
   "   /C'/ 16",              /*    literal "C'" in col 16;  */
   "   1-* next",             /*    entire record next; and  */
   "   /'/ next |",           /*    terminate with quote.    */
   "asmxpnd |",               /* Split to continuations.     */
   ">" fn "assemble a"        /* Write the new file.         */
```

The two change filters double any ampersands or quotes in the text. For each input record, specs builds an output record that has “DC” in column 10 and “C” in column 16, followed by the input record enclosed in single quotes. asmxpnd then examines each record to determine whether it extends beyond column 71; if so, it breaks the record up into two or more records formatted in accordance with the Assembler’s rules for continuations. And finally, > writes the reformatted records to a CMS file. Thus, if the input file were to contain the line:

The PACKAGE file records have ' &1 &2 ' in columns 1-7 and a filename, then these two records would appear in the output file:

```
         DC    C'The PACKAGE file records have '' &&1 &&2 '' in columns*
                1-7 and a filename,'
```

Selection Filters: There are many more *CMS Pipelines* filters to learn, but I want to mention one class in particular, the selection filters:

between	frlabel	nfind	outside	unique
drop	inside	nlocate	take	whilelab
find	locate	notinside	tolabel	

The selection filters are used to select certain records from among those passing through the pipeline, while discarding all others. A cascade of selection filters can quickly select the desired subset of even a very large file. I routinely use pipelines to filter files containing tens (or even hundreds) of thousands of records to select the records I need for some purpose.

One simple example is a filter I use with the NETSTAT CLIENTS command. NETSTAT CLIENTS produces hundreds of lines of output, several lines for each user who has used TCP/IP since the last IPL. The first line of the response for each user begins with the string “Client:” followed by the userid; and one of the other lines begins with the string “Last Touched:”. Usually, when I issue a NETSTAT CLIENTS command, I need to see only these two lines for each of four servers. The eight lines I want are easily isolated using two selection filters:

```
/* STATPIPE EXEC:      Display "Last Touched" for BITFTPn. */

'PIPE',
   'command NETSTAT CLIENTS |',
   'between   /Client: BITFTP/ /Last Touched:/ |',
   'notinside /Client: BITFTP/ /Last Touched:/ |',
   'console'
```

The command stage issues a NETSTAT CLIENTS command and routes the response into the pipeline. The between filter selects *groups* of records; its arguments are two delimited strings, describing the first and last records to be selected for each group. So, the between stage here selects groups of records that begin with a record that begins “Client: BITFTP” and that end with a record that begins “Last Touched:”. notinside then further refines the data by selecting only those records that are *not* between a record that begins with “Client: BITFTP” and a record that begins with “Last Touched:”. That leaves us with only those two lines for each client I am interested in, the ones whose userids start “BITFTP”.

You will likely find that many of your pipelines process the output of CP or CMS commands or CMS or MVS programs. The output from UNIX commands and programs is generally designed to be processed by a pipe, so it tends to be essentially “pure data”, with few headers and trailers. With CP, CMS, and MVS output, however, you generally need to winnow out the chaff to get down to the data. Although I cannot go over the selection filters in detail today, they are easy to use and quite powerful, so you should not hesitate to process listing files that were designed to be read by humans and that have complicated headers and trailers and carriage control. It is very easy to write a pipe that reads such a file and pares it down to the bare data.

LIST2SRC EXEC is an example of really using the selection filters; I will leave the detailed interpretation of LIST2SRC as an exercise for you. Basically, LIST2SRC reads a LISTING file produced by Assembler H and passes it through a series of selection filters, winnowing out the chaff in order to reconstruct the original source file. Although this is a “quick & dirty” program (and not quite complete), it is a good example of “pipethink”, of solving a complex problem by breaking it up into simple steps:

```
/* LIST2SRC EXEC:       Re-create the source from a LISTING file */
Signal On Novalue

Parse Arg fn .

'PIPE (name LIST2SRC)',
   '| <' fn 'listing *',          /* Read the LISTING file       */
   '| mctoasa',                   /* Machine carriage ctl => ASA */
   '| frlabel -  LOC',            /* Discard to start of program */
   '| drop 1',                    /* Drop that '-  LOC' line too */
   '| tolabel - POS.ID',          /* Keep only up to relocation  */
   '| tolabel -SYMBOL',           /*    dictionary or cross-ref  */
   '| tolabel 0THE FOLLOWING STATEMENTS',  /*    or diagnostics  */
   '| outside /1/ 2',             /* Drop 1st 2 lines on each pg */
   '| nlocate 5-7 /IEV/',         /* Discard error messages      */
   '| nlocate 41 /+/',            /* Discard macro expansions    */
   '| nlocate 40 /                  /',  /* Discard blank lines  */
   '| specs 42.80 1',             /* Pick out source "card"      */
   '| >' fn 'assemble a fixed'    /* Write new source (RECFM F)  */
```

Subroutine Pipelines

Once you have been using *CMS Pipelines* for a while, you may find that there are some sequences of stages that you use often:

pipe stage-a | **stage-b | stage-c | stage-d** | stage-e

pipe stage-x | **stage-b | stage-c | stage-d** | stage-y

In that case, it is time to move those stages into a subroutine pipeline, polish them a bit, generalize them a bit, and create your own little gem:

```
/* MYSUB REXX */

'CALLPIPE  *: | stage-b | stage-c | stage-d | *:'
```

Then whenever you need the function performed by your subroutine, you simply use its name as a stage name (“mysub” in this case):

pipe stage-a **|** **mysub** **|** stage-e

pipe stage-x **|** **mysub** **|** stage-y

The subroutine may look a bit mysterious, but it is simply a pipeline stage written in REXX. If we look at it again in portrait format, it can be demystified quickly:

```
/* MYSUB REXX:                Generic subroutine pipeline  */

'callpipe',                       /* Invoke pipeline       */
   '*: |',                        /* Connect input stream  */
   'stage-b |',
   'stage-c |',
   'stage-d |',
   '*:'                           /* Connect output stream */

Exit RC
```

There are just a few things one needs to understand about subroutine pipelines:

1. The *CMS Pipelines* command callpipe says to run a subroutine pipeline; callpipe has the same syntax and the same options as the PIPE command itself.

2. Those “*:” sequences are called “connectors”. The connector at the beginning tells the pipeline dispatcher to connect the *output* from the previous stage of the calling pipeline to the *input* of the first stage of this subroutine pipeline, stage-b. The connector at the end says to connect the *output* from the last stage of this subroutine pipeline, stage-d, to the *input* of the next stage in the calling pipeline.

3. When you use REXX to write an XEDIT subroutine, the default subcommand environment is XEDIT. Similarly, when you use REXX to write a *CMS Pipelines* subroutine, the default subcommand environment executes *CMS Pipelines* commands. Thus, if you wish to issue CP or CMS commands in your subroutine, you will need to use the REXX Address instruction.

4. When you use REXX to write an XEDIT subroutine, the subroutine has a filetype of XEDIT, but when you use REXX to write a *CMS Pipelines* subroutine, the filetype is *not* PIPE. It is REXX.

5. Arguments passed to a subroutine are available to the REXX Parse Arg instruction.

Let’s look at an example of a real subroutine pipeline, HEXSORT, which sorts hexadecimal numbers. An ordinary sort does not work for hexadecimal numbers *(i.e.,* base 16 numbers, expressed with the “numerals” 0-9, A-F), because the EBCDIC collating sequence sorts A-F before 0-9. This handy little subroutine pipeline sorts hexadecimal data correctly by using the trick of temporarily translating A-F to characters higher in the collating sequence than 0-9 (which are F0-F9 in hexadecimal):

```
/* HEXSORT REXX:           Hexadecimal sort,   0123456789ABCDEF  */

Parse Arg sortparms                          /* Get parms, if any     */

'callpipe (name HEXSORT)',                   /* Invoke pipeline       */
   '*: |',                                   /* Connect input stream  */
   'xlate 1-* A-F fa-ff fa-ff A-F |',        /* Transform for sort    */
   'sort' sortparms '|',                     /* Sort w/caller's parms */
   'xlate 1-* A-F fa-ff fa-ff A-F |',        /* Restore               */
   '*:'                                      /* Connect output stream */

Exit RC
```

The arguments to the xlate stages here are a column range, "1-*", which means the entire record, followed by pairs of character ranges specifying "to" and "from" translations. Records flow in from the calling pipeline through the beginning connector; they are processed through the xlate, sort, and xlate stages; and then they flow out through the end connector back into the calling pipeline. If the caller specifies an argument, that argument is passed to the sort stage to define a non-default sort operation. Here is a typical invocation:

'PIPE stem mdisk. | hexsort 7.3 | stem mdisk.'

That sorts an array of minidisk records from a CP directory into device address order. (The device addresses are hexadecimal numbers in columns 7-9 of the minidisk records.)

Of course, it is not necessary to put these operations into a subroutine. You could simply use the xlate-sort-xlate sequence in all your pipelines, whenever you need to do a hexadecimal sort, but it is much better to hide such complexity. Once you have this subroutine built, you can invoke it by name from any number of pipelines and need never think about the problem again.

Furthermore, by building the subroutine with a simple, well-defined interface and at the same time making its function as generic as possible, you create a piece of code that can be used over and over again. Here is another example of invoking HEXSORT:

'PIPE cp q nss map | drop 1 | hexsort 33-44 | > nss map a'

That issues a CP QUERY NSS command, drops the header line from the response, and sorts the remaining lines to produce a list of saved systems in memory address order. (The virtual memory addresses are hexadecimal numbers starting in column 33 of the response lines.)

A subroutine pipeline is often the cleanest way to package a function that you have implemented with *CMS Pipelines*. If you make it a subroutine pipeline, then the people you give it to can easily invoke it from their own pipes.

Writing REXX Filters

The time will come when you have a problem that cannot be solved by any reasonable combination of *CMS Pipelines* built-in programs. You will need to write a filter of your own, preferably in REXX. A REXX filter is similar to the simple subroutine pipelines we have just been looking at. It has a filetype of REXX; its subcommand environment executes *CMS Pipelines* commands; it is invoked by using its name as a stage in a pipeline; and it can receive passed arguments.

You will find writing your own pipeline filters in REXX to be very easy once you understand the basics. When I am writing a filter, I always start with this dummy filter that does nothing at all except pass records through unchanged:

```
/* 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    */
```

There are only a few new things one needs to learn to understand this REXX filter:

1. The *CMS Pipelines* command readto reads the next record from the pipeline into the specified REXX variable (“record” in this case).

2. The *CMS Pipelines* command output writes a record to the pipeline. The contents of the record are the results of evaluating the expression following the output command (again, in this case, the value of the REXX variable “record”).

3. The pipeline dispatcher sets return code 12 to indicate end-of-file. A readto command completes with a return code of 12 when the stage before it in the pipeline has no more records to pass on to it. An output command completes with a return code of 12 when the stage following it in the pipeline has decided to accept no more input records.

So, this filter, NULL, reads a record from the pipeline and writes it back to the pipeline unchanged. It keeps on doing that until an error is signalled, *i.e.*, until a non-zero return code is set. That causes a transfer to the label “Error” in the last line of the EXEC. The most likely non-zero return code would be a return code 12 from the readto command, which would indicate end-of-file on the input stream, but the output command could get return code 12 instead, or there could be a real error. If the return code is 12, then before exiting the filter sets its own return code to 0 to indicate normal completion. Any other return code is passed back to the caller.

The effect of including the NULL filter in a pipeline:

pipe stage-a | **null** | stage-b | stage-c is simply to make the pipeline run a bit slower. But once you understand NULL, you can quickly go on to writing useful filters, such as REVERSE, which reverses the contents of the records that pass through it:

```
/* REVERSE REXX:          Filter that reverses records  */
Signal On Error

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

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

We can make that example slightly more complex, to illustrate one more concept that you will need when writing filters. This filter reverses only the even-numbered lines passing through it:

```
/* BOUSTRO REXX:   Filter that writes records boustrophedon */
Signal On Error

Do recno = 1 by 1                      /* Do until EOF      */
   'readto record'                     /* Read from pipe    */
   If recno // 2 = 0                   /* If even-numbered  */
      Then record = Reverse(record)    /*    line, reverse  */
   'output' record                     /* Write to pipe     */
End

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

Each stage in a pipeline runs as a "co-routine", which means that it runs concurrently with the other stages in the pipeline. It is invoked once, when the pipeline is initiated, and remains resident. So, when BOUSTRO is ready for another record, it calls upon the pipeline dispatcher by doing a readto. The dispatcher may then decide to dispatch some other co-routine, but it will eventually return control to this one, which will continue reading and writing records until an error is signalled. Thus, when you are writing a *CMS Pipelines* filter, you need not worry (as I did at first) about where to save local variables, such as "recno" here, between "calls" to your filter. Your filter is called only once and then runs concurrently with the other stages in the pipeline. There is nothing special that your filter needs to do in order to run concurrently with the other stages; the pipeline dispatcher takes care of all that for you.

I would like to show one more example of a simple REXX filter, AVERAGE, which illustrates the point that your filter can decide not to write a record back to the pipeline for every record it reads from the pipeline. AVERAGE first reads all the input records; then, when it gets end-of-file on its input, it calculates the contents of a single output record and writes that to the pipeline:

```
/* AVERAGE REXX:                  Filter that averages input  */
Signal On Error

acum = 0                                 /* Initialize        */

Do nobs = 0 by 1                         /* Do until EOF      */
   'readto record'                       /* Read from pipe    */
   Parse Var record number .             /* Get number        */
   acum = acum + number                  /* Accumulate        */
End

Error:  If RC = 12                       /* If EOF, then      */
   Then 'output' Format(acum/nobs,,2)    /*    write average  */
Exit RC*(RC<>12)                         /* RC = 0 if EOF     */
```

Differences from UNIX Pipes

That is as many examples of using pipelines in CMS as we have time for right now. I have pointed out some of the differences between the UNIX and CMS implementations of pipelines. You may have noticed some of the others:

- As you would expect, *CMS Pipelines* is record-oriented, rather than character-oriented.

- *CMS Pipelines* implements asynchronous input, immediate commands, and dynamic reconfiguration of pipeline topology.

- *CMS Pipelines* implements multi-stream pipelines. These networks of interconnected pipelines allow selection filters to split a file into streams that are processed in different ways. The streams can then be recombined for further processing.

- Most *CMS Pipelines* stages run unbuffered; that is, they process each input record as soon as it is received and pass it on to the following stage immediately. (Of course, some pipeline stages, such as sort, must, by their nature, be buffered.) Running the stages unbuffered is necessary to allow records flowing through a multi-stream pipeline to arrive at the end in a predictable order. It can have the advantage of greatly reducing the virtual memory requirements. Thus, *CMS Pipelines* can often be used to perform operations that cannot be done with XEDIT because of virtual memory constraints.

- *CMS Pipelines* runs a pipeline only after all its stages have been specified correctly.

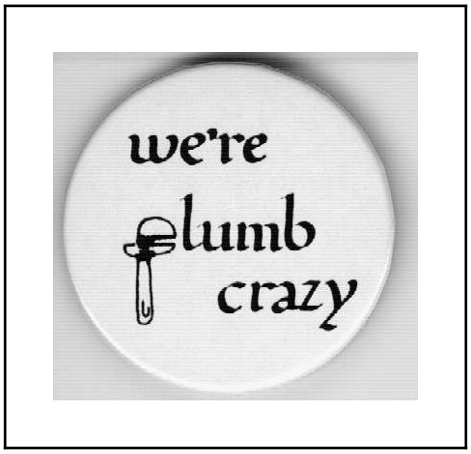
- *CMS Pipelines* programs can co-ordinate their progress via “commit levels” and can stop the pipeline when a program encounters an error.

- When the *CMS Pipelines* PIPE command completes, it sets its return code to the worst of the return codes set by the stages in the pipeline.

To sum up the differences between UNIX pipes and *CMS Pipelines*, let me quote a colleague of mine who said recently, "You know what I *really* miss in UNIX? *CMS Pipelines*!"

Advanced Topics

I have had time to give you only a flavor of *CMS Pipelines*. I have barely alluded to multi-stream pipelines, a very powerful extension to the basic pipeline concept with which you will want to become familiar. I also have not described using *CMS Pipelines* under GCS, TSO, and MUSIC. *CMS Pipelines* can now be ordered with MUSIC, and although it is not officially supported for TSO and GCS, it contains device drivers developed specifically for those environments. The *Program Directory* for the PRPQ has an appendix that discusses installing and using *CMS Pipelines* with all three of these systems.



III. WHY YOU SHOULD TAKE THE PLUNGE NOW

I have become convinced that any CMS user who writes REXX programs should learn to use *CMS Pipelines* as soon as possible. By the time I had been using *CMS Pipelines* for a few months, it had "saved my life" twice. In one case, I almost missed my plane to SHARE, due to a last-minute problem, but I was able to write a pipe to solve that problem before dashing out the door just in time. Then, a few weeks later, the systems in our SSI complex went into "yo-yo mode" after a service machine went into a loop creating spool files; I was finally able to get out of the problem by quickly keying in a command-line pipe to purge those files before the systems crashed again. *CMS Pipelines* can do the same sorts of things for you.

CMS Pipelines is a Powerful Application Enabler

CMS Pipelines makes CMS programmers more productive, so programs get written that would not get written without *CMS Pipelines*. (And programs that use *CMS Pipelines* are often much faster than if they had been written some other way.)

To give you a feeling for the variety of ways *CMS Pipelines* can be used, I will list a few of the ways I have used it so far myself:

- To analyze many kinds of data, including system accounting data, system performance data, and logs from service machines. Because *CMS Pipelines* is such a powerful tool, I find myself doing more thorough analyses and getting the answer down to a single page more often than I used to.
- To mend our system accounting data (more times than I care to admit).
- To charge for the use of our UNIX systems. This required writing a couple of Assembler-language filters, which turned out to be easy to do.
- To merge MVS RMF data into my VMAP ACUM files, so that I could plot the CPU utilization of our SPMODE native processor along with the utilization of the processors that the CP monitor knows about.
- To implement a full-blown service machine, with timer, IUCV, and I/O interrupts all handled without a line of Assembler code.
- To write a simple image enhancement program.
- To merge a PC database with a mainframe database.
- To augment and circumvent SES. (This seems to be a rapidly growing trend.)
- To build numerous tools to help me in my daily work, such as an XEDIT macro that understands CMS UPDATE control files and can pipe the next update onto the current file in the XEDIT ring.

I reached some sort of new plateau the first time I used *CMS Pipelines* to write a pipe to write a pipe. To celebrate that, I used *CMS Pipelines* to do this:

```
//  EXEC  PGM=PIPE,
//      PARM='literal Hello, World  |  change /World/Mom/  |  console'
```

(If you put *CMS Pipelines* into an MVS loadlib, it figures out where it is and struggles on. In this case, when the console stage finds no console to write to, it uses a WTP macro.)

You Can Get Lots of Help in Learning CMS Pipelines

There are several good sources for learning *CMS Pipelines* and for getting assistance if you have questions:

***CMS Pipelines Tutorial* (GG66-3158)**: This Washington Systems Center Bulletin provides an excellent introduction to *CMS Pipelines*, and I strongly recommend it to anyone who wants to plunge into *"Pipes"*. My advice is to read this manual and work the exercises at the end of each section. Then make a conscious effort to use *CMS Pipelines* in your daily work. Before long, you will find that "pipethink" has become second nature.

***CMS Pipelines User's Guide* (SL26-0018)**: This is a rather awe-inspiring manual—300 pages without an ounce of fat on it. It contains a good tutorial and is also the reference manual and the messages manual for *CMS Pipelines*. Read the "Syntax Notation" chapter before using it as a reference manual. (The same information can be obtained by typing the command pipe help syntax.) Incidentally, if you have only the “-00” version of this manual, you should order the “-01” version, which is substantially expanded and improved.

“Pipe help”: *CMS Pipelines* provides help files that can be used with the CMS HELP command, but using the pipe help command is less painful. One especially nice feature of the pipe help command is that if you issue it with no arguments, it gives you help for the last error message that *CMS Pipelines* issued, while pipe help 1 gives you help for the one before that, and so on.

VMSHARE: The VMSHARE electronic conference has several active files dealing with *CMS Pipelines*, notably Memo Pipeline and Prob Pipeline. If you ask a *CMS Pipelines* question on VMSHARE, you will almost certainly get it answered within hours. (Inside IBM, Pipeline Forum on IBMVM is another good place to get help with *CMS Pipelines*.)

CMSPIP-L: The BITNET discussion list for *CMS Pipelines*, CMSPIP-L, is a good place for asking questions if you do not have access to VMSHARE. Several “master plumbers” participate in the list. CMSPIP-L is housed at Marist College (MARIST on BITNET or vm.marist.edu on the Internet) and at the Institute for Medical Computer Science of the University of Vienna (AWIIMC12 on EARN or awiimc12.imc.univie.ac.at on the Internet). If you can contrive to get electronic mail into BITNET/EARN or the Internet, you can subscribe to this list by sending mail to LISTSERV at one of these two sites. The body of your mail should contain the command:

```
SUBSCRIBE  CMSPIP-L  your name
```

The LISTSERVs at Marist and Vienna maintain archives of the discussions from the list, as well as an archive of useful pipes. You can get a list of what the nearest archive has available by sending its LISTSERV mail containing the command:

```
GET  CMSPIP-L  FILELIST
```

Pipedemo: Chuck Boeheim, of SLAC, has written a wonderful program called Pipedemo which “animates” a pipeline to illustrate the flow of data from stage to stage. You can download Pipedemo from Note Pipedemo on VMSHARE or order it from LISTSERV. Pipedemo is also available on the VM Workshop Tools Tape for 1991.

To use Pipedemo, you simply write a normal pipeline specification but change PIPE commands to pipedemo and change callpipe commands to rexx pdcall. Running a few *CMS Pipelines* examples through Pipedemo is an excellent way to get a deeper understanding of how pipelines work. Pipedemo can also be a big help in understanding why one of your pipelines is not working. Pipedemo is itself a pipeline, of course, and is well worth reading as an example of skillful use of *CMS Pipelines*.

CMS Pipelines Explained: This new paper by John Hartmann, the author of *CMS Pipelines*, provides many extremely useful insights into how *“Pipes”* works. I strongly recommend it.

ESA Manuals: New *CMS Pipelines* manuals will soon be issued for ESA 1.1. The new *Pipelines User’s Guide* (SC24-5609) is essentially an updated version of the *Tutorial*; it reflects the changes in message numbers and the HELP facility that were required when *“Pipes”* was incorporated into CMS 8. There is also a completely new *CMS Pipelines* reference manual for CMS 8, *Pipelines Reference* (SC24-5592). However, I cannot recommend that book for any but the most casual users of *CMS Pipelines*. Even if you are running CMS 8, I suggest that you order the PRPQ manual. (Unfortunately, if you are on CMS 8, you will be stuck with help files based on the new *Reference*, unless you also order the PRPQ and load the help files from there.)

CMS Pipelines Pays Back Your Investment Quickly

CMS Pipelines is an extremely powerful facility with very rich function. There is a lot to learn. After not quite two years of using *CMS Pipelines*, I still frequently find myself saying, “Wow! I didn’t know that!” or “I never thought of using it *that* way!” Although I am still a long way from having completely mastered *CMS Pipelines*, it has, nevertheless, been making my life easier since the day I installed it.

You do not have to understand all of *CMS Pipelines* to benefit from using it. The learning curve, though long, is not steep. You do not need to read the entire *User’s Guide* before starting to use *CMS Pipelines*; indeed, you do not need to read the entire *Tutorial* before starting. I can guarantee that if you spend two or three hours reading the first few sections of the *Tutorial* and working the exercises, you will learn enough about *CMS Pipelines* that you will never again need to use EXECIO.

I recall that when I started learning REXX, there were several pleasant surprises:

- Programming in REXX was more fun than programming in other languages I had used.

- Because the REXX language was so powerful, I could write more programs and I could go further with them than I would have had the time for otherwise, so I ended up providing richer function and handling error conditions better.

- Even more pleasant was finding that my REXX programs tended to work correctly once I got them correct syntactically (or soon after that), which had seldom been my experience with programs I had written in other languages. The structure of the REXX language was disciplining my thinking so that I was programming not only more easily but also better.

I see these same effects even more strongly when I combine REXX with *CMS Pipelines*:

- Programming with *“Pipes”* is even more fun. It has restored my delight in CMS. I cannot imagine going back to not having *CMS Pipelines*. As my colleague Serge Goldstein was heard to exclaim a few weeks after we got it, “I can’t do *anything* without *Pipes*!”

- The power of *CMS Pipelines* allows me to write programs that I would not have found the time (or the stamina) to write before. I am writing more programs and giving them richer function.

- My *CMS Pipelines* programs have fewer bugs. The processes of applying “pipethink” and of visualizing the flow of data through my pipelines make me a better programmer.

If you will give it a try, I think you will find, as I have, that *CMS Pipelines* is a tool for unclogging the brain.

Appendix A

CMS PIPELINES BUILT-IN PROGRAMS

Device Drivers

- ***CMS files***

  - < Read a CMS file
  - > Replace or create a CMS file
  - >> Append to or create a CMS file
  - **Aftfst** Write information about open files
  - **Disk** Read, create, or append to a CMS file
  - **Diskback** Read a CMS file backwards
  - **Diskrand** Read a CMS file non-sequentially
  - **Diskslow** Read, append to, or create a CMS file
  - **Diskupdate** Replace records in a CMS file
  - **Getfiles** Read files
  - **State** Provide information about CMS files
  - **Statew** Provide information about writable CMS files

- ***CMS Libraries***

  - **Members** Extract members from a CMS simulated partitioned dataset
  - **PDSdirect** Write directory information from a CMS simulated partitioned dataset

- ***GCS Files***

  - **Adtfst** Write information about files

- ***REXX***

  - **Rexxvars** Retrieve variables from a REXX program
  - **Stem** Retrieve or set variables in a REXX or EXEC2 environment
  - **Var** Retrieve or set a variable in a REXX or EXEC2 environment
  - **Varload** Set variables in a REXX or EXEC2 environment

- ***Terminal***

  - **Console** Read or write the terminal in line mode
  - **Fullscr** Full-screen 3270 write and read to the console or attached screen

- ***Unit Record***

  - **Printmc** Print lines
  - **Punch** Punch cards
  - **Reader** Read from a virtual card reader
  - **Uro** Write unit record output

- ***XEDIT***

  — **XEDIT** Read or write an Xedit in-storage file
  — **Xmsg** Issue XEDIT messages

- ***Other Device Drivers***

  — **Delay** Suspend stream
  — **Emsg** Issue messages
  — **Hole** Destroy data
  — **Immcmd** Write the argument string from immediate commands
  — **ISPF** Access ISPF tables
  — **Literal** Write the argument string
  — **Mdiskblk** Read blocks from CMS minidisk
  — **QSAM** Read or write sequential datasets through a DCB
  — **SQL** Interface to SQL
  — **Sqlcodes** Write the last 11 SQL codes received
  — **Stack** Read or write the program stack
  — **Storage** Read or write virtual machine storage
  — **Tape** Read or write tapes
  — **VMC** Write VMCF reply
  — **XAB** Read or write External Attribute Buffers (XABs)

Filters

- ***Assembler Files***

  — **Asmcont** Join multiline Assembler statements
  — **Asmxpnd** Expand joined Assembler statements

- ***Buffering***

  — **Buffer** Buffer records
  — **Instore** Load the file into an in-storage buffer
  — **Outstore** Unload a file from an in-storage buffer
  — **Sort** Order records

- ***Cut and Paste***

  — **Chop** Truncate the record
  — **Join** Join lines
  — **Pad** Expand records
  — **Split** Split records relative to a target
  — **Strip** Remove leading or trailing characters

- ***Format Conversion***

  — **Block** Block to an external format
  — **Buildscr** Build a 3270 data stream
  — **Deblock** Deblock external data formats
  — **Fblock** Block data, spanning input records

- **Iebcopy** Process IEBCOPY data format
- **Pack** Pack records as done by XEDIT and COPYFILE
- **Unpack** Unpack a packed file

• ***Printer Files***

- **Asatomc** Convert ASA carriage control to CCW operation codes
- **C14to38** Combine overstruck characters to single code point
- **Mctoasa** Convert CCW operation codes to ASA carriage control
- **Optcdj** Generate table reference character (TRC)
- **Overstr** Process overstruck lines
- **Xpndhi** Expand highlighting to space between words

• ***Record Format***

- **Change** Substitute contents of records
- **Specs** Rearrange contents of records
- **Xlate** Transliterate contents of records

• ***Selection Filters***

- **Asmfind** Select statements from an Assembler file as XEDIT FIND
- **AsmNfind** Select statements from an Assembler file as XEDIT NFIND
- **Between** Select records between labels
- **Drop** Discard records from the beginning or the end of the file
- **Find** Select lines as XEDIT FIND
- **Frlabel** Select records from the first one with leading string
- **Inside** Select records between labels
- **Locate** Select lines with a string
- **NFind** Select lines by XEDIT NFIND logic
- **Nlocate** Select lines without a string
- **Notinside** Select records not between labels
- **Outside** Select records not between labels
- **Take** Select records from the beginning or end of the file
- **Tolabel** Select records to the first one with leading string
- **Unique** Discard or retain duplicate lines
- **Whilelab** Select run of records with leading string

• ***Other Filters***

- **Count** Count lines, blank-delimited words, and bytes
- **Duplicate** Copy records
- **Fmtfst** Format a File Status Table (FST) entry
- **Tokenise** Tokenise records
- **Vchar** Recode characters to different length

Control Stages

- **Append** Put output from a device driver after data on the primary input
- **Pipcmd** Issue pipeline commands
- **Preface** Put output from a device driver before data on the primary input stream
- **Runpipe** Issue pipelines, intercepting messages

Functions

- **Evaluate** Issue result of a REXX function as pipeline command

Gateways

- **Collate** Collate streams
- **Fanin** Catenate streams
- **Faninany** Copy records from whichever input stream has one
- **Fanout** Copy records from the primary input stream to all output streams
- **Lookup** Find references in a directory using a key field
- **Maclib** Generate a macro library from stacked members in a COPY file
- **Merge** Merge streams
- **Overlay** Overlay data from input streams
- **Synchronise** Synchronise records on multiple streams
- **Update** Apply an update file

Host Command Processors

- **CMS** Issue CMS commands, write response to pipeline
- **Command** Issue CMS commands, write response to pipeline
- **CP** Issue CP commands, write response to pipeline
- **Starmsg** Write lines from a CP system service
- **Subcom** Issue commands to a subcommand environment

Look-up Routines

- **Ldrtbls** Resolve an entry point from the CMS loader tables
- **Nucext** Call a nucleus extension
- **Rexx** Run a REXX program to process data
- **Runat** Call a routine whose address is provided as a parameter

Service Programs

- **Help** Display help for *CMS Pipelines*
- **Query** Query *CMS Pipelines*