PLUNGING ON

Apprentice Plumbing

Melinda Varian

Office of Computing and Information Technology
Princeton University
87 Prospect Avenue
Princeton, NJ 08544 USA
—.—
Email: maint@pucc.princeton.edu
Web: http://pucc.princeton.edu/~melinda
Telephone: 1-609-258-6016

VM Academy '95
La Hulpe, Belgium
November, 1995

I. INTRODUCTION

With the shipment of VM/ESA Release 1.1, *CMS Pipelines* at last became an official part of CMS.[1] Now that VM finally has indoor plumbing, all CMS programmers can benefit from the tremendous productivity gains provided by *CMS Pipelines*.

In this paper, I will be assuming that you are familiar with the basic constructs of *CMS Pipelines*, such as simple straight pipelines and simple REXX filters and subroutine pipelines. I am hoping to take you on to the next stage in your apprenticeship as a "plumber", to introduce you to some very powerful concepts and techniques that were not covered in *Plunging into Pipes*.

II. THE APPEND AND PREFACE CONTROL STAGES

The first new concept I want to introduce is an easy one and very useful. As you will recall, input device drivers (stages that read data from a host interface into the pipeline) must be the first stage in a pipeline. Output device drivers can be in any position other than the first. And some stages are input device drivers if they are a first stage, but output device drivers elsewhere.

As you can imagine, however, there are cases in which it would be useful to read data into a pipeline from more than one source, which means that you need a pipeline with more than one input device driver. Suppose, for example, that you want to process the contents of two CMS files in one pipeline. To handle such cases, *CMS Pipelines* provides two "control stages" called append and preface:

---

[1] It is still available as a PRPQ for those of us who have not yet migrated to ESA 1.1 or later.

```
'PIPE',
   '< file one |',                 /* Read FILE ONE.          */
   'append < file two |',          /* Put FILE TWO after it.  */
   'xlate 1-* e2a |',              /* Convert them to ASCII.  */
   '> one two a'                   /* Write out as one file.  */

'PIPE',
   '< file one |',                 /* Read FILE ONE.          */
   'preface < file two |',         /* Put FILE TWO before it. */
   'xlate 1-* e2a |',              /* Convert them to ASCII.  */
   '> two one a'                   /* Write out as one file.  */
```

What these stages do is attach another stage to the pipeline and allow it to behave as a first stage. The argument you specify with append or preface is the stage to be attached to the pipeline. The difference between these two control stages is that append puts the data from its device driver *after* the records that were already in the pipeline, while preface puts the data from its device driver *before* the records that were already in the pipeline. In the two examples here, the first < stage reads FILE ONE into the pipeline. Then the control stage runs an additional < stage to read FILE TWO into the pipeline, after which all the records are translated from EBCDIC to ASCII and written out as a single file. In the append example, the records from FILE TWO are at the end of the output file, while in the preface example, they are at the beginning of the output file. You can use as many append and preface stages as you need, to attach as many input device drivers as you need.

Another common use of append is to force the record created by a literal stage to be written at the end of the data going through the pipeline. literal can occur in any position in a pipeline, but (wherever it occurs) it puts the record it creates ahead of the records already in the pipeline. However, by appending a literal stage, you can add a record to the *end* of the data stream:

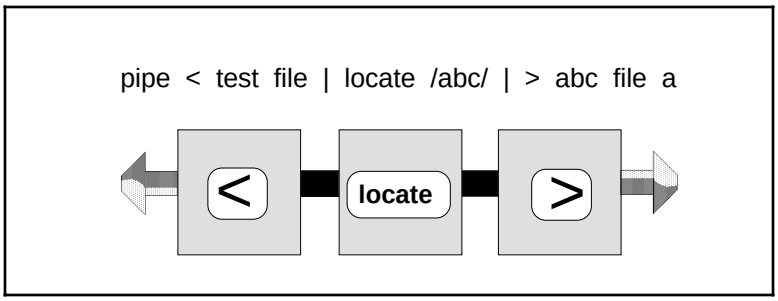
```
'PIPE',
   '<' fn ft fm '|',               /* Read a file.            */
   'literal * TOF *|',             /* Prepend a TOF marker.   */
   'append literal * EOF *|',      /* Append an EOF marker.   */
   '> output file a'               /* Write it out.           */
```

So, this example inserts a “* TOF *” record and an “* EOF *” record at the beginning and end of the file, respectively.

III. MULTI-STREAM PIPELINES

Now let us turn to the concept I would most like to help you master, multi-stream pipelines. You may well find that the learning curve gets a bit steep here for a while, but once you have mastered this concept your pipelines will be able to branch, which will enormously expand their possibilities.

Let's start with a simple straight pipeline:

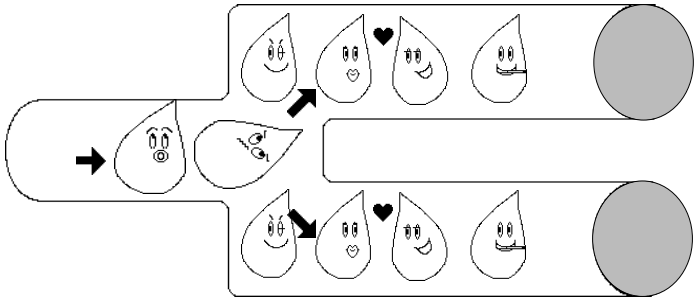


Data flow from the left to the right through this pipeline, going from the output stream of one stage to the input stream of the next stage. Like every other stage in every other pipeline, the three stages in this pipeline each have an input stream and an output stream defined. These are known as their "primary" input and output streams. Every stage has primary streams defined for it automatically when it is first dispatched. Note, however, that although the input stream for the first stage and the output stream for the last stage are *defined*, they are not *connected*. They are at the ends of the pipeline. (In the diagram above, you will see that the streams that are defined but not connected are portrayed as pipes with caps on the end. This convention will be used throughout.)

The pipeline shown above reads a file, selects only the records that contain the string "abc", and writes those records to a disk file. But what about the other records, the ones that the locate stage discards because they do not contain "abc"? What if you needed to process those records, too? Suppose you wanted to put them through another simple pipeline, such as this one:

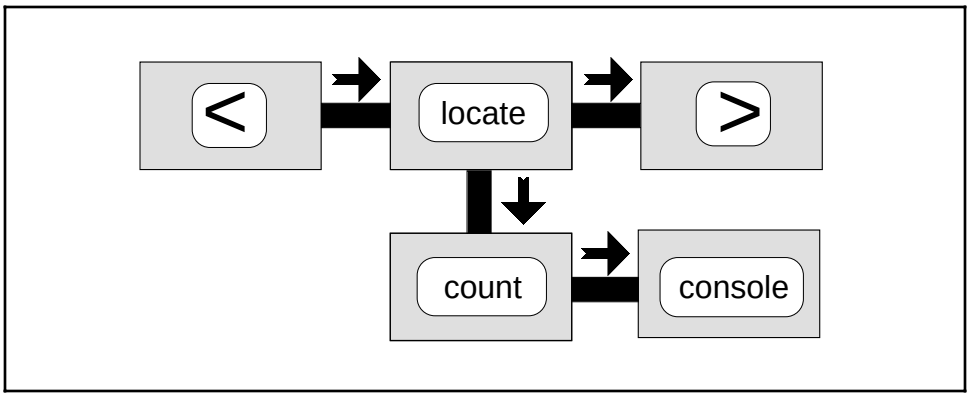
```
. . . | count lines | console
```

You can do that if you build a multi-stream pipeline (that is, if you put a branch into your pipeline). A branching pipeline can be visualized quite readily with real plumbing:



Water flows into a house through a single water pipe, but the flow is soon split into two streams, to deliver water to two rooms in the house.

In our example of a data pipeline, data flow in through a single stream, but the flow is then split into two streams, which are processed differently:



Conceptually, this is very straightforward. < reads the file; locate selects the records that contain "abc" and writes them on its primary output stream, just as before. And, just as before, > receives those records on its input stream and writes them to a file. But, now, instead of discarding the records that do not contain "abc", locate writes them on another output stream, its "secondary" output stream. This output stream is connected to the count stage, so count reads those records, counts them, and writes a record containing the count on its output stream. console receives that record and displays it on the terminal.

What we have built here is a multi-dimensional structure, so the question soon arises of how to portray that in a CMS command, which is inherently one-dimensional. The approach adopted in *CMS Pipelines* is to recognize that even the most complexly branching pipeline is composed of a number of straight pipeline segments connected to one another in some fashion. Thus, a multi-stream pipeline is coded in the argument to a PIPE command as a series of straight pipeline segments separated from one another by pipeline end characters:

```
PIPE (endchar ?) segment-1 ? segment-2 ? . . . ? segment-n
```

Or, in portrait format:

```
PIPE (endchar ?)
   segment-1
?
   segment-2
?
   . . .
?
   segment-n
```

The character to be used as the pipeline end character is specified as an option on the PIPE command. The individual pipeline segments may still be composed of many stages, and those are separated from one another by stage separator characters, just as before.

So, our example would be represented as two pipelines in the specification of a single PIPE command:

```
PIPE (endchar ?)
   < test file | locate /abc/ | > abc file a
?
   count lines | console
```

The first pipeline would consist of the <, locate, and > stages, and the second pipeline would consist of the count and console stages, with the pipeline end character between the two pipelines.

That allows all of the stages to be specified in a single one-dimensional argument string, but the problem remains of how to portray the connections between the pipeline segments, of how to show the place where the pipeline branches. That is done using labels.

CMS Pipelines labels look like REXX labels, a word followed by a colon. For a stage to have the ability to cause a branch in a pipeline, it must have a label. So, our locate stage would have a label:

. . . | loc: locate /abc/ | . . .

The label loc: on this locate stage defines a place where the pipeline can branch. It says that this stage can have more than one stream flowing out of it (or into it).

Then, to show where the second stream branches *to*, the label is used again, as a stage all by itself:

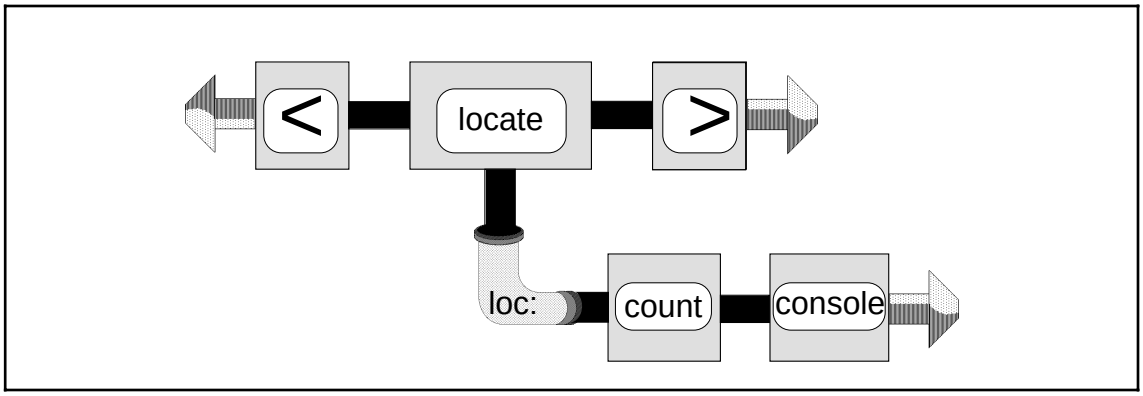
. . . ? loc: | . . .

at the beginning of the pipeline segment that the secondary stream flows into:

. . . ? loc: | count lines | . . .

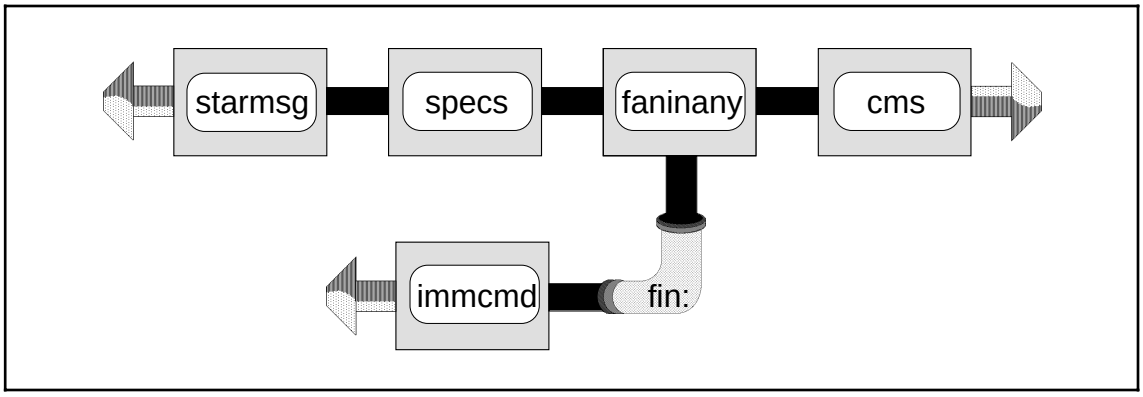
Or, to put it all together (in portrait format):

```
'PIPE (endchar ?)',          /* Declare end character.     */
   '     < test file a |',   /* Read file.                 */
   'loc: locate /abc/ |',    /* Select records with "abc". */
   '     > abc file a',      /* Write them to a file.      */
'?',                         /* End of first pipeline.     */
   'loc: |',                 /* Non-abc records to here.   */
   '     count lines |',     /* Count them.                */
   '     console'            /* Display count.             */
```



Records flow into this locate stage from the < stage that precedes it. Records can flow out of this locate stage in *two* directions. The records containing "abc" flow out on locate's primary output stream to the > stage. The records that do not contain "abc" flow out on locate's secondary output stream, which is defined by the second occurrence of its label, loc:, and which is connected to the primary input stream of the count stage.

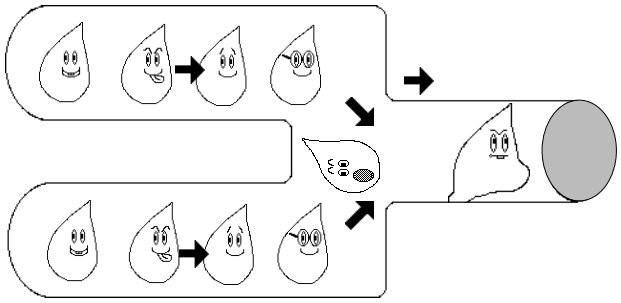
Exactly the same notation is used for a stage that can have more than one *input* stream. faninany is such a stage. The job of faninany is simply to read input from any input stream that has a record and to write all the records it gets to its primary output stream; thus, it is used for collecting records from two or more sources. This diagram shows a multi-stream pipeline that uses faninany:



The starmsg stage here communicates with the CP *MSG system service and routes any messages received via that service into the pipeline, where a specs stage is used to remove the message header from each record. The immcmd stage accepts immediate commands from the virtual machine console and routes them into the pipeline. faninany receives records written by either specs or immcmd and sends them on to the next stage, cms, which executes them as CMS commands.

I should point out here that starmsg and immcmd are both the first stage in a pipeline. Both have unconnected primary input streams, as indicated by the capped pipes on their input side. Thus, this is another way to build a pipeline with more than one input device driver.

Again, this sort of structure is easily visualized in terms of real plumbing, *e.g.*, the drains from two bathtubs flowing into a single line:



But, again, how is this represented in the one-dimensional argument string for a PIPE command? It is done exactly the same way as before. The stage that has multiple input streams must have a label to mark the site of the branch in the pipeline, the site at which one pipeline segment can be connected to another:

. . . | fin: faninany | . . .

As before, the pipeline specification is divided into two pipeline segments separated by a pipeline end character. And the flow of data between the two pipelines is specified by using a label. In this case, the second occurrence of the label defines the labelled stage's secondary *input* stream:

```
'PIPE (endchar ?)',
   '     starmsg |',           /* Listen for messages.    */
   '     specs 17-* 1 |',      /* Delete message header.  */
   'fin: faninany |',          /* Collect all input.      */
   '     cms',                 /* Issue as commands.      */
'?',                           /* End of first pipeline.  */
   '     immcmd CMD |',        /* Listen to console, too. */
   'fin: '                     /* Send to faninany.       */
```

Records flow into faninany on its primary input stream from the specs stage, and records also flow into faninany on its secondary input stream, which is defined by the second occurrence in the pipeline specification of its label, fin:. This second occurrence, must, as before, be in a stage that contains only a label. The second occurrence of the label fin: connects the output stream of the immcmd stage to the secondary input stream of the faninany stage. When a command starting with the letters "CMD" is typed on the virtual machine console, it is captured by the immcmd stage. immcmd then writes the command to its output stream, which flows through the label in the following stage to faninany's secondary input stream. You will note, again, that immcmd is the first stage in a pipeline; it appears immediately following a pipeline end character.

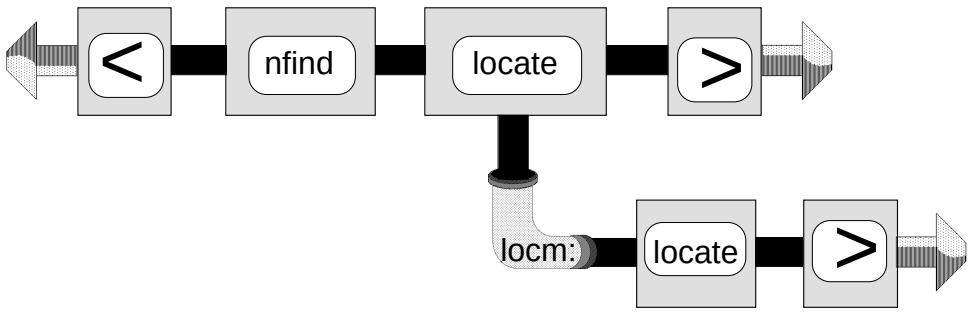
It takes a while for most people really to become accustomed to this notation, but I think you will agree that the underlying concept of a multi-stream pipeline is easily visualized. You may find it easier to accept the difficulty of the notation if you consider the problem of portraying *n* dimensions in 1.

As a science fiction fan, I find it easiest to envision the two separate pipelines in the specification of this PIPE command as two separate universes. Then the label fin: becomes a "space-time junction box", a "wormhole in space" that exists in both universes and connects them to one another. fin: behaves like the dual white/black hole of science fiction; that is, records disappear from one universe by being sucked into the black hole called fin:, and they appear in the other universe by being spewed out of the white hole called fin:.

So, to review: the label fin: on the faninany stage says that this stage can cause a branch in the pipeline; that is, it may have more than one input or output stream (or both). Whether it actually does have a secondary stream depends on whether its label fin: occurs again further on in the pipeline specification. If the label fin: does occur again, the second occurrence is in a stage all by itself (and is known as a "label reference"). Records from the stage preceding the label reference flow through the label into the faninany stage as its secondary input stream. (Another way to say this is that the primary output stream of the stage preceding the label reference, immcmd, is connected to the secondary input stream of the labelled stage, faninany.)

Now, to convince you that this is all worthwhile, let's look at what is frequently the first use people find for multi-stream pipelines, simple Boolean operations, such as selecting all the records that contain one string *or* another string. This example shows a pipeline that selects the MDISK and LINK cards from a CP directory, discarding all other records:

```
'PIPE (endchar ?)',              /* Declare pipeline endchar. */
   '      < user direct |',      /* Read directory.           */
   '      nfind *|',             /* Discard comments.         */
   'locm: locate /MDISK/ |',     /* Select MDISK cards.       */
   '      > mdisk file a',       /* Write them to a file.     */
'?',                             /* End of first pipeline.    */
   'locm: |',                    /* Non-MDISK cards to here.  */
   '      locate /LINK/ |',      /* Select LINK cards.        */
   '      > links file a'        /* Write them to a file.     */
```

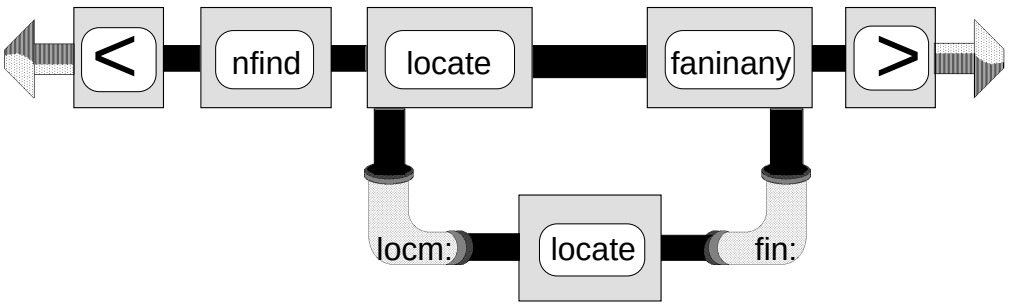


The nfind * stage discards all comments cards, and then records that contain the string "MDISK" are selected by the first locate stage. They flow directly from locate's primary output to the input of the first > stage, which writes them to MDISK FILE. The label locm: on the first locate stage is referenced in the second pipeline segment, so the records that do not contain the string "MDISK" are written to the secondary output stream of the first locate stage, which is connected to the primary input stream of the second locate stage, via the label reference, locm:.

The second locate stage has no label, so it has no secondary streams. Therefore, it discards any records that it receives that do not contain the string “LINK”. It selects any records that do contain that string and writes them to its primary output stream, which is connected to the input of the second > stage, which writes those records to LINKS FILE.

It may be, however, that what you really want to do is to build one file that contains both the MDISK cards and the LINK cards and that you want them to be in the same order in that output file as they were in the input file. To achieve that, you need to solder another branch into your pipeline:

```
'PIPE (endchar ?)',              /* Declare pipeline endchar. */
   '      < user direct |',      /* Read directory file.      */
   '      nfind *|',             /* Discard comments.         */
   'locm: locate /MDISK/ |',     /* Select MDISK cards.       */
   'fin:  faninany |',           /* Collect MDISKs and LINKs. */
   '      > output file a',      /* Write them to a file.     */
'?',                             /* End of first pipeline.    */
   'locm: |',                    /* Non-MDISK cards to here.  */
   '      locate /LINK/ |',      /* Select LINK cards.        */
   'fin:'                        /* Send them to faninany.    */
```



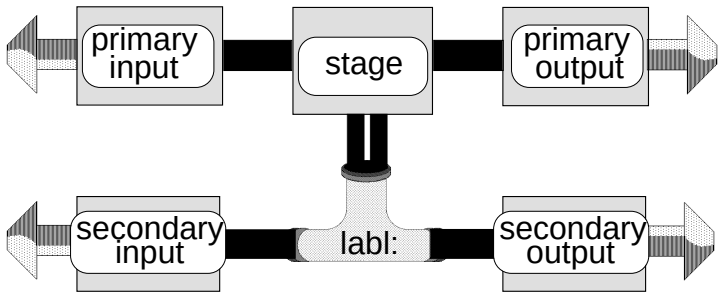
In this case, the records that the first locate stage selects flow directly into the primary input of the faninany stage. The records that the second locate stage selects flow via the reference to the label fin: into the secondary input of the faninany stage. So, all the records that contain “MDISK” or “LINK” (or both) flow through the faninany stage to the stage that follows it, and (such is the magic of the pipeline dispatcher) they all arrive at the end of the pipeline in the same order they entered it.

Many of the stages built into *CMS Pipelines* use more than one input or output stream. All of the selection filters, such as find, locate, tolabel, between, *etc.*, can use a secondary output stream. If they have a secondary output stream, then instead of discarding the records that do not meet their selection criteria, they write them to their secondary output.

Some of the most useful *CMS Pipelines* stages, such as lookup and update, require multiple streams. In fact, these two (and several others) may have both multiple input streams and multiple output streams. The notation for that is a simple combination of what we have seen so

far. If a stage has a secondary input stream and a secondary output stream, both flow through the second occurrence of the label for that stage:

```
'PIPE (endchar ?)',
   '      < input one |',
   'labl: stage |',              /* Labelled stage.        */
   '      > output one a',
'?',                             /* End of first pipeline. */
   '      < input two |',
   'labl: |',                    /* Label reference.       */
   '      > output two a'
```



Like any other stage, this one (“stage”) has a primary input stream and a primary output stream defined. As always, these streams are connected through the stage separators immediately before and after the stage. But because its label is referenced further on in the pipeline specification, this stage also has secondary input and output streams defined. And, in this case, both of those streams are actually connected, not just defined. Records flow from the second < stage through the label reference into the labelled stage’s secondary input stream. Records flow from the labelled stage’s secondary output stream through the label reference into the second > stage. Thus, records flow in both directions through this space-time junction box.

A point that may not be clear here is that the label reference in the second pipeline segment does *not* connect the stage before it to the stage after it. No data flow directly between those two stages. What the label reference does is connect both those stages to the labelled stage, one on its input side and the other on its output side. Thus, the label reference puts two “elbows” into the pipeline.

There is no limit to the number of streams a stage may have. The notation remains the same, no matter how many more streams are defined. A label reference always defines both an input stream and an output stream. (Neither or one or both of those streams may actually be connected.) And the stream number is determined by how many times that label has occurred previously in the pipeline specification. The third occurrence of a particular label in a pipeline specification defines the tertiary input and output streams for the stage with that label; the fourth occurrence defines the quaternary streams; and so forth.

There are commands you can use when writing a stage to make sure that you have no more streams connected than you are expecting. All *CMS Pipelines* built-in programs perform such checks.

IV. AUGMENTING XEDIT

Once people gain some fluency in *CMS Pipelines*, they find themselves using it to do things for which they formerly would have written XEDIT macros. Pipelines are usually simpler than the corresponding XEDIT macros, and they can be astonishingly faster. However, one still uses XEDIT for non-repetitive operations, so there are times when it becomes convenient to augment XEDIT with a pipeline. Suppose, for example, that you are editing a file that should be no more than 80 columns wide, but, in fact, has buried in it somewhere one record that has a length of 81. This simple command devised by Rob van der Heij can find that long record for you instantly:

```
:0  pipe  xedit | locate  81
```

Typing this command on the XEDIT command line will cause the current line pointer to be positioned immediately after that 81-character record. The xedit stage here reads from the current file into the pipeline. (Since an xedit stage always starts reading at the current line, preceding the PIPE command with a LOCATE :0 XEDIT subcommand assures that the file is read from the beginning.) locate, in the form used here, selects records that have any character in the specified column (even a blank), so this locate selects a record that extends into column 81.[2]

Now, let me show you a nifty trick I learned from Harry Williams, of Marist College. Suppose you have been developing a pipeline by typing the PIPE command on the CMS command line, iterating until you have it the way you want it. Having done that, you decide to save it in an EXEC file, but you know that if you rekey it you may get it wrong. Harry's trick is to invoke XEDIT to open the new EXEC file and then type on the XEDIT command line:

```
pipe  console | xedit
```

Next use your CP RETRIEVE key to summon up your successful PIPE command. When you have it on the command line, press ENTER. console will read the command from the command line into the pipeline. Press ENTER again; the console stage will terminate, because it has received a null input line. The xedit stage will write the retrieved PIPE command into your XEDIT session, so then all you need to do is add some quotes and comments.

As these two examples have shown, the *CMS Pipelines* xedit stage belongs to the class of device drivers that either read data into the pipeline or write data from the pipeline, depending on their position in the pipeline. When an xedit stage is first in the pipeline, it reads from the current XEDIT ring into the pipeline. When an xedit stage is anywhere other than at the beginning of a pipeline, it writes from the pipeline into a file in the current XEDIT ring.

---

[2] One might not understand at first why this pipeline terminates as soon as it finds the *first* record that extends into column 81, but the explanation is not difficult. A locate stage terminates "prematurely" when it discovers that its output is not connected, so this locate stage terminates as soon as it selects a record and tries to write it to its output stream. When it terminates, it returns to the pipeline dispatcher, which severs its connection to the xedit stage. Whenever an xedit stage resumes after having given up control, before it reads any further records from the file it checks to be sure that its output stream is still connected. In this case, when xedit regains control after having written the first 81-character record into the pipeline, locate will already have terminated, so xedit will see that its output is no longer connected and will terminate also, without having read any more records from the XEDIT session. Thus, the XEDIT current line pointer is left positioned immediately after the first record that met locate's selection criteria.

Whether reading or writing, one must already be in XEDIT in order to issue a PIPE command containing an xedit stage, and the desired file must be in the current ring and must be positioned at the appropriate line within the file. When an xedit stage is used to write from a pipeline into an XEDIT session, if the file is RECFM F, the records from the pipeline must be the correct length.

Where in the file the records go depends on where in the file you are positioned. If the current line is at the bottom of the file, then the records from the pipeline are appended to the end of the file. For example, if you type this command on the XEDIT command line (in a RECFM V file):

```
:*  pipe cms query cmslevel | xedit
```

the LOCATE :* XEDIT subcommand positions to the end of the current file, and then the PIPE command uses a cms stage to issue a QUERY CMSLEVEL command and feed the response into the pipeline, where the xedit stage receives it and appends it to the file.

If the file is positioned other than at the end, then existing records are replaced by records from the pipeline. As each record is written from the pipeline into the XEDIT session, the current line pointer is advanced by one. The next record from the pipeline replaces the next record in the file.

An xedit stage cannot be used to insert records between existing records. For that, you can generally use a subcom xedit stage to send INPUT commands to XEDIT. Thus, if you wished to insert that QUERY CMSLEVEL response between existing records, you would position to the correct line and issue this command:

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

The change stage converts each record into an XEDIT INPUT subcommand, and subcom xedit sends those commands to XEDIT to execute, thereby inserting the records into the file following the current line pointer.

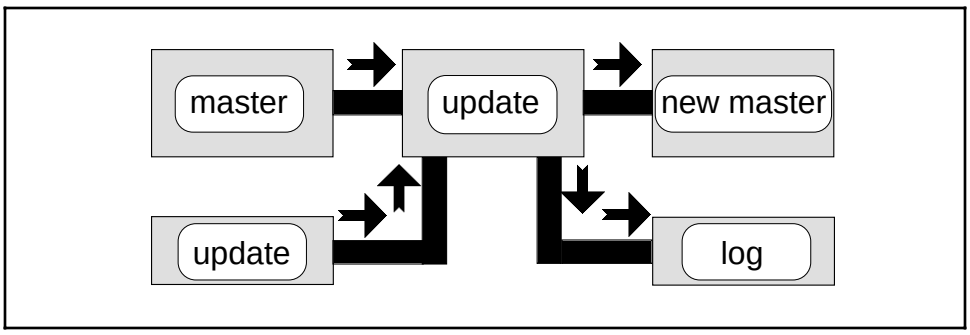
Using xedit and subcom xedit stages in the same pipeline can be an especially powerful technique. This example from Chuck Boeheim numbers the “End-of-file” lines in a TAPE MAP file, to make it easier to know how many tape forward-space-file operations are needed:

```
/* MAPNUMB XEDIT:  Number "End-of-file" lines in a tape map.  */

'ALL /End-of-file/'            /* Select the EOF lines.       */
Address Command 'PIPE',
   'xedit |',                  /* Read the EOF lines.         */
   'strip |',                  /* Strip away blanks.          */
   'specs',                    /* Append a sequence number.   */
   '   1-* 1',
   '   number nextword |',
   'change //+1REPLACE / |',   /* Make into REPLACE commands. */
   'literal TOP|',             /* Start with a TOP command.   */
   'buffer |',                 /* Hold until end-of-file.     */
   'subcom xedit'              /* Send commands to XEDIT.     */
'ALL'
```

Although one could do the same thing with a pipeline outside XEDIT,[3] this is a nice example of augmenting XEDIT with *Pipes*. A specs stage is used to append a record number to each line read from the file, and then the change stage transforms each of these modified lines into an XEDIT REPLACE subcommand. subcom xedit sends the TOP subcommand generated by the literal stage to XEDIT, followed by the REPLACE subcommands. (The buffer stage holds all the commands until the earlier stages have processed the entire file, to prevent the subcom xedit stage from changing the file while it is still being read by the xedit stage.)

There is much more to learn about using *CMS Pipelines* with XEDIT. The *CMS Pipelines Tutorial* (GG66-3158) is quite good in this area, so I recommend it to you for further guidance. However, to reinforce the earlier lesson in multi-stream pipelines, let me discuss one more example of using *CMS Pipelines* in an XEDIT macro.

CMS Pipelines provides an update stage that does what the CMS UPDATE command does; it applies an update to a master file using the same control statements that UPDATE and XEDIT use for applying updates. The update stage requires two input streams and two output streams:



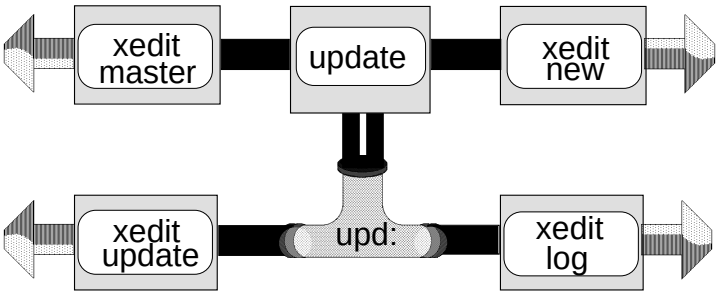
[3] The equivalent pipeline uses multiple streams and illustrates an idiom that is frequently needed, *i.e.*, splitting the input, processing part of it, and then recombining the streams:

```
/* MAPNUMB EXEC: Number the "End-of-file" lines in a tape map */

'PIPE (endchar ?)',
   '     < tape map a |',         /* Read the tape map.        */
   'loc: locate /End-of-file/|', /* Select EOF lines.         */
   '     strip |',               /* Strip away blanks.        */
   '     specs',                 /* Append a sequence number. */
   '        1-* 1',
   '        number nextword |',
   'fan: faninany |',            /* Collect all records.      */
   '     > tape map a',          /* Rewrite the tape map.     */
'?',                             /* End of first pipeline.    */
   'loc: |',                     /* Non-EOF lines come here.  */
   'fan:'                        /* Send them to faninany.    */
```

update reads the master file from its primary input stream and the update file from its secondary input stream. It writes the updated master file to its primary output stream and the update log to its secondary output stream.

This fragment is from an XEDIT macro that we use at Princeton when fitting our modifications onto a new level of the system:

```
'PIPE (endchar ?)',
   '     xedit' fn ft fm '|',      /* Read master file.     */
   'upd: update |',
   '     xedit' fn 'new a',        /* Write updated master. */
'?',
   '     xedit' fn updtft fm '|',  /* Read update file.     */
   'upd: |',
   '     xedit update log a'       /* Write update log.     */

updrc = RC                         /* Update's return code. */
```



We use this macro to pipe the next update onto the current file in the XEDIT ring, as we refit our modifications to a particular module, one-by-one. Earlier in the macro, there is code to figure out what the next update is and to bring the update file into the XEDIT ring and position it at the top of the file. There is also code to position the master file at its top and to create empty files to contain the new master file and the update log.

The four xedit stages in this pipeline specification feed the two required inputs into the update stage and accept the two required outputs from the update stage. The two xedit stages that are at the beginning of pipeline segments are input device drivers. (These are the first and third xedit stages in the pipeline specification. They correspond to the two stages with capped input streams in the diagram.) These two stages read files into the pipeline. So, the master file flows into the update stage on its primary input stream, and the update file flows in on update's secondary input stream, which is connected to the output of the stage preceding the second occurrence of the label upd:. The two xedit stages that are not at the beginning of pipeline segments are output device drivers; they write records from the pipeline into files in the XEDIT ring. Thus, the updated master file flows out on update's primary output stream and is written to a temporary file. The update log flows out on update's secondary output stream, which is connected to the input of the stage following the second occurrence of the label upd:.

Following this fragment, the macro takes actions based on the return code set by this PIPE command. If the update was successful, the return code is zero; the old master is deleted from the ring and the new master is renamed and made the current file in the XEDIT ring. If the update was not successful, the return code from PIPE is not zero; the macro discards the new master and makes the update log the current file, so that the problem can be analyzed.

V. PEEKTO

Let's review briefly the process of writing pipeline filters in REXX. In *Plunging into Pipes*, I suggested that you use this simple REXX filter as a basis for building your own filters:

```
/* 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.   */
```

When this little filter, null, is named as a stage in a pipeline, it simply loops, reading each record from the pipeline and writing it back to the pipeline unchanged, continuing until it encounters end-of-file.

However, that was a simplification that left out a concept that becomes important once you have begun writing multi-stream pipelines. In fact, this is the model to use in writing REXX filters that behave well when invoked in the multi-stream portion of a pipeline:

```
/* NULL REXX:                      Dummy pipeline filter  */
Signal On Error

Do Forever                            /* Do until EOF.    */
   'PEEKTO record'                    /* Examine input.   */
   'OUTPUT' record                    /* Write it to pipe.*/
   'READTO'                           /* Consume input.   */
End

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

When this null filter is named as a stage in a pipeline, it, too, simply copies its input records to its output unchanged, but the relative timing may be different. This null first examines each record

with a peekto command, which reads the record into the specified REXX variable but does not remove it from the input stream. Only after it has written its own output record, does this null issue a readto to "consume" the input record. (readto without an argument just discards the input record; this is less expensive than reading it into a REXX variable.)

Pipeline stages that behave this way are said not to "delay the record". If all of the stages in the multi-stream portion of a pipeline follow this protocol, then the records that flow through that pipeline will arrive at the end of the multi-stream portion in the same relative order they entered it. That is, a record will not "overtake" an earlier record by taking a different path through the pipeline.

The mechanism for achieving this desirable result is very simple. When a stage issues an output command, it becomes "blocked"; it does not regain control until its output record has been consumed. If a stage issues an output command to write Record 1 to, say, its primary output, it won't get a chance to write Record 2 to a different output until the stage reading from its primary output has consumed Record 1. If that stage doesn't consume Record 1 until after it has written Record 1 to *its* output, and if all of the other stages in the multi-stream portion of the pipeline behave in the same way, then Record 1 gets safely through the pipeline before Record 2 begins its journey, so there is no chance for Record 2 to overtake Record 1 by following a different path.

Thus, the reverse filter discussed earlier is better written like this:

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

Do Forever                              /* Do until EOF.    */
   'PEEKTO record'                      /* Examine input.   */
   'OUTPUT' Reverse(record)             /* Write it to pipe.*/
   'READTO'                             /* Consume input.   */
End

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

VI. DYNAMIC RECONFIGURATION OF PIPELINE TOPOLOGY

The last topic I want to introduce is dynamic reconfiguration of pipeline topology. What this means is that your pipelines can be written so that they change their shape depending on the data that flow through them, growing new stages here and there as required. Even the easiest ways of doing this can markedly simplify the logic of your pipelines; and the uses that "master plumbers" make of dynamic reconfiguration are very powerful (as well as truly mind-bending).

I will be touching only lightly on this topic today, but I hope that this will be enough to get you started doing apprentice-level dynamic reconfiguration. There is an appendix that discusses this topic further.

Variable Stages

The first technique for doing dynamic reconfiguration is simply to use a REXX variable for a whole stage in your pipeline. Before executing the PIPE command, you "calculate" the value of the variable based on prevailing conditions. This is so easy to do that it is often overlooked, but it can be quite effective.

To illustrate, let's start with a simple, non-dynamic pipeline:

```
/* 3AM EXEC:   Run DAWN program at 3 a.m. tomorrow  */

'PIPE',
   'literal 27|',           /* Specify 27 o'clock.  */
   'delay |',               /* Wait until then.     */
   'specs /dawn/ 1 |',      /* Reformat to command. */
   'cms'                    /* Run DAWN at 3 a.m.   */
```

This is an EXEC that will wake up at 3 a.m. tomorrow and run a CMS program called DAWN. The literal stage inserts a record containing "27" into the pipeline. The delay stage reads that record and interprets it as an instruction to wait until "27 o'clock", *i.e.*, 3 a.m. tomorrow. When that time arrives, delay wakes up and writes the record that says "27" to its output stream. specs reads that record and converts it into a record that says "dawn".[4] The cms stage reads that record and executes it as a CMS command.

That is all very well, but if you were writing this EXEC to run in a service machine, you might want it to call DAWN every morning. That is easily arranged:

```
/* 3AM EXEC:   Run DAWN program at 3 a.m. daily.    */

'PIPE',
   'literal 27|',           /* Specify 27 o'clock.  */
   'duplicate * |',         /* Forever.             */
   'delay |',               /* Wait until then.     */
   'specs /dawn/ 1 |',      /* Reformat to command. */
   'cms'                    /* Run DAWN at 3 a.m.   */
```

By inserting a duplicate * stage between literal and delay, you provide delay with an endless supply of records that say "27". Each time delay wakes up, it writes a record to its output stream and then reads the next record from its input stream, which tells it again to wait until 3 a.m. tomorrow. So, it keeps running forever, waking up once each morning to produce one output

---

[4] Note that it is possible for a record written by a specs stage to contain none of the fields from the input record.

record, which subsequently gets converted into a command and executed. (In case you are wondering why the duplicate * stage does not flood virtual memory with records that say "27", this is because it cannot write another record until the previous one has been read by delay.)

One problem remains, however. If the system crashes and this service machine is reinitialized between midnight and 3 a.m., DAWN will not get called at 3 a.m. today. You might try building logic into the pipeline itself to address this problem, but the simplest way is to let REXX do the work for you:

```
/* 3AM EXEC:   Run DAWN program at 3 a.m. daily     */
/*             (including today, if necessary.)     */

If time('Hours') > 2        /* Now 3am or later?    */
   Then today=''
   Else today='literal 3|'  /* No, wait till then.  */

'PIPE',
   'literal 27|',           /* 3am tomorrow         */
   'duplicate * |',         /* Forever              */
    today,                  /* Maybe 3am today?     */
   'delay |',               /* Wait                 */
   'specs /dawn/ 1 |',      /* Reformat to command. */
   'cms'                    /* Run DAWN at 3 a.m.   */
```

Before the PIPE command is executed, the value of the variable today is determined. If the hour is greater than 2, then today is a null string. There will be no stage between duplicate and delay, and the first invocation of DAWN will be tomorrow morning, just as before. However, if the hour is less than 3, the value of today is set to the string "literal 3|". With that stage in the pipeline immediately before the delay stage, then as soon as the pipeline starts up, delay will receive a record that says "3", so it will wake up once at 3 o'clock (today) and at 27 o'clock forever after that, as it begins receiving the records produced by the first two stages.

Again, this is a very easy technique, but one that you should keep in mind, because it can greatly simplify your pipelines.

Callpipe

A somewhat more complex technique for building dynamically reconfiguring pipelines is to use the callpipe pipeline command to add another pipeline to the running set. In *Plunging into Pipes*, I introduced subroutine pipelines, REXX stages that use a callpipe command, such as mysub here:

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

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

Exit RC
```

This callpipe command creates a subroutine pipeline and executes it. The *: connector at the beginning of the subroutine pipeline takes over the input stream for the stage that contains the callpipe command, and the connector at the other end takes over the output stream for that stage. So, records flow in from the stage's primary input stream through the beginning connector. They are processed by stages b, c, and d in the subroutine pipeline, and then they flow out through the ending connector onto the primary output stream of the mysub stage.

Used this way, callpipe is primarily a tool for hiding complexity by packaging a cascade of filters together as a single stage, but callpipe's general function is to invoke another pipeline and run it to completion, and that is where the dynamic reconfiguration gets serious.

The time has come to learn to use callpipe inside your own REXX filters, so that your filters can add pipelines at will. The reason for doing this is simply to avail yourself of *CMS Pipelines* function within your REXX filters. Say, for example, that within a filter you need to know how many files are in the reader:

```
'CALLPIPE cp query files | specs word 2 1 | change /NO/0/ | var filecnt'
```

We have seen little pipelines like this before. It just issues a CP command, parses the response, and stores a value in a REXX variable. Although in this particular case you could use a PIPE command, rather than a callpipe pipeline command, it is better to get into the habit of using callpipe to add pipelines in your filters, because you will need callpipe when you take the next small step and start connecting your added pipelines to the existing ones. For example, assume that in the previous case what you really wanted to do was to write a record containing the file count into the main pipeline. That is easily done, simply by changing the very end of your subroutine pipeline:

```
'CALLPIPE cp query files | specs word 2 1 | change /NO/0/ | *:'
```

Now, instead of storing the file count in a REXX variable, the added pipeline connects to the existing pipeline and writes a record containing the file count. The *: connector at the end of this subroutine pipeline connects its output to the output stream of the stage that issued this callpipe command, so the file count record is written into the main pipeline.

Here is a complete REXX filter that uses callpipe in this way:

```
/* READLIST REXX:     Send contents of files into the pipe   */

/* Input: filenames;  Output: contents of the files          */

Signal On Error

Do Forever                          /* Do until get EOF.      */

   'PEEKTO record'                  /* Examine next 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.*/

   'READTO'                         /* Consume input record.  */
End

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

readlist does essentially the same thing as the getfiles stage. Its input is a list of files; its output is the *contents* of those files. As readlist reads each record from the pipeline, it parses it as a CMS file name and then uses callpipe to create a pipeline to read the records from the specified file.

This is a function you are likely to need frequently. For example, you may have a pipeline with a sequence like this:

pipe cms listfile outstat * * | readlist | find SYSTCPU | . . .

The cms stage would create a list of all your OUTSTAT files and would feed their names into the pipeline. readlist would then read each of those files and put the contents of the files into the pipeline for further processing.

In the callpipe example from earlier, there was a connector at both ends of the subroutine pipeline, because records were flowing in from the main pipeline and later flowing back out to the main pipeline. The subroutine pipeline in readlist is different. In this filter, it is the peekto command that is bringing records in from the pipeline. After those records are parsed, nothing further happens with them; they never go back to the pipeline. The callpipe command is *not* bringing records into the subroutine pipeline from the main pipeline. It is reading a CMS file and then sending the records from the file through its end connector into the main pipeline.

The subroutine pipeline established by the callpipe command runs until it has finished reading the specified CMS file. After all the records have been read and passed to the main pipeline, callpipe completes, and the Do Forever loop continues to iterate, allowing the peekto to pull another filename record in from the main pipeline, at which point callpipe will be invoked again to create another pipeline specifically to read that file into the pipeline.

Why use callpipe here? Because a pipeline is the fastest and easiest way to read a CMS file from within an EXEC! The same thing could have been done with EXECIO, I suppose, but probably not without looking up the command syntax (yet again).

readlist becomes even more useful when you enhance it to do something that getfiles does not (yet) do, allow callers to pass a filter to be applied to the files before they are put into the pipeline. For example, you might want to be able to tell readlist to put only the first record from each file into the main pipeline:

```
pipe  . . .  | readlist  take first 1 |  . . .
```

Here is how to make readlist handle that:

```
/* READLIST 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

Parse Arg filter                    /* Get any passed filter.  */
If filter <> ''                     /* If one, need additional */
   Then filter = filter'|'          /*    stage separator.     */

Do Forever                          /* Do until get EOF.       */

   'PEEKTO record'                  /* Examine next record.    */
   Parse Var record fn ft fm .      /* Break out file name.    */

   'CALLPIPE',                      /* Invoke pipeline.        */
      '<' fn ft fm '|',             /* Put file into stream.   */
       filter,                      /* Passed filter, if any.  */
      '*:'                          /* Connect into main pipe. */

   'READTO'                         /* Consume input record.   */
End

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

You will note that this is another case where an entire stage in a pipeline is a variable. filter, the variable stage here, becomes either the null string, if the caller passes no argument, or the stage the caller passes, such as take first 1.

We can take this example one step further and allow the caller to pass more than one filter to be applied to the files:

```
pipe  . . .  | readlist  take 10  !  drop 1 |  . . .
```

```
/* READLIST REXX:     Send contents of files into the pipe     */
/*                    after processing with the filter(s)      */
/*                    passed as an argument                    */

/* Input: filenames;  Output: (modified) contents of the files */

Signal On Error

Parse Arg filters                    /* Get any passed filters. */
If filters <> ''                     /* If any, need additional */
   Then filters = filters'!'         /*    stage separator.     */

Do Forever                           /* Do until get EOF.       */

   'PEEKTO record'                   /* Examine input record.   */
   Parse Var record fn ft fm .       /* Break out file name.    */

   'CALLPIPE (stagesep !)',          /* Invoke pipeline.        */
      '<' fn ft fm '!',              /* Put file into stream.   */
       filters,                      /* Passed filters, if any. */
      '*:'                           /* Connect into main pipe. */

   'READTO'                          /* Consume input record.   */
End

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

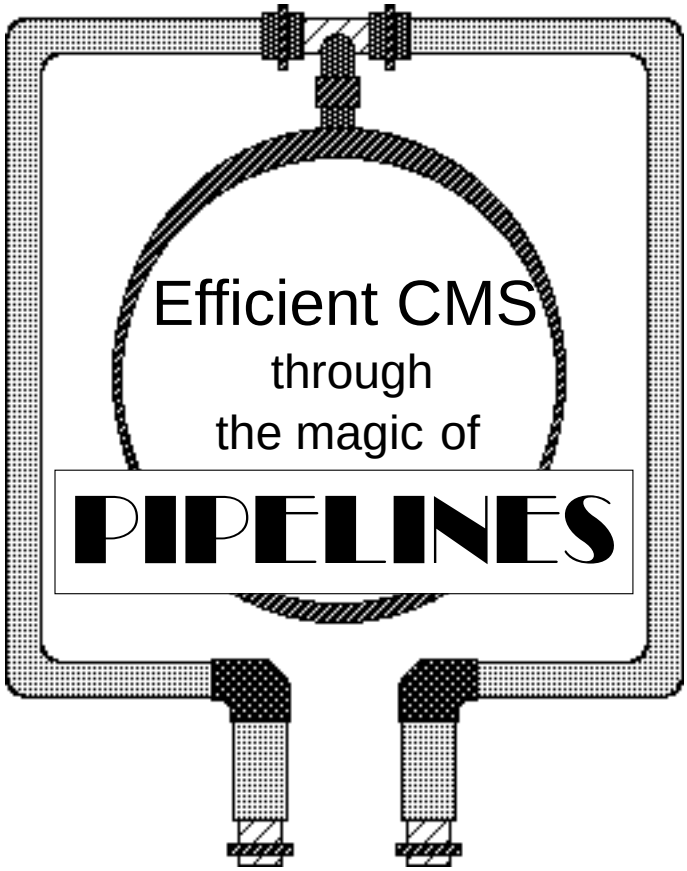
In this case, the callpipe uses the stagesep option to make its stage separator character an exclamation mark, rather than a vertical bar. The caller can then pass a string of as many stages as needed by using this same convention. Because an exclamation mark has no special significance in the caller's pipeline, the string "take 10 ! drop 1" will be passed to readlist as its argument string and substituted into the callpipe command, where the exclamation mark will be recognized as a stage separator.

Addpipe and Pipcmd

There are two other major tools used for building dynamically reconfiguring pipelines, the addpipe pipeline command and the pipcmd stage. Discussion of either of these is beyond the scope of this presentation, but they are both introduced in the Appendix. Briefly, addpipe is similar to callpipe, but the streams of the pipelines it adds can be connected to the existing pipelines in more varied ways, and the pipelines it adds run asynchronously, rather than synchronously. pipcmd is similar to other "host command processor" stages, but, in its case, the records it reads from its input stream are treated as pipeline commands to be issued. (Typically, these are callpipe commands.)

VII. CONCLUSION

There is still more to be learned about *CMS Pipelines*, but once you have absorbed the concepts introduced in this paper, you are past the hard parts and well on your way to earning your wrench.



Appendix A

ON PIPCMD, CALLPIPE, AND ADDPIPE

This appendix is a journeyman-level discourse on the tools used to build pipelines that can reconfigure themselves dynamically. I wrote it originally as an attempt to teach myself to use these tools and to explain to myself the delicious examples master plumbers were sending to me. I offer it in the hope that it may assist others as well.

Pipcmd

pipcmd is a pipeline stage that treats the contents of its input records as pipeline commands to be issued. (The commands are usually callpipe.) Conceptually, this is very simple, but in practice using pipcmd typically involves writing a specs stage that writes another specs stage, so examples tend to look rather formidable. However, pipcmd is well worth learning to use, because it is an easy way to capture data from a record both before and after the record is transformed in some way. In most cases where pipcmd is used, the same function could be achieved by putting the callpipe command into a REXX stage, but using pipcmd is somewhat faster than is invoking a REXX stage.

pipcmd seems to be most easily understood by studying real-life uses, so here are two annotated examples:

```
/* FORBOB EXEC:  Read contents of a list of files into the       */
/*               pipeline, prefixing each record with the        */
/*               file name and the record number.                */

/* Parms:  Arguments to be used for LISTFILE command.            */

/* This pipe generates a series of pipelines and runs them       */
/* with pipcmd.  The generated pipelines are of the form:        */
/*                                                               */
/* CALLPIPE (stagesep !),                                        */
/*    ! < fn ft fm,                          * Read a file.      */
/*    ! spec ¢fn ft fm¢ 1 number 21 1-* 31, * Prefix each record.*/
/*    ! *:                                   * Send to caller.   */

'PIPE',
   '  command LISTFILE' Translate(Arg(1)),  /* Get list of files.*/
   '| spec /callpipe (stagesep !)',   /* Build CALLPIPE commands.*/
           ' ! </ 1 1-* nextword',
           '/! spec ¢/ next 1-* next /¢ 1 number 21 1-* 31',
           ' ! *:/ next',
   '| pipcmd',                        /* Run CALLPIPE commands.  */
   '| > output file a'                /* Write output to disk.   */
```

The FORBOB EXEC shown on the preceding page is equivalent to the following EXEC and REXX stage:

```
/* FORDAVE EXEC:  Read contents of a list of files into the    */
/*                pipeline, prefixing each record with the     */
/*                file name and the record number.             */

/* Parms:  Arguments to be used for LISTFILE command.          */

'PIPE',
   '  command LISTFILE' Translate(Arg(1)),  /* List of files.  */
   '| fordave',                          /* Bring them into pipe. */
   '| > output file a'                   /* Write output to disk. */
```

```
/* FORDAVE REXX:      Send contents of files into the pipe     */
/*                    preceded by file name and record number. */

Signal On Error

Do Forever                             /* Do until get EOF.       */

   'PEEKTO record'                     /* Examine the next record.*/

   'CALLPIPE',                         /* Create a pipeline.      */
      '<' record '|',                  /* Read specified file.    */
      'specs',                         /* Reformat each record:   */
         '/'record'/ 1',               /*   file name 1-20,       */
         'number next',                /*   record number 21-30,  */
         '1-* next |',                 /*   original record 31-*. */
      '*:'                             /* Send into main pipeline.*/

   'READTO'                            /* Consume input record.   */

End

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

The PUHASH command invoked in the next example returns its response via the stack. The sequence command | hole | append stack is an idiom that is frequently used with such commands. The command stage issues the command and captures any messages it writes. hole discards the messages captured by the command stage. Then the append stack stage retrieves the command responses from the stack and routes them into the pipeline.

```
/* HASHD REXX:                        Encrypt password in USER card */
Signal On Error

/*  The following CALLPIPE command reads an input record and
    uses it to build and execute another CALLPIPE command of
    the form:

    CALLPIPE (STAGESEP !)
       COMMAND PUHASH password userid !   /* Encrypt password.   */
       HOLE !                           /* Discard any messages.   */
       APPEND STACK !                   /* Get encrypted password. */
       SPEC ¢USER userid¢ 1             /* Reformat USER card.     */
          1.8 15
          ¢rest-of-card¢ 24 !
       *:                               /* Output to caller.       */
                                                                    */
'CALLPIPE (endchar ?)',
   '*: |',                              /* Input is USER card.     */
   'spec /',                            /* Build CALLPIPE command: */
      'callpipe (stagesep !)',
      'command PUHASH/ 1  w3 nextword  w2 nextword  /!',
      'hole !',
      'append stack !',
      'spec',
         '¢/ nextword 1.14 next /¢ 1',
         '1.8 15',
         '¢/ next 24-* next /¢ 24 !',
      '*:/ next |',
   'pipcmd |',                          /* Issue CALLPIPE command. */
   '*:'                                 /* Output to caller.       */

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

Addpipe and Callpipe: Connector Syntax

addpipe and callpipe are *CMS Pipelines* commands that are used to add pipelines to the running set. Just like the PIPE command itself, they take a pipeline specification as their argument. Their pipeline specifications can have one feature that PIPE's cannot: connectors. Connectors allow the pipelines added by addpipe and callpipe to connect to the streams of the existing pipelines, so that data can flow between them.

Connectors begin with an asterisk and end with a colon and are usually just that, "*:". But, between the asterisk and the colon, there may be two optional components. If these components are specified, they must begin with a period.

The first optional component may be .INput or .OUTput, indicating whether this is an input or output connector. When the connector is at the beginning of the pipeline, .INPUT is the default.

When the connector is at the end of the pipeline, .OUTPUT is the default. So, for example, the following two commands are equivalent:

```
'callpipe *: | take 5 | *:'

'callpipe *.input: | take 5 | *.output:'
```

The second optional argument specifies the stream to be connected to. .streamnum specifies the number of the stream to which the connection is to be made. For example, the following commands are equivalent and would write to the secondary output stream (stream 1) of the stage that invoked them:

```
'callpipe <' fn ft fm '| *.output.1:'

'callpipe <' fn ft fm '| *..1:'
```

.streamid specifies the stream identifier of the stream to which the connection is to be made. .* specifies that the connection is to be made to the currently selected stream. (This is the default.)

Fine Points of Using Callpipe

callpipe was introduced in the body of the paper, but there is more to be said on the subject:

- The connectors used by callpipe are called “redefine connectors” (see below). A redefine connector detaches a stream from the stage that issued the callpipe command and attaches it to the new subroutine pipeline. However, when an end-of-file condition is transferred from the subroutine pipeline through such a connector to the stage that issued the callpipe command, the original connection is restored. For example, after the completion of the following subroutine pipeline:

  ```
  'callpipe <' fn ft fm '| *:'
  ```

  the stage that issued the callpipe can write further records to its output stream.

- When an end-of-file condition is transferred from a subroutine pipeline through an *input* connector, there may still be records in the input stream; if so, the stage that issued the callpipe can read them. This situation arises when certain of the selection filters are used in subroutine pipelines. When their secondary output is not connected, those filters that would ordinarily discard the remainder of the input stream upon reaching a specified record instead simply signal end-of-file and terminate, leaving the remainder of the input stream intact. This allows one to “sip” data from an input stream.

  For example, this callpipe command could be used to position the input stream at the next record that has a comma in column 1 (or to read to end-of-file if there is no such record):

```
'callpipe *: | tolabel ,| hole'   /* To next leading comma. */
'peekto'                          /* Was there one?         */
If RC = 12  Then Exit             /* If not, exit.          */
```

And the following subroutine pipeline would copy the next five input records to the output stream while leaving any remaining records available to the stage:

```
'callpipe *: | take 5 | *:'
```

On Addpipe

The addpipe command is used to add one or more pipelines to the set of running pipelines and allow the stage that issued the addpipe to continue to execute in parallel with the newly added pipelines.[5]

When the addpipe command is invoked to add a new pipeline, the resulting changes in pipeline topology may include the breaking of connections between the invoking stage and other stages. A stream connected to a stage that issues an addpipe will be disconnected from that stage if it is referenced by a connector in the new pipeline.

What happens to the disconnected stream depends on the configuration of the connector that references it:

- It may become permanently connected to the new pipeline and have no further connection to the stage that invoked addpipe.
- It may be suspended while another stream temporarily occupies its connection to the stage that invoked addpipe.
- It may be connected to the new pipeline, which is in turn connected to the stage that invoked addpipe.

Thus, connectors may be specified in three possible configurations:

- A “redefine connector” detaches a stream from the stage that invoked addpipe and permanently attaches it to the new pipeline. A connector is a redefine connector when it is either:

  — An input connector at the beginning of a pipeline, or
  — An output connector at the end of a pipeline.

  This is an example of a redefine connector that transfers the invoking stage’s input stream to an added pipeline:

  ```
  addpipe *.input: | xlate upper | > output file a
  ```

  After this command is issued, the stage that issued it will get an end-of-file indication if it tries to read its input stream, because its input stream is no longer connected. The diverted input stream is processed by the new pipeline, which upper-cases the records as they pass through and then writes them to a file.

---

[5] This section represents an attempt to elaborate on the discussion of addpipe in the *CMS Pipelines User’s Guide and Filter Reference* (SL26-0018). It is heavily indebted to both that manual and the *Toolsmith’s Guide* (SL26-0020).

The following pipeline, which has redefine connectors at both ends, is identical to a pipeline short operation:

```
addpipe *.input: | *.output:
```

The stage that issues this addpipe command transfers both its input stream and its output stream to the added pipeline (which connects them to one another).

- A "prefix connector" suspends a connection between the stage that invokes addpipe and another stage and replaces that connection with one between the new pipeline and the invoking stage. If it is an input connector, records flow from the added pipeline through the connector into the invoking stage's input stream. If it is an output connector, records flow from the invoking stage's output stream through the connector into the added pipeline. A connector is a prefix connector when it is either:

  — An input connector at the end of a pipeline, or
  — An output connector at the beginning of a pipeline.

  When a stream is connected to a new pipeline with a prefix connector, the old connection is saved on a stack. End-of-file on the new connection sets return code 12 for a readto, peekto, or output command issued by the invoking stage. A sever command can then be used to restore the stacked connection.

  Here is an example of a prefix connector that temporarily connects an added pipeline to the input stream of the invoking stage (for the purpose of allowing that stage to read a parameter file before beginning its main work):

```
"addpipe < parm file| *.input:" /* Connect input to parm file. */
"nocommit"                      /* Disable automatic commit.   */
"readto line"                   /* Read first line of file.    */
do while RC=0                   /* Keep reading until EOF.     */
   "readto line"
end
"sever input"                   /* Re-instate input stream.    */
"commit 0"                      /* See if other stages are OK. */
if RC<>0 then exit 0            /* Exit quietly if not.        */
```

The addpipe command takes over the stage's input stream and begins reading the parameter file into it. The readto commands that the stage subsequently issues get the parameter records that were put into its input stream by the added pipeline. After the stage has read the last of those records, its next readto gets a return code 12 (end-of-file), which causes it to exit from the Do While group. It then issues a sever command, which re-instates its original input stream, and a commit 0 command, which waits for the other stages to be ready for data to flow through the pipeline.

- "Hybrid connectors" cause a stream to flow through both the added pipeline and the stage that added the pipeline. A pipeline has hybrid connectors when it has either:

  — Input connectors at both ends, or
  — Output connectors at both ends.

  When the added pipeline has input connectors at both ends, the input stream for the stage flows through the added pipeline before it becomes available to the stage. For example, this addpipe command would cause the input records for the stage to be deblocked before being read by any readto commands in the stage:

  ```
  addpipe *.input: | deblock net | *.input:
  ```

  When the added pipeline has output connectors at both ends, the output stream from the stage flows through the added pipeline before flowing into another stage. For example, this addpipe command would cause any records produced by output commands in the invoking stage to be upper-cased before being passed to the next stage:

  ```
  addpipe *.output: | xlate upper | *.output:
  ```

A single stage may invoke the addpipe command more than once. When the added pipelines use prefix or hybrid connectors, a given stream may be referenced repeatedly, resulting in stacking of its connections.

Addpipe with no connectors: A pipeline created by addpipe need have no connections to any of the other pipelines in the pipeline set. It may simply run in parallel with the other pipelines without data flowing between them. One way such an unconnected pipeline can be useful is in monitoring long-running pipelines, such as service machines. For example, the following addpipe might be used in a service machine that receives requests in the form of reader files:

```
'ADDPIPE (name THI)',              /* Timer-driven displays.   */
   '  literal +1:00',              /* Once a minute.           */
   '| duplicate *',                /* Forever.                 */
   '| delay',                      /* Wait for timer pop.      */
   '| specs /QUERY FILES/ 1',      /* Format Q FILES command.  */
   '| cp',                         /* Give it to CP.           */
   '| nlocate /NO RDR/',           /* Be quiet if caught up.   */
   '| specs',                      /* Format the display.      */
        '/Backlog is:/ 1',
        '7.5           nextword',
        '/files./      nextword',
   '| console'                     /* Put it on the console.   */
```

The added pipeline displays the backlog of requests once per minute, operating independently of the pipelines that are processing the requests.

Addpipe with hybrid input connectors: This example of using addpipe with hybrid input connectors is from Chuck Boeheim. It is a variant of the readlist stage discussed in the body of the paper. The pipeline added by the addpipe command processes the input stream for this stage before it is read by the readto command, converting each record into a traditional CMS ":READ" card, which is later written to the output stream by the output command before the subroutine pipeline created by the callpipe writes the contents of the specified file to the output stream. Note the use of lookup and the read keyword of spec:

```
/* GETFDATE REXX:     Send contents of files into the pipe       */
/*                    preceded by a :READ record bearing the     */
/*                    filename, disk label, and timestamp.       */

/* Input: filename;  Output: ":READ" card followed by the file.  */

Signal On Error

/* Add a subroutine pipe before this stage to put the filenames  */
/* into the format we need.                                      */

'AddPipe (endchar ?)'           ,  /* Pre-process input stream.. */
  '  *.input:'                  ,  /* ..for this stage.          */
  '| nfind *'                 || ,  /* Remove comments.           */
  '| nfind &TRACE'            || ,  /* And any control statement. */
  '| change /&1 &2 //'          ,  /* Get rid of exec args.      */
  '| change /&3 //'             ,  /* Some lists have three.     */
  '| state'                     ,  /* Ask CMS for the FST info.  */
  '| disk: lookup 19.1 15.1'    ,  /* Look up the disk info.     */
  '       detail master'       ,  /*                            */
  '| spec /:READ / 1'           ,  /* Put :READ word in,         */
  '       1.20     8'          ,  /*   then the fileid,         */
  '       57.17   36'          ,  /*   then the timestamp; and  */
  '       read'                ,  /*   from the next record,    */
  '       1.6     29'          ,  /*   get the disk label.      */
  '| *.input:'                 ,  /* Connect to ourselves.      */
'?'                             ,
  '  cms query search'          ,  /* Generate the disk table.   */
  '| disk:'                    ,  /* and feed to lookup.        */

Do Forever                         /* Do until get EOF.          */
   'readto record'                 /* Get next input record.     */
   'output' record                 /* Write back to stream.      */
   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              */
```

Note also that the addpipe command is issued before the stage issues any input commands to read from its input stream. This is required to establish the connection into the input stream before data begin flowing on it. The same consideration applies to the example below of using hybrid output connectors; the addpipe command must be issued before the stage issues any output commands, if the added pipeline is to process all the output records.

Addpipe with hybrid output connectors: The following example of using addpipe with hybrid output connectors is from Gregory DuBois, of SLAC. This addpipe command would be used in a stage that produces very large binary records as its output. The user wishes to save the data in a format that will later allow another pipeline to upload it with varload:

```
'ADDPIPE (name GENVAR long endchar ?)', /* Post-process output.. */
   '   *.output: |',              /* ..stream from this stage.    */
   '   fblock 64000 |',           /* Form into 64,000-byte recs.  */
   '   specs',                    /* Start making VARLOADable:    */
   '      number 1',              /*    subscript in 1-10;        */
   '      /:/ next',              /*    then delimiter (:);       */
   '      1-* next |',            /*    then the record.          */
   '   strip leading |',          /* Discard blanks in subscript. */
   '   change //:BINARY./ |',     /* Prefix delimited stem name.  */
   'c: count lines |',            /* Record count to secondary.   */
   'f: faninany |',               /* All data recs plus BINARY.0. */
   '   *.output:',                /* Output to next stage.        */
'?',                              /* End of first pipeline.       */
   'c: |' ,                       /* Record count to here.        */
   '   strip |' ,                 /* Deblank count for BINARY.0.  */
   '   specs',                    /* Make it VARLOADable:         */
   '      /:BINARY.0:/ 1',        /*    delimited name;           */
   '      1-* next |',            /*    then value.               */
   'f:'                           /* Merge into output stream.    */
```

A stage invokes this addpipe command to insert the GENVAR pipeline into its output stream with hybrid connectors. Thus, the records that flow out of the invoking stage flow through GENVAR before they get to the next stage. The output produced by the invoking stage is split into 64,000-byte records, and each record is prefixed by the string :BINARY.n:, where n is its record number. The record count is put into another record of the form :BINARY.0:count, and all the records are written to the output stream. Further on in the calling pipeline, these records might be written to disk. Later, they could be read from disk and put through a varload stage, which would store them as a stemmed array, making it convenient to use them for further computations.

Addpipe with prefix input connectors: The following example is from John Hartmann, the author of *CMS Pipelines*. It is a REXX stage that uses a stack of addpipe commands with prefix connections, restoring the stacked connections as its recursion winds down. The inclpack stage processes an input stream containing a PACKAGE file, which lists the files to be distributed as part of a software package. The files in the list may themselves be PACKAGE files, nested to any depth. The lists are processed recursively to produce output records for all the files required for the package. The PACKAGE file records have " &1 &2 " in columns 1-7 and a filename, filetype, and filemode in the next 20 columns.

```
/* INCLPACK REXX:              Include PACKAGE files recursively */
Signal On Novalue

Call Dofile                         /* Begin the recursion.        */
Exit                                /* Exit when done.             */

Dofile: Procedure
Parse Arg stack                     /* Packages being done now.    */

Do Forever
   'readto in'                      /* Next record from input.     */
   If RC <> 0  Then Leave           /* Leave if no more.           */
   If Left(in,7) ¬== ' &1 &2 '      /* Comment record?             */
      Then Iterate                  /* Yes, ignore.                */
   'output' in                      /* File record: to output.     */
   Parse Var in . . fn ft fm .
   If ft <> 'PACKAGE'               /* Iterate if not name of...   */
      Then Iterate                  /* ...another PACKAGE file.    */
   fid = fn'.'Left(fm,1)            /* Iterate if this package...  */
   If Find(stack, fid) > 0          /* ...already being processed. */
      Then Iterate
   'ADDPIPE <' fn ft fm '| *.input:' /* Add a pipe to put this...*/
   If RC <> 0  Then Exit RC         /* ...PACKAGE file into input. */
   Call Dofile stack fid            /* Process recursively.        */
   'SEVER input'                    /* Restore previous stream.    */
End

If RC = 12  Then Return             /* Return on end-of-file.      */
Exit RC                             /* Otherwise exit.             */
```

The Dofile procedure processes a PACKAGE file recursively. The argument to the procedure is a list of the PACKAGE files already being processed, which is used to prevent a loop caused by a package including itself (or including another package that in turn includes it).

The Do Forever loop reads a record and checks whether it names a file. If it does, the record is copied to the output. If the record names a PACKAGE file, a check is made to determine whether that PACKAGE file is among those currently being processed.

If the package is not being processed, then an addpipe command is used to inject the contents of the PACKAGE file into the pipeline. As the added pipeline has a prefix connector that references the stage's input stream, the current input stream is saved on a stack of dormant input streams, and the input stream for the stage is connected to the new pipeline, allowing its < stage to read the PACKAGE file into the pipeline.

The Dofile procedure is called to process the new package. When it is done, the input stream (which is now at end-of-file) is severed. That re-instates the stream on top of the dormant stack to continue reading the most recently interrupted file. This process continues until all the input streams have been processed and an output record has been written for every file required for the

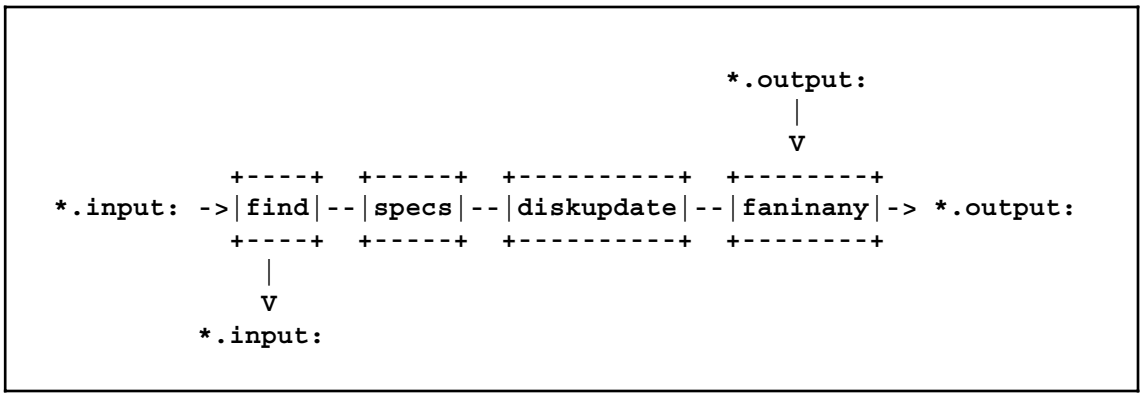
package. (As there is nothing here to prevent a file from being named twice, there would typically be a sort unique stage further on in the pipeline to discard duplicate records.)

Addpipe with hybrid input and output connectors: The following example is from Glenn Knickerbocker, of IBM. It is a stage that is used to apply updates to several different files. The input stream contains records suitable for use by a diskupdate stage, *i.e.*, each record is preceded by a 10-character field giving the number of the record it is to replace. There is an added wrinkle here, however: in front of the record number in each input record is a 20-character field containing the blank-delimited filename, filetype, and filemode of the file to be updated. This stage uses addpipe to create a new pipeline for each file that is to be updated:[6]

```
/* PUTFILES REXX */               /* Update specified files.     */
Signal On Error

Do Forever                        /* Do until end-of-file.       */

   'PEEKTO line'                  /* Examine next record.        */
   Parse Var line fn ft fm .      /* Extract file identifier.    */
   findname = Translate(Left(line,20),'_',' ') /* Fill with _'s. */

   'ADDPIPE (end / name PUTFILES)', /* Add a pipe for this file. */
      '|    *.input:',            /* Input from calling pipeline */
                  ,               /*   or other ADDPIPEs.        */
      '| a: find' findname,       /* Record for this file?       */
      '|    specs 21-* 1',        /* Yes, remove name field.     */
      '|    diskupdate' fn ft fm, /* Update specified file.      */
      '| b: faninany',            /* Merge the two streams.      */
      '|    *.output:',           /* Output to calling pipeline  */
                 ,                /*   or other ADDPIPEs.        */
   '/',
      '  a:',                     /* Updates for other files.    */
      '|    *.input:',           /* Input to other ADDPIPEs in  */
                ,                 /*   this stage (or PEEKTO).   */
   '/',
      '     *.output:',          /* Output from other ADDPIPEs  */
                 ,                /*   in this stage.            */
      '| b:'                     /* Go send to calling pipeline.*/

End

Error: Exit RC*(RC<>12)           /* RC = 0 if end-of-file.      */
```

[6] A pipeline specification may have a stage separator between the global options and the first stage. This may be necessary to distinguish between global options and options that apply only to the first stage.

The pipelines added by the addpipe command in this stage have the following topology:



A single pipeline of this topology can be very powerful. Because this pipeline begins and ends with input connectors, it has a hybrid connection into the input stream for the stage that invoked addpipe. This causes input records to flow through this pipeline before they can be seen by the stage. Because this pipeline also begins and ends with output connectors, it also has a hybrid connection into the output stream for the stage. This causes it to receive any output records from the stage before they flow into the next stage. Having hybrid connectors at both ends gives this pipeline the additional characteristic of being able to shunt records from the input for the stage to the output for the stage without their being seen by the stage.

In this example, this concept is carried even further, because multiple pipelines of this topology are created, one for each file to be updated. The result is "nested" hybrid connections on both the input stream and the output stream.

When the stage starts, the peekto command examines the first record, and the addpipe creates a pipeline for processing the file named in the first record. After that, the first added pipeline will divert any records for its file before they can be seen by the peekto. The next record that gets through to the peekto will be for another file, so addpipe will be used again to create a second pipeline for processing records for that file. This process will continue until a pipeline has been added for each of the files named in the input stream, after which the peekto will see no more input records. Each pipeline will read its input stream and divert records for its file to its output stream, while sending records for other files along its hybrid input connection, so that they can be passed to the other added pipelines until eventually reaching the right one. Output from the multiple added pipelines is similarly cascaded.

Addpipe with prefix input and output connectors: In the following example (from John Hartmann), a stage that already has primary and secondary streams defined issues an addstream pipeline command to establish tertiary input and output streams for itself:

```
'ADDSTREAM BOTH'                    /* Define tertiary streams.    */
```

It then issues an addpipe command to create a pipeline that will connect to those tertiary streams using prefix connectors, with the stage's output connected to the input of the added pipeline and *vice versa:*

```
'ADDPIPE (endchar ? name NODESYN)',
      '*.output.2: |',            /* Input from tertiary output. */
      'xlate |',                  /* Upper-case domain name.     */
      'pad 49 |',                 /* Pad to full key.            */
   'l: lookup 1.49 master |',     /* Find matching RSCS nodename.*/
      'spec /+/ 1 50-* next |',   /* Remember we got a match.    */
   'f: faninany |',               /* Join output streams.        */
      '*.input.2:',               /* Output to tertiary input.   */
'?',
      '< bitftp nodesyn |',       /* Read in nodename synonyms.  */
   'l: |',                        /* Non-matches come to here.   */
      'change //-/ |',            /* Remember didn't get match.  */
   'f:'                           /* Route to FANINANY.          */
```

This creates a permanent subroutine into which the stage can now and then throw a record and get back a response. The advantage of this arrangement over repeatedly creating a similar pipeline with callpipe is that the added pipeline is initiated only once; this might be a significant savings if, for example, the master file being read by the lookup stage is large.

Once the added pipeline has been affixed to the invoking stage, the stage can send records into the pipeline by writing on its tertiary output stream and can receive records from the pipeline by reading from its tertiary input stream. A convenient way to do this might be to use a subroutine pipeline such as the following:

```
'CALLPIPE (endchar ? name GETSYN)',
   'var usernode |',              /* Domain-style nodename.      */
   '*.output.2:',                 /* Into the added pipeline.    */
'?',
   '*.input.2: |',                /* From the added pipeline.    */
   'take 1 |',                    /* Stop when get one record.   */
   'find +|',                     /* Select for matches.         */
   'change /+// |',               /* Remove marker.              */
   'var nodename'                 /* Set user's nodename.        */
```

This inexpensive callpipe, which might be invoked repeatedly, writes one record to the added pipeline, receives one record back, and then terminates, returning control to the stage that issued both the addpipe and the callpipe.

For this scheme to work, the added pipeline must produce a known number of records from each input record and must not delay the record. If the stage uses output to write to the added pipeline and readto to read from it, then the added pipeline must have a one-record “elastic” (*e.g.,* a copy stage) to consume the record and thus allow the output to complete, so that the readto can be issued. If the stage uses a subroutine pipeline to connect to the added pipeline, then the subroutine pipeline must send an end-of-file out through both of its (redefine) connectors to the

invoking stage, so that the stage's connections to the added pipeline are restored when the callpipe command completes.

How Addpipe Differs from Callpipe

addpipe is similar to the callpipe command in that they both add pipelines to the running set. However, addpipe and callpipe differ from one another in several important ways:

- When callpipe is invoked, the stage that invokes it is suspended until the new pipeline has run to completion. When the callpipe command completes, its return code is the return code resulting from running the added pipeline.

  When addpipe is invoked, the stage that invokes it regains control as soon as the new pipeline has been created. The added pipeline runs in parallel with the stage that created it (and, in fact, the added pipeline can continue to run after the invoking stage has ended). Neither is able to examine the other's return code. The return code from the addpipe command itself indicates only whether its pipeline specification was syntactically correct.

- callpipe can use only redefine connectors. (This follows from the fact that the stage is blocked, so no data could flow on a prefix-style connection.) When a subroutine pipeline created by callpipe decides that it will process no more records and sets end-of-file to flow out through a connector to the invoking stage, the invoking stage's original connection is automatically re-instated.

  addpipe can use redefine connectors, but callpipe is more suitable for most cases where redefine connectors are required. When a redefine connector is used with addpipe, the original connection cannot be restored. When a prefix connector is used with addpipe, the restoration of the stage's original connection is not automatic, but requires an explicit sever of the added stream.

- A pipeline added by callpipe can run only at the same commit level[7] as the invoking stage (or a lower one). If a subroutine pipeline created by callpipe attempts to commit to a higher level than the invoking pipeline, it is suspended until the invoking pipeline reaches that commit level.

  The commit level of a pipeline added by addpipe is independent of the commit level of the invoking pipeline. Data can flow on a connection established by addpipe even if the pipelines on the two sides of the connection are not at the same commit level.

---

[7] For an explanation of commit levels, see the *CMS Pipelines User's Guide and Filter Reference* (SL26-0018) or *CMS Pipelines Explained*, by J.P. Hartmann, Proceedings of SHARE 78, Anaheim, CA, March, 1992, pp. 590-607.