CMS Pipelines **Explained**

John P. Hartmann, IBM Danmark A/S

SHARE 88

VM Cluster Session 9112

Abstract: This paper explains how *CMS Pipelines* works. It describes:

- The pipeline scanner which scans the argument string to the PIPE command as well as pipeline specifications issued in other ways.
- The pipeline dispatcher which passes control between stages (invocations of programs) to make data flow in the pipeline.
- The library of built-in programs from which the pipeline programmer selects building blocks to "solder" together to perform a particular task.

It then explains:

- How to write programs that use *CMS Pipelines* facilities in the Procedures Language VM/REXX.
- How to specify that a stage has more than one input or output stream and how to connect such stages to other pipeline stages to form multistream pipelines.
- How to reason about the relative order of records as they flow through a multistream pipeline network.
- How to change the pipeline topology dynamically.
- How to use "commit levels" to build co-operating programs that can co-ordinate their resource allocation and ensure that a function implemented by several stages that all have side effects is performed in total or not at all.
- How to deal with the dreaded pipeline stall.

The appendix describes some terms in common use that take on a specific meaning in the context of *CMS Pipelines* and others that become ambiguous.

It is assumed that the reader has some knowledge of CMS and REXX. Some experience with *CMS Pipelines* may help, but this is not an absolute requirement.

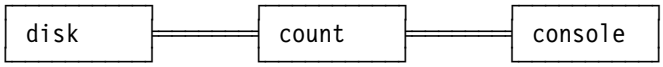
Introduction

CMS Pipelines pumps data through programs. Here is a command that runs a pipeline:

```
pipe disk input file | count words | console
```

This PIPE command reads a CMS file, counts the number of words in it, and writes the count to the terminal. You can enter this command directly from your terminal or you can put it in an EXEC; it is a normal CMS command in all respects. The argument string to the PIPE command tells it to find three programs, connect them, and run them:

Figure 1. Three Programs in a Pipeline.



CMS Pipelines consists essentially of the single CMS command, PIPE, which has three components:

- The *pipeline specification parser* (or informally: the *scanner*) scans the *pipeline specification* (PIPE's argument string) to identify programs to run and the connections between these programs.

  An invocation of a program in a pipeline is called a *stage*. A solid vertical bar (|) separates the specification of two programs in a pipeline specification; it represents the pipeline *connection* between them. By tradition, records flow from left to right through a connection; the stage to the left of a connection writes (or *produces*) records, the stage to the right reads (or *consumes*) them. Thus, the left-hand side of a stage is often called the *input side* and similarly the right-hand side is called the *output side*. A pipeline has a *first stage*, which is not connected on its input side to any other stage, and a *last stage*, which is not connected on its output side.

  Each stage runs a particular *invocation* of a program. Pipeline programs are re-entrant and can thus be invoked concurrently as often as needed.

- The *pipeline dispatcher* dispatches the specified programs to make data flow in an orderly fashion.

  The programs in a pipeline do not exchange data directly; rather, they exchange data via the pipeline dispatcher. They call the pipeline dispatcher to read a record from an *input stream* or to write a record to an *output stream*. This makes the pipeline interface *device independent*.

  The dispatcher does not buffer records between stages; instead, the dispatcher minimises the number of records *in flight* by giving priority to stages that can pump data out of the pipeline.

- A library of *built-in* programs.

  These programs access host interfaces (such as disk files or the virtual machine console), select records, transform records, and perform many other functions. The library has more than a hundred built-in programs.

The unit of information passed between programs in a pipeline is a *record* containing zero or more bytes. The pipeline dispatcher imposes no practical maximum size for records in the pipeline, but because a record being passed from one stage to the next is in virtual storage, the size of the virtual machine is an upper bound on the size of a record. There is in general no limit on the size of a file being processed by *CMS Pipelines*.

The Scanner: How to Specify a Pipeline

A pipeline specification is a string of characters that is processed by the pipeline specification parser (the scanner) to build a control block structure describing the programs to run and the connections between the programs.

Recall that the pipeline specification is the argument to the PIPE command; when you type it at the terminal, it is just a single line, as we have seen. (This is said to be a *landscape* representation.) In EXECs it is more convenient to write the pipeline one stage per line with a comma at the end of each line to indicate continuation. Here is the word counting pipeline from above in *portrait* format:

```
/* Count the number of words                                    */
'PIPE',
   ' disk input file',              /* Read input file          */
   '| count words',                 /* Count                    */
   '| console'                      /* Display result           */
exit RC                             /* Done!                    */
```

This format makes it convenient to keep a running commentary with each stage. It also makes it easy to add, delete, or move stages to *fine-tune* a pipeline specification.

To process this pipeline, the scanner first counts the number of stage separators to determine the number of stages. Each stage is described by the string between two stage separators (or to an end of the pipeline if the stage is first or last). The first blank-delimited word in each stage is the name of the program to run; the remainder of the string is the *argument string* to the particular invocation of the program.

To resolve the name, the scanner looks first in its directories of built-in programs; if it does not find the program there, it looks for a file with file type REXX; such a file is assumed to be a pipeline program written in the Procedures Language/REXX.

When the scanner has resolved all the programs to run, it checks the syntax of the argument strings to built-in programs. Some checks are decided by inspecting the *program descriptor* for the built-in program; other checks are performed by calling the *syntax routine* that the program descriptor specifies. For example, the program descriptor for the **<** program specifies that it must be the first stage in a pipeline; its syntax routine may verify that there are two or three words in the argument string and that the specified file exists.

In the sample pipeline in Figure 1 on page 2, the scanner performs the syntax check for *disk*, then for *count*, and finally for *console*. If any syntax check fails, the entire pipeline is abandoned.

When all syntax checks have been performed successfully, the scanner hands the pipeline specification over to the pipeline dispatcher to run the programs and pass data between them.

The Pipeline Dispatcher: How Pipeline Programs Run

So how does the pipeline dispatcher go about making data flow in a pipeline? This section tries to answer this in a simple fashion. To make the explanations simple, we are glossing over several important issues that will be explained in due course.

Using the example in Figure 1 on page 2, let us first look at what the *disk* stage does and ignore the rest of the pipeline. We shall assume that the input file contains only one record.

Figure 2. One Stage's Interactions with the Pipeline Dispatcher

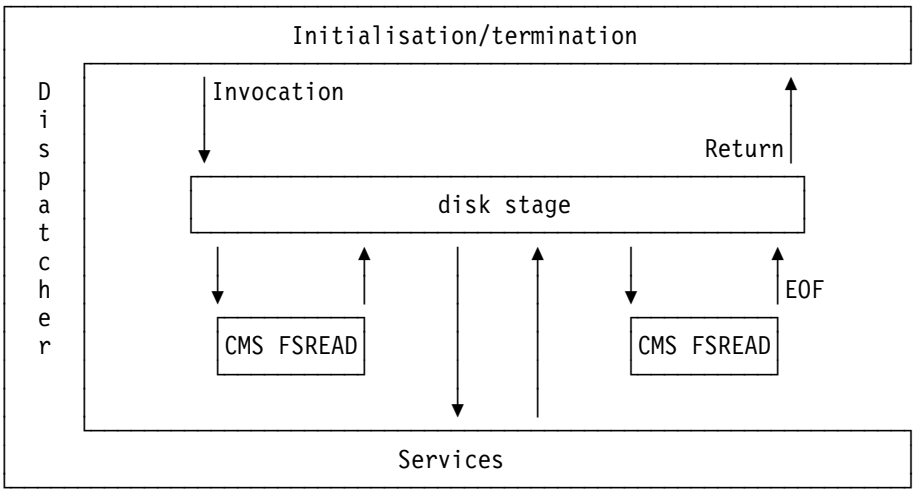


Figure 2 shows the steps that the dispatcher and *disk* go through. The arrows pointing up and down from left to right represent events as a function of time:

1. The dispatcher calls the entry point for *disk*.
2. *disk* allocates a storage area and then calls CMS to read the first record of the file into this area.
3. CMS returns to *disk*. As far as CMS is concerned, it is returning to the last command issued, PIPE.
4. *disk* calls the dispatcher entry point that writes a record into the pipeline to pass the record that it read from the file. (The dispatcher passes the record on to the next stage in the pipeline, but we shall not concern ourselves with this just yet.)
5. The dispatcher returns to *disk* when the output record has been processed and a new record is required.
6. *disk* calls CMS to read the next record from the file.
7. Because the only record in the file has already been read, CMS sets a return code to indicate end-of-file.
8. *disk* has done all it should do: It has read the file and passed it to the dispatcher. All that remains for *disk* to do is to release the storage area that it read the file into. *disk* then returns to the dispatcher.

You can see that *disk* is a very simple little program that has to worry only about CMS files; it just reads a file and passes it on. It neither knows nor cares about other stages in the pipeline.

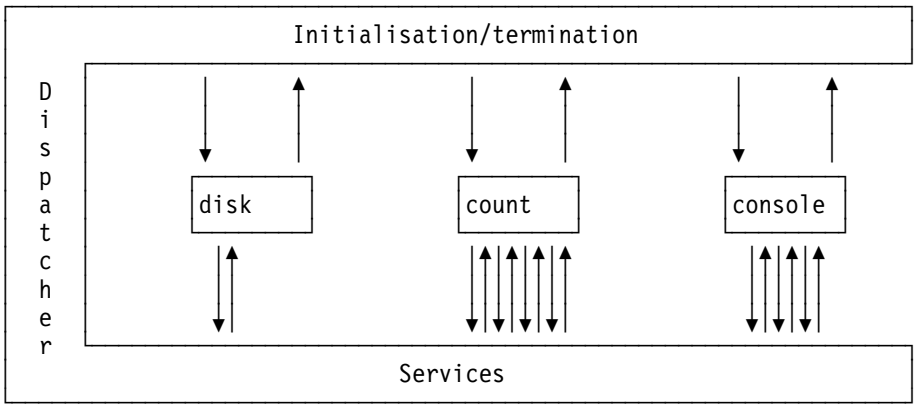
Consider that the dispatcher both called *disk* and was called from *disk*. The dispatcher has two faces: one that calls the stages in a pipeline and one that is called from these stages. Once a stage is invoked, the dispatcher never calls it again. Instead, the dispatcher returns control to the stage at the point where the stage last called the dispatcher. Unlike OS, the dispatcher never interrupts a stage in what it is doing; it gains control only when it is called by a stage to perform some function or when a stage returns on the original call from the dispatcher. (In computer science jargon, the pipeline dispatcher does not pre-empt the tasks it manages.)

The point that has not yet been explained is how the dispatcher goes about passing the record on in step 4. One might think that the dispatcher will gobble up all the output from *disk* and then present it as a unit to *count*; after all, most of us are brought up to finish one

task before beginning the next; we might perhaps think that the pipeline completes one stage before starting the next. Such a strategy would, however, limit the size of files that could be processed and cause unnecessary complications in the dispatcher; instead, the dispatcher transfers control between the stages in a pipeline using a strategy that is designed to expedite the flow of data and minimise the number of records in flight.

To understand how the dispatcher passes data between stages, look at the complete picture in Figure 3:

Figure 3. Interactions with the Pipeline Dispatcher



As before, the arrows pointing down represent calls; the arrows pointing up represent the return on the previous call by a stage or the dispatcher, but the calls and returns do not happen strictly from left to right this time:

1. The dispatcher calls *disk*, which reads a record from the CMS file and calls the dispatcher to write the record into the pipeline.
2. The dispatcher tests whether the stage that is connected to *disk*'s output is waiting to read a record. This is not the case; the dispatcher must suspend *disk*, because it cannot return to *disk* until the record has been processed. It then looks for other work to do.
3. The dispatcher now starts the next stage, *count*. That is, it calls the entry point for *count*.
4. *count* calls the pipeline dispatcher to read a record from the pipeline. *disk* and *count* are now at a *rendez-vous*; *disk* has produced a record and *count* intends to consume one. The dispatcher already has a record, which it passes on to *count* as it returns to *count*. (Note that it does not call *count* again; there are no recursions into stages.)
5. *count* scans the record it has just obtained and counts the words. It then calls the dispatcher to get the next record, in effect discarding the record it has just processed.
6. The dispatcher suspends *count* because it has no record available for it. But now the record that *disk* wrote has been processed; the dispatcher can resume *disk* in the hope that it will then write another record, which can be passed on to *count*.
7. Alas, the file has only one record. *disk* has no more to do; it returns to the dispatcher.
8. The initialisation/termination part of the dispatcher regains control and cleans up after *disk*. It notices that *count* is waiting for its next input record, but clearly there will never be one; the dispatcher sets a return code for *count* to indicate end-of-file and resumes *count*; that is, it returns to *count*.

9. *count* notes the end-of-file condition and writes a record containing the count of input records to its output. It does this by calling the dispatcher entry point that writes a record into the pipeline just as *disk* did.
10. *count* is now in the same situation as *disk* in step 2; *count* is suspended; the dispatcher finally starts *console*.
11. *console* calls the dispatcher to read a record.
12. The dispatcher makes available the record that *count* just wrote and returns to *console*.
13. *console* writes a line on the terminal and calls the dispatcher to read another record.
14. The dispatcher suspends *console* and resumes *count*.
15. *count* has no more to do and returns to the dispatcher.
16. The dispatcher resumes *console* indicating end-of-file.
17. *console* returns to the dispatcher.
18. All stages of the pipeline have now returned to the dispatcher; the PIPE command returns to CMS.

Each stage returns with a return code. The dispatcher *aggregates* the individual return codes into a single return code for the pipeline as a whole because a CMS command can return only a single return code. The return code selected is the maximum one if all stages return with non-negative return codes; if one or more stages return with a negative return code, the aggregate return code is the minimum of the return codes set by the individual stages.

As you can see, the pipeline dispatcher has a bit of magic inside, but not much. The unusual sequences are:

- When a stage calls a service entry point (the *back end* of the pipeline dispatcher), the dispatcher can revert to the initialisation part (the *front end*) to call the entry point for some other stage.
- When a stage returns to the termination routine, the pipeline dispatcher can go to the back end and return on some other stage's call to a service entry point.

The technical term for this behaviour is *co-routine dispatcher*. Thus, pipeline stages run as *co-routines*. A stage never calls another stage directly; rather, it causes one to be resumed, and is itself resumed in turn. There are many technical reasons why this is true *goodness*, but let this suffice.

The Library: Types of Pipeline Programs

The pipeline dispatcher does not require any particular protocol; a program running in a pipeline can issue calls to the dispatcher in any sequence that the program finds meaningful. The program can also access any host (CP or CMS) interfaces that it decides to use.

However, because the *CMS Pipelines* tradition calls for small simple programs, most of the built-in programs can be fitted into this taxonomy:

- *Device drivers* connect the pipeline to a host interface. These are typical ways for a device driver to work:
  - Read from a device into the pipeline. For example, **<** reads a CMS file into the pipeline. Such a device driver is always first in a pipeline.
  - Copy data from the pipeline to a device, and pass the data on to the following stage if there is one. This is like a T-junction. **>** and **>>** are examples of such device drivers. Such a device driver is never first in a pipeline.

  - Move data to or from the pipeline, depending on the position in the pipeline. For example, *console* reads from the terminal when it is first in a pipeline; it writes to the terminal when it is not first in a pipeline.
  - Write data from the pipeline to a device and write the device's response into the pipeline. *fullscr* does this.

- *Host command processors* issue host commands. Most direct the command response into the pipeline. The command is run to completion before any output is written. Stages that issue host commands can issue more than one command. If an initial command is specified with the arguments, it is issued first. The stage then reads input lines and issues them as commands.
- *Filters* perform some transformation on an input record to generate an output record. Filters do not reference host interfaces; they typically have no memory of previous records.
- *Selection stages* select records depending on some condition; for example, *locate* selects records that contain a particular character string. Records that are not selected are *discarded*, unless the secondary output stream is connected, in which case they are written to that stream.

  A subclass of the selection stages partitions the input stream into two segments; for instance, *take* selects a number of the input records and then discards the remainder.
- Several other stages compare the contents of records by key; for example, *lookup* finds references in a master file using keys in the input record. An in-storage *sort* is provided, as well as *merge* to merge ordered subfiles and *collate* to insert detail records into a master file.
- Control stages interact with the inner workings of *CMS Pipelines*; for example, to inject data *downstream* in a pipeline or to process input records as pipeline specifications. Some uses of control stages have been described as awesome, mind-boggling, or just plain magic.

The library that is available at a particular time consists of programs that are resident in the main pipeline module as well as programs in *filter packages* that have been loaded into storage and declared to *CMS Pipelines*. Four filter packages are loaded automatically if the corresponding modules are available when *CMS Pipelines* initialises itself during the first PIPE command in a CMS session.

Any number of additional filter packages can be installed by the user. A filter package installs itself when it is invoked as a CMS command. A filter package is removed from the *CMS Pipelines* search order by the CMS command NUCXDROP.

Your Own Programs: REXX Pipeline Programs

A REXX program running as a pipeline stage has a default command environment that processes *pipeline commands* rather than, say, CMS or XEDIT commands. Pipeline commands read and write records and perform other *CMS Pipelines* functions. The full REXX language is also available, of course; in particular, the Address instruction can address commands to an environment other than the default one. You might think perhaps of the command environment for REXX filters as PIPE, but it has no name and cannot be addressed explicitly. (There are convincing technical arguments for this state of affairs, even though you may at first feel that you are being cheated out of something.)

To invoke a REXX program that has the file type REXX, simply mention its name in a pipeline specification (the arrowhead indicates that the lines are output lines):

```
 pipe literal abcd | bagvendt | console
► dcba
►Ready;
```

Figure 4 on page 8 shows the REXX pipeline program, *bagvendt*:

Figure 4. A REXX Program to Reverse Lines.

```
/* BAGVENDT REXX -- Reverse the contents of lines in the pipeline    */
signal on error
do forever                              /* Loop through all lines      */
   'peekto data'                        /* Read a line into "data"     */
   'output' reverse(data)               /* Write the reverse           */
   'readto'                             /* Done with input record      */
end
error: exit RC*(RC<>12)
```

Let us go through the program line by line:

```
/* BAGVENDT REXX -- Reverse the contents of lines in the pipeline    */
```

The boiler-plate comment that indicates that the program is in fact in Procedures Language VM/REXX. ("Bagvendt" is Danish for reverse.)

```
signal on error
```

Transfer to the label `error` when a command gives a non-zero return code. A non-zero return code typically indicates end-of-file.

```
do forever                              /* Loop through all lines      */
```

The beginning of a loop to process all records.

```
  'peekto data'                         /* Read a line into "data"     */
```

Read a record from the pipeline and store it in the variable `data`; the variable is set as a side effect. The variable `RC` is set to the return code, as REXX does for all commands. At end-of-file, the return code on the PEEKTO pipeline command is 12; this causes control to transfer to the error label, because a Signal on error is active.

```
  'output' reverse(data)                /* Write the reverse           */
```

Evaluate the expression and write the result to the pipeline as a record.

```
  'readto'                              /* Done with input record      */
```

Inform *CMS Pipelines* that the stage is finished with the input record.

```
end
```

The end of the loop; control transfers to the beginning to process the next input record.

```
error: exit RC*(RC<>12)
```

REXX transfers control to this statement when a pipeline command sets a non-zero return code, most likely 12, which means end-of-file. The novel aspect is that the end-of-file condition could be on the output rather than the input. In either case, the return code from *reverse* should be zero; end-of-file is not an error.

Subroutine Pipelines: A REXX program often needs to run a pipeline to perform a subtask. Though it is possible to use the Address instruction to issue the CMS command PIPE from a REXX pipeline program, this is not particularly useful, because the new pipeline cannot connect to the streams defined for the REXX program.

The pipeline command CALLPIPE is provided to run a pipeline that can be connected to the original pipeline in place of the stage that issues CALLPIPE. The CALLPIPE pipeline command has the same syntax as the PIPE command. A pipeline specification that is issued with CALLPIPE is called a *subroutine pipeline*. Here is a typical subroutine pipeline:

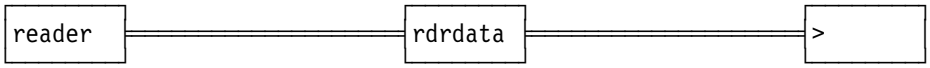
Figure 5. A Subroutine Pipeline

```
/* RDRDATA REXX -- Get data portion of a reader file                */
signal on novalue
'callpipe',
   '|*:',                              /* Input comes in here       */
   '|strfind x41',                     /* Look for cards            */
   '|spec 2-* 1.80',                   /* Discard opcode; pad to 80 */
   '|deblock netdata',                 /* Get original records      */
   '|strfind xc0',                     /* Discard control information */
   '|spec 2-* 1',                      /* Remove flag byte          */
   '|*:'                               /* Pass on to next stage     */
exit RC
```

The `*:` at the beginning and end of this pipeline specification are the magic incantations that cause it to be connected to *rdrdata*'s neighbour stages. They are called *connectors*; we shall have more to say about them in a later section. When this CALLPIPE pipeline command is run, *rdrdata*'s input and output streams are disconnected from the streams in the neighbour stages; those streams are then connected to the subroutine pipeline created by the CALLPIPE pipeline command. When all stages in the subroutine pipeline have completed, the streams are restored to *rdrdata* and the CALLPIPE pipeline command completes.

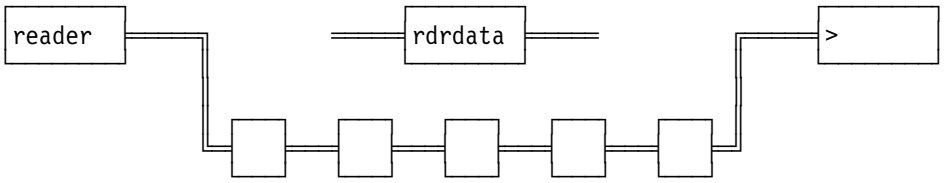
Here is a typical application of *rdrdata*, between a stage to read a reader file and a stage to write the file to disk. The original pipeline looks like this:

Figure 6. A Pipeline with Three Stages.



After the CALLPIPE pipeline command in *rdrdata* has added its pipeline specification to the set of running pipelines, the pipeline set looks like this:

Figure 7. A Pipeline with a Subroutine Pipeline



The connection between *reader* and *rdrdata* has been replaced with a connection from *reader* to the first stage of the subroutine pipeline (the first *strfind*); the input stream to *rdrdata* is temporarily not connected. Similarly, the connection between *rdrdata* and `>` has been replaced with a connection from the last stage of the subroutine (the second *spec*) to `>`; the output stream from *rdrdata* is also temporarily not connected. When all stages of the subroutine pipeline have completed, the original connections are restored and *rdrdata* is resumed.

Short Circuits: Another useful pipeline command is SHORT. It causes the input stream and the output stream that are connected to the stage's streams to be connected directly to one another, bypassing the stage altogether.

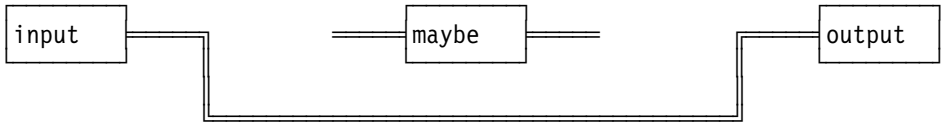
For example, you may often want to inspect the first record in the pipeline (or the first few records) to determine how to process the rest of the file; in some cases no additional proc-

essing is required; the data should just be copied to the output without modification. The SHORT pipeline command copies any remaining input to the output more efficiently than can be done in a loop, because the dispatcher does this by manipulating the connections between streams instead of moving data.

To preface a title record when the first record does not have the character '1' in the first column:

```
/* Maybetitle REXX                                              */
'peekto firstline'                  /* Read first record        */
If RC=12                            /* No input?                */
   Then exit                        /* Ignore...                */
If left(firstline, 1)/= '1'         /* Is title provided?       */
   Then 'output 1'arg(1)            /* Supply our own if not    */
'short'                             /* Copy everything else     */
```

Figure 8. After SHORT Pipeline Command



If you think that the first record of the file has been lost in this example, read on; all is explained in the next section.

There are many more pipeline commands; we shall mention them in the following when the corresponding concept is described.

Moving Data and Not: Producing, Consuming, and End-of-file

The stage on the left-hand of a connection *produces* a record when it writes one into the pipeline. When the stage on the other side of the connection *consumes* the record, the *rendez-vous* between the producer and consumer is over; each can continue its own processing. The record has moved across the connection: the producer can re-use the output area; the consumer can read another record.

A particularly useful facility of *CMS Pipelines* is the PEEKTO pipeline command which *peeks* at a record without consuming it. A peek is a non-destructive read; a particular record can be peeked any number of times before being consumed with a READTO pipeline command. When a peek completes, the producer is guaranteed to be blocked in a write, or there is no producer—the stream has been severed and is at end-of-file. Blocking the producer ensures that the contents of a record are not changed while it is being processed by a subsequent stage.

Thus, a stage can determine whether it likes a record before consuming it; in fact, a stage can produce a derivative of a record it has peeked before it consumes that record. Stages that do this do not *delay the record*. A stage can also peek at the first record and select the subroutine pipeline that is appropriate to the format of the data on hand.

The ability to peek at a record lets a stage that partitions the input terminate without consuming the record that it is looking for. That is, it can partition the input *before* some particular record. This is especially useful in subroutine pipelines, where it can be used to "sip" the data, that is, to process records only up to the next marker record. For example, a *tolabel* stage at the beginning of a subroutine pipeline can stop at a marker record:

```
/* Position input at first USER card */
'callpipe *: | tolabel USER | hole'
```

The records up to the marker are consumed in the subroutine pipeline by *hole*, which sucks up all input records and produces no output. *tolabel* terminates when it sees the marker record (here a USER card in a CP directory); the marker record and the rest of the input records stay in the input pipeline where they can be read (peeked or consumed) by the stage that issued the CALLPIPE pipeline command when it is resumed after the subroutine pipeline completes.

As part of the clean-up after a stage has returned to the dispatcher, the dispatcher goes through the stage's streams and *severs* those that are still connected. The stage at the other side of such a connection now receives end-of-file; by definition an *unconnected* stream is at end-of-file. Once a stream becomes unconnected, no other stage can reach it and data can never flow on it again.

If a stage is blocked in a read or write for a stream that becomes unconnected, it receives return code 12 and is made dispatchable. Subsequently, a stage receives return code 12 immediately if it reads or writes to an unconnected stream.

It should be no surprise that a device driver at the beginning of a pipeline (for example, <) terminates when it has read the file into the pipeline; the stage after the device driver gets return code 12 when it tries to read and the input stream is not connected. As we saw in the explanation of Figure 3 on page 5, this caused the following stage to terminate, and so on; thus, the stages *propagated* end-of-file (*forward* in this case).

But some stages can set end-of-file on their input sides as well. For example, let us add a stage to the example above:

```
/* Position input at first USER card */
'callpipe *: | xlate upper | tolabel USER | hole'
```

The *xlate* stage ensures that the records passed to *tolabel* are in uppercase. Watch as a USER record moves through this pipeline:

1. *xlate* peeks at the record, builds an uppercased record in a buffer and writes it to the pipeline.
2. *tolabel* peeks at the record, determines it has seen what it wants, and returns without consuming the input record.
3. The dispatcher cleans up after *tolabel*. It sets end-of-file for *hole*. No doubt, we all expected as much. But it also sets end-of-file on *xlate*'s write!
4. *xlate* has no reason to continue; no one will ever read its output records; it can perform no useful function and, therefore, it returns without consuming the USER record, which remains in the input pipeline.

Thus, end-of-file can propagate *backwards* in a pipeline as well.

Here is yet another twist to this concept:

```
/* Position input at first USER card */
'callpipe *: | xlate upper | fromlabel USER '
```

fromlabel is the converse of *tolabel*. It discards records until it meets one that starts with the specified string. The matching record and the remaining input are passed to the output. But when *fromlabel* in this example writes the USER record to its output, it receives end-of-file immediately (think of the dispatcher waiting to pounce!) and returns without consuming the USER record. This, too, causes end-of-file to propagate backwards.

You might be wondering what happens when the end-of-file condition reaches the connector. We explain the semantics of CALLPIPE in a later section; suffice it to say here that end-of-file does not propagate outwards through a connector.

Recall that device drivers that are neither first nor last in a pipeline copy the records to the output as well as to a host interface. Such a program continues to process records until it sees end-of-file on its input, because its primary task is to access the host interface; copying records to the following stage is a convenience; the fact that the output stream is no longer connected has no effect on the host object being created or modified.

As we have seen *count* so far, it reads all its input and produces a single output record when its input is at end-of-file; thus, it *propagates* end-of-file only from its input side. *count* can also operate as a "gas meter", measuring the data flowing through it; it enters this mode when it finds that it has a secondary output stream. In the meter mode, *count* propagates end-of-file both ways; that is, when its primary output stream becomes unconnected, it writes the current count to its secondary output stream and returns to the dispatcher without consuming (or counting) the record that caused it to terminate.

Note that the decision on what to do at end-of-file is taken by the individual program; the pipeline dispatcher does not require or enforce any particular protocol. In general, however, the *CMS Pipelines* built-in filters propagate end-of-file in both directions, while the device drivers that are not first in a pipeline propagate end-of-file only from their input.

Propagating End-of-file for Real, Summarily Stopable Stages

Propagating end-of-file forwards in a pipeline is seldom a challenge for the pipeline programmer; after all, when there is no further input, it is natural to stop.

Propagating end-of-file backwards as quickly as possible is not as easy as it may seem. To terminate a program as soon as it is clear that no further productive work can be accomplished, a change in the semantics of pipeline commands is needed. Consider this pipeline segment:

```
... | bagvendt | take 1 | ...
```

bagvendt (which was shown in Figure 4 on page 8) adheres to the rules for robust pipeline programming explained so far. Now watch as the first record flows through this pipeline segment:

1. *bagvendt* peeks at the record and writes the reverse of it. This causes *bagvendt* to be blocked waiting for its output to be consumed.
2. *take* peeks the record and passes it on. Eventually its output record is consumed and it consumes its input record.
3. Both *bagvendt* and *take* can now run. Let us assume that *take* is resumed first; it terminates without doing further pipeline I/O.
4. *bagvendt* consumes its input record and peeks for the next.

At this point, if *bagvendt*'s output stream is not already severed, it will be severed very soon; but *bagvendt* does not discover this fact immediately. Once it has issued the PEEKTO pipeline command, it must wait for an input record to arrive or the input stream to be at end-of-file. It will discover (from the return code on OUTPUT) that its output stream is at end-of-file when it produces the next output record.

Built-in programs can propagate end-of-file backwards faster than *bagvendt* can do because they can be defined as *summarily stopable*. Essentially, a summarily stopable program has declared to the pipeline dispatcher that it may be terminated at any time it is waiting for input and its input stream or its output stream is severed. In return, the stage is restricted to use buffers that are declared to the pipeline dispatcher.

Some stages cannot be made summarily stopable. For example, stages that access host interfaces often need to release a resource before they terminate; others, such as *unique* LAST still need to write a record when they get end-of-file on their input stream. For such

filters, a facility is available to modify the semantics of the PEEKTO pipeline command. Such stages can now receive return code 8 on PEEKTO when their output stream is or becomes severed before the next input record is available.

Note that the notion of a stage being summarily stopable is not fool-proof. If the consuming stage were not to run after it consumed the input record, *bagvendt* might get dispatched, consume the first record and manage to sneak a peek at the second record before the consuming stage could terminate. This is a theoretical exposure more than a practical one, however.

Clearly, the *rexx* interface itself cannot be made summarily stopable, because it must unwind the REXX environment before it can return to the pipeline dispatcher. Instead, the pipeline command EOFREPORT ALL is used to modify the behaviour of PEEKTO to set return code 8 when the output stream is severed. A revised *bagvendt*, which propagates end-of-file backwards as fast as is possible, is shown in Figure 9.

Figure 9. Revised BAGVENDT

```
/* BAGVENDE REXX -- Reverse the contents of lines in the pipeline    */
trace off
'eofreport all'                        /* Prepare to propagate EOF    */
trace failure
signal on error
do forever                             /* Loop through all lines      */
   'peekto data'                       /* Read a line into "data"     */
   'output' reverse(data)              /* Write the reverse           */
   'readto'                            /* Done with input record      */
end
error: exit RC*(RC<0)
```

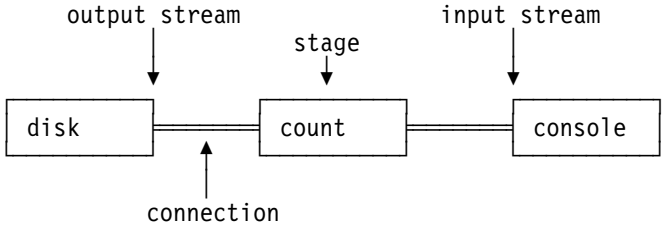
The change from the previous example is the three lines added at the beginning of the file (lines two through four). Line three (EOFREPORT ALL) enables the new interface. The second line gives backwards compatibility; it suppresses REXX trace of the negative return code, which would be issued if the version of *CMS Pipelines* being used does not include support for EOFREPORT. The fourth line enables tracing of subsequent failures.

The last line of the example is amended to cater for the possible return codes. PEEKTO still returns 12 when the input stream is at end-of-file, that is, when the stage should propagate end-of-file forwards. Thus, any positive return code is now considered OK, rather than just 12.

Multistream Pipelines: Sending Data on many Paths

Stages read and write records on streams. The examples in previous sections all show a single pipeline where each stage has one input connection and one output connection. The connections are, in fact, between streams; the stage reads from and writes to a stream:

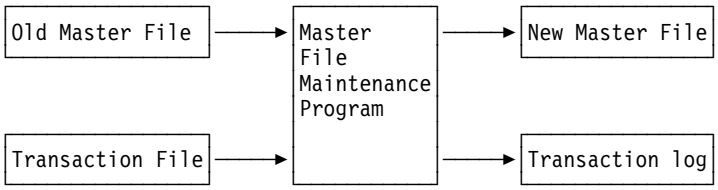
Figure 10. Pipeline Topology



A stage always has a primary input stream and a primary output stream defined, although they are not necessarily connected. A configuration where one or more stages have more than one input or output stream is called a *multistream pipeline*.

The multistream pipeline support was originally designed for master-file update, where a stage needs access to multiple input and output streams:

Figure 11. Master File Update



Though introduced for *update*, multistream pipelines are now more often used with selection stages. For example, *locate* selects records that contain a specified string and discards the records that do not. When *locate* has only one output stream, the records that are not selected are indeed discarded, but when it has two output streams, the discarded records are written to the secondary output stream rather than being tossed into the bit bucket:

Figure 12. LOCATE with Two Output Streams

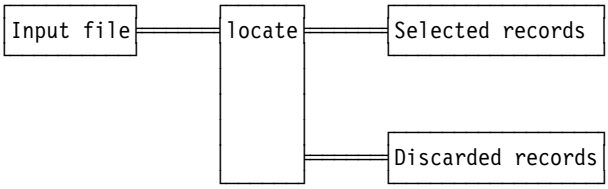


Figure 12 shows the topology of *locate* when it is used to create two output files, one containing the selected records and one containing the discarded records. Between them, the two output files contain all the records from the input file; no records are lost and none are duplicated when a selection stage has both output streams connected.

There are two pipelines in Figure 12; the upper one (which reads the input file, selects records, and writes the “selected” output file) and the lower one (which writes the discarded records to the “discarded” file). You might think of the two pipelines like this:

```
< input file | locate /Sid/ | > selected  records
                *magic*     | > discarded records
```

The key point is that the one invocation of *locate* is in **both** pipelines; it reads from its *primary input stream*, writes selected records to its *primary output stream*, and writes discarded records to its *secondary output stream*.

Labels are used to specify that a stage is in more than one pipeline; one part of the magic is simply a reference to a label:

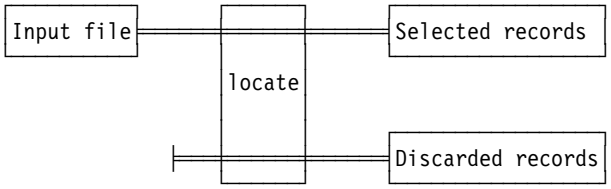
```
< input file | Loc: locate /Sid/ | > selected  records a
               Loc:              | > discarded records a
```

A label consists of one to eight characters that are not all digits; a trailing colon completes the label. Here, the label is `Loc:`; this is not the same label as `loc:`, because case is respected in labels. The label is written in front of the specification of the stage it refers to, in front of the name of the program to run. The first occurrence of a label defines the label along with the stage it represents; when a label is used subsequently, it is a reference

to the stage where the label was defined. The second time a particular label is used in a pipeline specification, it specifies how the labelled stage's secondary streams are connected; what is to the left of this label is connected to the labelled stage's secondary input stream; the secondary output stream from the labelled stage is connected to what follows the label. A secondary input stream is *not connected* when there is no stage separator before the label reference; likewise, a secondary output stream is not connected when there is no stage separator after the label reference. The third time a label is used defines the tertiary streams (a few built-in programs support three or more streams). Because the program to run is specified when the label is defined, the second and subsequent uses of a label (*label references*) occur by themselves between stage separators.

So far, so good; it is easy enough to draw a diagram of the required pipeline topology and it seems conceptually simple to have two pipelines. Maybe it would help to redraw Figure 12 on page 14 to show the tubing going through *locate*. The second pipeline is capped at the beginning to show that the secondary input stream to *locate* is not connected:

Figure 13. LOCATE is in Two Pipelines.



Take a moment to look at this figure. From the perspective of a programmer who is accustomed to procedural code, it represents a program (*locate*) and its associated input and output streams. *CMS Pipelines*, however, emphasises data flow, not programs; to a *master plumber* the figure shows two pipelines that *intersect* in the *locate* stage; the connections are arranged to accomplish the particular task on hand.

But how can we write the multidimensional pipeline structure as an argument string? PIPE is a CMS command; CMS provides a single parameter string to the program when the command is issued.

Just as stage separators are used between stages, so are pipeline separators (called *end characters*) used between pipelines. However, there is no default pipeline separator character; you must use the option ENDCHAR (often abbreviated to END) to declare one in each multistream pipeline specification. Pipeline options are written in parentheses at the beginning of a pipeline specification:

```
/* Findm REXX */
'PIPE (end ?)'
```

This defines the question mark as the end character for this pipeline specification. Then write the upper pipeline just as you would have done without the second pipeline, but remember to put a label on the *locate* stage:

```
/* Findm REXX */
'PIPE (end ?)',
   '< input file | Loc: locate /Sid/ | > selected  records a'
```

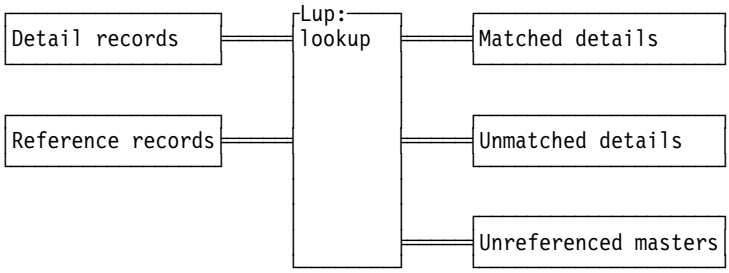
This pipeline selects all records that contain the string "Sid". To process records that are rejected, add a comma to indicate that the command is continued on the following line and write a question mark to separate the two pipelines, followed by the lower pipeline. Space it out to look nice even though the scanner does not require a particular layout; this will make it easier to understand for others as well as yourself:

```
/* Findm REXX */
'PIPE (end ?)',
   '< input file | Loc: locate /Sid/ | > selected  records a',
  '?              Loc:               | > discarded records a'
```

The secondary input stream to *locate* is not connected, because the label reference is immediately after an end character.

As an example of a more complex stream arrangement, consider *lookup*. It looks up a key field in records on the primary input stream in a reference directory it has built from the records on the secondary input stream; thus, it uses two input streams. The records on the primary input stream (the detail records) that have a matching record in the reference are written to the primary output stream (by default followed by the matching reference record). The detail records that have no matching reference record are written to the secondary output stream. After all input records have been processed, those reference records that were never matched are written to the tertiary output stream:

Figure 14. Using LOOKUP



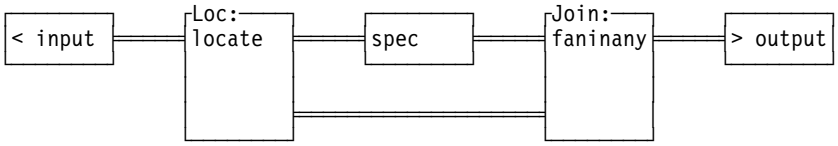
Here is the corresponding pipeline specification, assuming that the key is in columns 1-10 of both detail and master records:

```
/* JARGMSP EXEC */
'PIPE  (end ?)',
   '|     < detail records',
   '|Lup: lookup 1.10 details',
   '|     > matched details a',
   '?     < reference records ',
   '|Lup: ',
   '|     > unmatched details a',
   '?Lup:',
   '|     > unreferenced masters a'
```

Note that the label is used three times; it is a common beginner's mistake to define four pipelines for this particular task, thinking perhaps that a label must always be next to an end character. The second occurrence of the label defines both the secondary input stream and the secondary output stream, both of which are connected. The third occurrence of the label defines the tertiary input and output streams, but only the output stream is connected.

A multistream pipeline specification needs not have more than one input device driver and one output device driver. Figure 15 shows a pipeline topology where two pipelines join back into a single stream before the result is written to a file. Some of the records in the file are passed through the *spec* stage; others are not:

Figure 15. Rejoining Pipeline



The corresponding pipeline specification is:

```
/* Markem EXEC */
'PIPE (end ?)',
   '|      < input file',
   '|Loc:  locate /Sid/',
   '|      spec 1-* 1 /(!)/ next',
   '|Join: faninany',
   '|      > output file a',
   '?Loc:',
   '|Join:'
```

The second pipeline has no stages between the two secondary streams, it consists of just the two label references.

To recap, a stage has one or more streams at its disposal when it begins. That is, one or more input and output streams are *defined* by the scanner. A stream can be *connected* or *unconnected*. The input stream of the first stage in a pipeline is not connected, nor is the output stream of the last stage of a pipeline. A one-stage pipeline has no connected streams, but its streams are defined nevertheless.

The *scope* of a label (that is, where it is recognised) is a pipeline specification. There is no memory of a label once the pipeline specification is passed from the scanner to the dispatcher; indeed, a stage cannot determine that a label is declared for it, but not referenced.

A pipeline program uses the dispatcher calls already described to read and write, even when it has more than one pair of streams defined. The dispatcher reads from the currently selected input stream and writes to the currently selected output stream. Initially the primary streams are selected; the stage issues the SELECT pipeline command to specify the stream to be used for subsequent reads or writes (or both).

The SELECT pipeline command specifies the side to operate on (input, output, or both) and the stream to select. The stream can be specified as a number (the primary streams have number 0, the secondary have number 1, and so on) or as a *stream identifier*, which is one to four characters that are not all numeric digits (because that would be interpreted as a stream number).

A stream identifier may be declared when a label is used in a pipeline specification; not all uses of a particular label need have an associated stream identifier:

```
...
| label.old: mystage
| ...
? label.new:
| ...
| label:
```

In this example, *mystage* has three pairs of streams. Both primary streams are identified by `old`; the secondary streams are identified by `new`; the tertiary streams have no identifier. As in labels, case is respected for stream identifiers. The scope of a stream identifier is the stage identified by the label.

When using stream identifiers, the program becomes independent of the particular order in which the streams are specified in the pipeline specification; this should make life easier for the pipeline programmer.

Several pipeline commands support multistream REXX programs. MAXSTREAM sets the return code to the largest number that can be used in a SELECT pipeline command; it sets return code zero when a stage can access only the primary streams. STREAMSTATE sets the return code to indicate the status of a specified stream (for example: defined, connected, or

data available). A stage can issue the ADDSTREAM pipeline command to add an unconnected stream to it. Once a stream is defined (by the scanner or the ADDSTREAM pipeline command), it cannot be discarded; only the connections to the stream can be manipulated.

We saw earlier that SHORT causes the dispatcher to connect the streams up to bypass a stage. When a stage has multiple streams, SHORT refers to the currently selected streams; other streams remain intact.

When a stage has multiple streams, EOFREPORT ALL will cause PEEKTO and SELECT ANYINPUT to terminate with return code 8 when all output streams are at end-of-file. This is satisfactory for many multistream programs, but some programs, such as *gate*, may wish to propagate end-of-file on individual streams. The command EOFREPORT ANY supports writing such programs in REXX, but it is a sharp tool, which should be used with care.

When EOFREPORT ANY is active, a stage will receive return code 8 on PEEKTO and SELECT ANYINPUT, as described for EOFREPORT ALL. In addition, the stage will receive return code 4 on PEEKTO or SELECT ANYINPUT if any stream changes state before a record is available on input; and it will receive return code 4 on OUTPUT if any stream changes state before the output record was peeked at by the consumer stage.

When the stage sees return code 4 it can use STREAMSTATE ALL to obtain the state of all input and output streams with a single command. The variable set by STREAMSTATE ALL can then be parsed to determine the state of the individual streams.

The stage would normally restart the input or output operation after it has propagated the end-of-file condition. Thus, stages that use EOFREPORT ANY must wrap loops around the I/O operations they perform, for example like this:

```
signal off error                      /* Need to inspect RCs         */
do forever                            /* All severed streams         */
   'output' record                    /* Write record                */
   if RC ¬=4                          /* Some other stream severed?  */
      then leave                      /* No, I/O done or failed      */
   call figurestreams                 /* See what was severed        */
end
```

The “figurestreams” routine would then scan the state of the streams and propagate end-of-file.

```
figurestreams:
'streamstate all states'
do stream=0 while states ¬= ''
   parse var states state states      /* Get one stream pair         */
   if wordpos(state, '0:0 12:12')>0   /* OK or both at EOF?          */
      then iterate
   parse var state instat ':' outstat /* Get individual streams      */
   select
      when instat=12                  /* Input at EOF?               */
         then side='output'           /* then sever output           */
      when outstat=12                 /* Output at EOF?              */
         then side='input'            /* then sever input            */
      otherwise                       /* Neither at EOF              */
         continue                     /* Ignore                      */
   end
   'select' side stream               /* Prepare to sever.           */
   'sever' side
end
```

Delaying the Record: The Order of Arrival

Recall the multistream pipeline in Figure 15 on page 16. Assuming the input file contains the records on the left, we would expect the output file to contain the records on the right:

```
Don't forget
to tell Sid
about this.
```

```
Don't forget
to tell Sid(!)
about this.
```

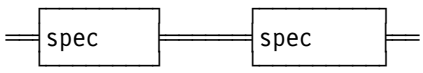
There would be cause for concern if the records in the output file were not in the same order as the records in the input file, but how can we be sure this is the case and how can we reason about other pipeline topologies?

Let us take a closer look at how *spec* goes about processing an input record, for the moment ignoring small matters like end-of-file:

1. *spec* takes a peek at the input record (this operation is also known as doing a *locate-mode* read). When the peek completes successfully, the producer of the record is blocked in an incomplete write operation to ensure that the record is not modified before being consumed.
2. *spec* builds an output record in a work area.
3. *spec* writes the record from the work area to the output stream, itself being blocked until the record has been consumed.
4. Once the output record has been consumed, *spec* consumes the input record (this operation is also known as doing a *move-mode* read for zero bytes). This releases the producer stage from its blocked write. Both *spec* and the stage to its left can now run; conceptually they run in parallel, but *spec* will be blocked if it peeks and there is no new record yet available.

Now consider what happens when two *spec* stages are *cascaded*:

Figure 16. Two Cascaded SPEC Stages.



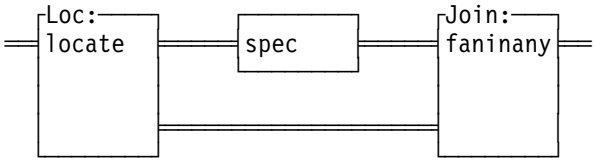
As before, the first *spec* stage writes an output record. When the second *spec* stage has peeked successfully at that record, we know that the first *spec* stage must be blocked in step 3 writing the record, and that its producer, in turn, is also blocked. Thus the producer cannot produce another record while this one is *in flight*.

In general, in a cascade of stages that do not delay the record, a single record moves from stage to stage, leaving all the stages it passes through blocked. When the record reaches the end of the pipeline, then the last stage, which does no output operation to the pipeline and thus does not become blocked in a write, can consume the record; this unblocks the next-to-last stage, which can then consume its input record, unblocking the previous stage, until all the stages unblock, like dominoes falling.

Thus, a cascade of filters that do not delay the record behaves as a unit. Only one record will be traversing the cascade at any one time; the order of the individual records cannot be changed in the course of traversing the cascade. This remains true even if the cascade includes multiple streams and even if there are different numbers of stages in different paths. Records leave the cascade in the same order they enter it; that is, they are not delayed relative to one another.

Let us now walk the three records through the pipeline network we saw earlier:

Figure 17. Multistream Segment



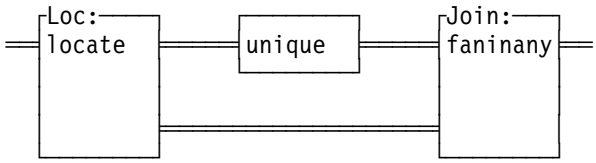
1. *locate* peeks at the first record, which does not contain the required string; *locate* writes the record to its secondary output stream rather than discarding it. *locate* is now blocked in a write. What the dispatcher now does is "unspecified", but it is clear that no further data can flow before *faninany* reads from its secondary input stream.
2. *faninany* starts and issues a call to the dispatcher to wait until a record is available on any one (or more) of its input streams.
3. *faninany* is resumed, because there is a record available on its secondary input stream. We know that *locate* is blocked after it has written the first record to its secondary output stream. *spec* is probably waiting for an input record, but *locate* cannot produce one on its primary output stream while it is blocked waiting to complete the write on its secondary output stream.
4. *faninany* peeks at the record on its secondary input stream and writes it to its primary output stream. The record has now left the multistream part of the pipeline; *locate* is still blocked waiting for its output to complete.
5. The stage connected to *faninany*'s output stream consumes the record. *faninany* can now run; it consumes the record on the secondary input stream (which releases *locate*) and waits for the next input record on any of its input streams.
6. *locate* consumes the first input record and peeks at the second one, which does contain the specified string; *locate* writes that record to its primary output stream.
7. *spec* finally gets a record to work with. It builds the output record in a work area (copying the input record and appending the literal string) and writes the amended output record. Both *locate* and *spec* are now blocked waiting for their output to complete.
8. *faninany* is resumed; it peeks at the record, writes it, consumes it (which releases *spec*), and waits for the next record wherever it may appear.
9. *spec* is resumed; it in turn consumes the record (releasing *locate*) and peeks for some more to do.
10. *locate* consumes the second record, peeks the third, and writes it to the secondary output stream because it does not contain the specified string. *locate* is now blocked in a write; *spec* is blocked in a peek; again, only *faninany* can run.
11. *faninany* writes the third record, consumes it (releasing *locate*), and waits for the next one.
12. *locate* receives end-of-file on its input and returns. This severs its two output streams.
13. *spec* is resumed with an indication of end-of-file; it also returns, severing its output stream.
14. *faninany* is resumed with an end-of-file return code because all of its input streams are now severed; it returns as well.

We see that the output records arrive in the expected order, because peeking a record ensures that the producer cannot run and thus cannot produce another record while the first one is still in flight. The important piece of information is that *spec* does not delay the

record. This means that a record being processed by *spec* cannot be overtaken by another record that takes a different path.

So far we have met only built-in programs that do not delay the record; are there any that do? *sort* must of necessity see all input records before it can begin to produce output. However, a more subtle form of delay occurs with *unique* LAST, which compares pairs of records and discards duplicates, retaining the last of a set of identical records. Clearly, *unique* LAST must buffer one record internally while it examines the next one; as a result the output record is delayed relative to records that take a path that bypasses the *unique* LAST stage. Now, insert *unique* LAST instead of *spec* in the multistream example and run the file through it again:

Figure 18. Topology of Record Delay



Watch this happen:

1. The first record does not contain the specified string; it bypasses *unique* and becomes the first output record.
2. The second record is read into *unique*'s buffer; *unique* now peeks to see the next input record.
3. The third record bypasses *unique* as well. Thus it becomes the second output record.
4. Finally, *unique* receives end-of-file; it writes the record from its buffer, making it the third output record.

Note that even though *unique* has only a one-record delay, the effect of the pipeline topology may mean that many more records can overtake the one being buffered.

Thus, to reason about this in general, the concept of a *record delay* (or simply delay) is introduced. This delay is not the time it takes for a stage to process a record; rather, it represents a change in the relative order of records that pass through different paths in a multistream pipeline. If a record takes a path entirely through stages that do not delay the record, then the record must arrive at the end of the pipeline ahead of any record that enters the pipeline path after it does. If a record passes through a stage that delays records, then the record may arrive at the end of the pipeline later than a record that takes a path without delay.

Now consider this (apparently utterly redundant) REXX program to copy a file unchanged:

Figure 19. Null REXX Program

```
/* COPY REXX */
signal on error
do forever
   'readto record'
   'output' record
end
error: exit RC*(RC/=12)
```

The READTO pipeline command performs a consuming read; the producer is not blocked and can produce another record while the OUTPUT pipeline command in this REXX program is blocked in its write. Whether this actually happens depends on the implementation of

the pipeline dispatcher; this is deliberately unspecified. The copy program can be thought of as a one-step *elastic;* it can delay one record, but it needs not.

All *CMS Pipelines* built-in programs are written not to delay the record, except where required by the function they perform. A user-written stage might delay records if it processes them with a READTO-OUTPUT loop; it will not delay them if it processes them with a PEEKTO-OUTPUT-READTO loop.

Dynamic Change to Pipeline Topology

Earlier we saw a simple subroutine pipeline; this section describes CALLPIPE and ADDPIPE in the general case.

CALLPIPE suspends the stage that has issued the pipeline command and then runs a pipeline specification, usually with the suspended stage's streams connected to the subroutine pipeline. The suspended stage is resumed when all stages of the subroutine pipeline have completed and all connections that were changed by connectors have been restored. ADDPIPE, in contrast, creates a pipeline specification that runs in parallel with the stage that created it.

The argument string to ADDPIPE and CALLPIPE is a pipeline specification, which is processed by the scanner as described earlier. The new pipeline specification is often connected to one or more of the streams that were connected to the stage that issued the command to create a pipeline specification (though it needs not be) through one or more *connectors*. A connector may be specified at the beginning of a pipeline or at the end of a pipeline, but not in the middle of one. That is, the connector is at the beginning or the end of the pipeline specification or adjacent to an end character.

A connector ends with a colon. In general, it has three components that are separated by periods:

- An asterisk (*) to distinguish a connector from a label.
- A keyword to specify whether the stream to connect is an input stream or an output stream.
- A number or a stream identifier to specify the particular stream to be connected. An asterisk specifies the currently selected stream.

The second or the third component of a connector (or both) can be omitted. The default is the currently selected input stream when the connector is first in a pipeline; the default is the currently selected output stream when a connector is at the end of a pipeline.

Connectors specify how streams in the new pipeline are to be connected to the streams that are connected to the stage that issues CALLPIPE to create a subroutine pipeline. Streams that are connected to the stage's input streams can be connected to input streams in the subroutine pipeline, and streams that are connected to the stage's output streams can be connected to output streams in the subroutine pipeline. None, some, or all of the streams that a stage is connected to can be redefined in this way; the corresponding streams in the stage become temporarily unconnected, but this causes no problems, because the stage will be suspended while the subroutine pipeline runs. Here are some examples of CALLPIPE with connectors:

Figure 20. Connectors with CALLPIPE

`callpipe *: | ...`
`callpipe *.input: | ...`
`callpipe *..*: | ...`
	The currently selected input stream will be connected to the beginning of these pipelines.

`callpipe *.input.1: | ...`	The secondary input stream will be connected to the beginning of this pipeline (irrespective of which input stream is currently selected).
`callpipe ... | *..new:`	The output stream identified by `new` will be connected to the end of this pipeline (irrespective of which output stream is currently selected).

Recall that end-of-file can travel in both directions in a pipeline. End-of-file can traverse the connector from the outside into the subroutine. For example, when a stage on the outside severs an output stream that is connected to an input stream in the subroutine, the stage in the subroutine sees end-of-file as on a normal connection. But end-of-file never propagates out through a connector; instead, the original connection to the stage that issued the CALLPIPE pipeline command is re-instated when end-of-file reaches a connector from the inside of a subroutine pipeline. That is, the connection is handed back to the calling stage; it cannot be accessed until the subroutine has completed.

Let us now consider ADDPIPE. It also passes a pipeline specification to the scanner, but a stage that has issued ADDPIPE can resume as soon as the scanner has handed the pipeline specification to the dispatcher; the return code on ADDPIPE is non-zero only if there are syntax errors in the pipeline specification. There are two new ways that streams can be connected when a pipeline specification is issued with ADDPIPE:

Figure 21. Connectors with ADDPIPE

`addpipe ... | *.input:`	The currently selected input stream is disconnected; the new pipeline will be connected to the currently selected input stream.
`addpipe *.output: | ...`	The currently selected output stream is disconnected; the new pipeline will be connected to the currently selected output stream.

The connectors described for CALLPIPE (which are called *redefine* connectors) are also allowed with ADDPIPE, but these are useful only when you wish to propagate end-of-file outwards through a connector, because the change is permanent; the stream is not restored when end-of-file travels out of the new pipeline specification.

The new connectors are called *prefix* connectors, because they allow a stage to prefix itself with a new pipeline. Prefix connectors were designed to provide a way to read from an alternative source temporarily, for example to process Script `.im` control words or `%include` statements in a PL/I program. To support nested imbeds, more than one ADDPIPE command can be issued to prefix a particular input stream; the stream that is replaced is pushed on a stack.

Though the name reflects its heritage, a prefix connector is just as useful on the output side, for example to store records in several files (for example, the reverse of an include

processor). We shall, however, constrain ourselves to prefix connectors on the input side in this paper.

The pipeline command SEVER INPUT severs the currently selected input stream and pops the stream at the top of the prefix stack, if there is one; SEVER OUTPUT performs a similar operation on the output side.

To illustrate, assume that this pipeline is running:

```
pipe < pipejarg script | jdotim | > alljarg script a
```

jdotim looks for records beginning with `.im`; when it finds such a record, it reads the referenced file using an ADDPIPE pipeline command:

Figure 22. JDOTIM REXX, a Simplistic Include Processor

```
/* JDOTIM REXX -- Simplistic imbed                                  */
/*                               John Hartmann  23 Feb 1992 14:18:21 */
signal on novalue
level=1                               /* Doing main file            */
do forever
   signal off error                   /* Allow errors               */
   'peekto in'                        /* Read next line             */
   signal on error                    /* Catch errors               */
   select                             /* Decode                     */
      when RC=12                      /* End-of-file                */
         Then
            Do
               level=level-1          /* Decrement nesting level    */
               If level=0             /* Done?                      */
                  Then leave          /* Yes, this is eof on main   */
               'sever input'          /* Back to previous file      */
               'readto'               /* Consume the .im record     */
            End
      When RC/=0                      /* Some other error?          */
         Then exit RC                 /* Get out quick if so        */
      When left(in, 4)/=='.im '       /* An imbed record?           */
         Then,                        /* No, get to one:            */
            'callpipe *: | tolabel .im | *:'
      Otherwise                       /* Imbed a file               */
         parse var in . fn .          /* Get filename               */
         If left(fn, 1)/='('          /* Explicit file type         */
            Then file=fn 'script'     /* No, take default           */
            Else parse var in '(' file ')'          /* Get full name */
         'addpipe <' file '| *.input:'/* Read this file             */
         level=level+1                /* Bump nesting level         */
      End
end
error: exit RC*(RC<>12)
```

This program shows some of the strings that a pipeline REXX programmer has to her bow.

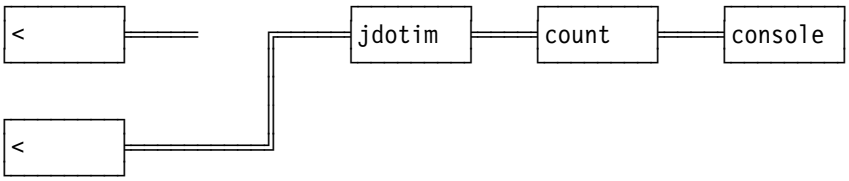
jdotim searches the input file for imbed control words. When it meets such a control word, it stops reading the current input file and starts reading the file to be imbedded. When it comes to end-of-file, it restores the file that it was reading when it met the imbed and then goes on looking for further imbeds. The point is that *jdotim* also looks for imbeds in files that it imbeds.

The loop begins with a PEEKTO pipeline command to peek at the next input record; the status is decoded in a Select instruction:

- Return code 12 means end-of-file. The current input stream is discarded with SEVER; this restores the stream at the top of the stack of streams that have been temporarily suspended by the ADDPIPE pipeline command below. The nesting level is decremented, making it zero when the end-of-file return code is for the original input stream. This, in turn, terminates the loop.
- Any other non-zero return code indicates an error; the program terminates.
- A record that does not begin with the desired Script control word is just a plain record in the file; a subroutine pipeline is called to pass the input directly to the output until an imbed is met or until end-of-file. (Recall that *tolabel* stops without consuming the record when it sees one that begins with the specified string, including the trailing blank.)
- An imbed is decoded and a pipeline is added by the ADDPIPE pipeline command to read the file into *jdotim*. The nesting level is incremented.

When the first imbed is met and the ADDPIPE pipeline command has completed, the pipeline topology is this:

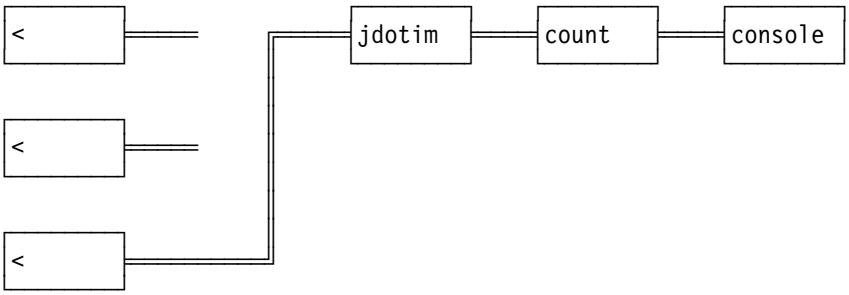
Figure 23. Topology after ADDPIPE



The topmost **<** is not blocked in its OUTPUT pipeline command, but it usually makes little difference when this record is consumed. Even if **<** got to run, it would become blocked as soon as it produced a record; and it would stay that way until its output stream was connected back to *jdotim*.

This is the topology when there is an imbed in an imbedded file:

Figure 24. Topology after Nested ADDPIPE



When end-of-file is reflected at the end of the second imbed file, *jdotim* severs its input stream (which is no longer connected). This restores the stacked input stream; the topology is back to the one shown in Figure 23. In a similar way, the connection to the original input file is restored when the first imbedded file comes to end-of-file.

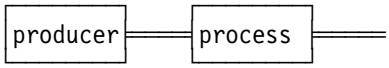
Let us convince ourselves that this does work at least in a fashion by running it against this paper:

```
 pipe < pipjarg script | count bytes words lines | console
►278 53 22
►Ready;

 pipe < pipjarg script | jdotim | count bytes words lines | console
►114344 18214 2979
►Ready;
```

ADDPIPE supports another connector configuration, known as a *hybrid* connector. Using both a redefine and a prefix connector for the same side, a stage can insert a pipeline segment between itself and a stage it is connected to. Assume this is the initial topology:

Figure 25. A Producer and a Process

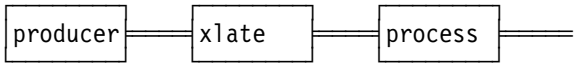


If the *process* stage issues this pipeline command:

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

Then the result is that the *xlate* stage is inserted in front of it:

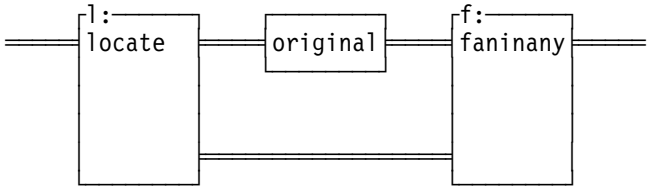
Figure 26. After the Hybrid



A similar configuration can be used to insert a pipeline segment on the output side of a stage. A stage even can do this with both its input and its output simultaneously, as in this configuration, where records that are 10 bytes or longer are shunted directly to the output, while records shorter than 10 bytes are processed by the original stage:

```
'addpipe (end ?)'
   ' *.input:  | l: nlocate 10 | *.input: ',
   '?*.output: | f: faninany   | *.output:',
   '?l: | f:'
```

Figure 27. After a double Hybrid ADDPIPE



We now know enough to define formally the terms *pipeline set*, *pipeline specification*, and *pipeline*.

A *pipeline set* is established by the PIPE command. Initially, it consists of the pipeline specification that is the argument to PIPE.

A *pipeline specification* consists of one or more pipelines, separated by end characters. A pipeline specification that is issued with ADDPIPE or CALLPIPE is added to the current pipeline set. Pipeline specifications issued with PIPE run as separate sets; the previous set is resumed when a PIPE command completes.

A *pipeline* consists of stages and label references, separated by stage separators. When ADDPIPE or CALLPIPE adds a pipeline specification to the set, a connector is optional at

either end of a pipeline; a pipeline that has two connectors needs not have stages or label references; a pipeline with only two connectors shorts an input stream to an output stream (it is a *short-through* pipeline).

There is one more way to define pipelines dynamically: the *runpipe* control stage. It reads input records and issues them as pipeline specifications in their own pipeline set. The pipeline set containing the *runpipe* stage is suspended until the new pipeline set completes or issues a message. When it issues a message, the new pipeline set is suspended and the message is written by *runpipe* to its output stream. Control reverts to the new pipeline set when the message is consumed in the original pipeline set.

Another control stage, *pipcmd*, issues the records it reads as pipeline commands without going through a REXX command environment. It is used, for example, under the covers of *getfiles*, where a subroutine pipeline turns the file identifiers into CALLPIPE pipeline commands that read the files; these pipeline commands are then passed to *pipcmd* to be run.

Commit Levels

Recall that the scanner checks the syntax of the arguments to built-in programs and abandons the complete pipeline specification if any one syntax check fails. The syntax check is based on information in the program descriptor, possibly augmented by a call to a syntax routine in a built-in program. Several types of errors, however, cannot be detected by the syntax check:

- REXX programs have no program descriptor; the scanner cannot check the syntax of the arguments to REXX pipeline programs.
- Programs that wish to ensure that a particular stream is connected (or not) cannot determine this at the time of the syntax check, because the connections are not connected until the pipeline specification is passed from the scanner to the dispatcher.
- Programs that use a resource that needs to be de-allocated explicitly cannot allocate such a resource in their syntax routine because they will not be informed if the pipeline specification is abandoned.

The pipeline notion of a *commit level* was designed to address these problems. For a REXX program, the effect is that the part of the program up to the first read or write operation acts logically as a syntax checking routine; if the REXX program returns with a non-zero return code without having performed a read or a write, then the complete pipeline is abandoned. This also caters for a syntax error in the first or only subroutine pipeline that is issued by a REXX program (still assuming it has performed no read and no write).

In general, *CMS Pipelines* commit levels allow programs to allocate resources assured that all are successful or all fail. This is how it works:

The starting commit level for a program is specified in its entry in the entry point table where it is resolved or in the program's descriptor; REXX programs start at commit level -1 by default. The initial commit level for a pipeline specification is the minimum of the starting commit levels for its stages (which are between -2147483647 and +2147483647 inclusive, but to be practical it is usually non-positive). During the lifetime of the pipeline specification the commit level can increase, but it can never decrease.

In any pipeline specification, the dispatcher runs only stages that are at the lowest commit level; other stages have either not yet started (because their starting commit level is higher than the current one for the pipeline specification) or they have performed a COMMIT operation to be suspended until the pipeline specification reaches a specified commit level. The stages at the lowest commit level must eventually commit to a higher level or return to the dispatcher. When all stages at the lowest commit level have committed to a higher level

or returned, the dispatcher raises the pipeline specification's commit level to the next level where a stage is waiting to start or commit.

Take this pipeline as an example:

```
pipe < input file | locate /pipe/ | rexx yourprog
```

It reads an input file, selects lines, and passes the lines on to one of your REXX programs. The three stages are arranged like this when the pipeline specification is handed from the scanner to the dispatcher:

Figure 28. Initial Commit Levels

```
 0: | <
-1: |                          yourprog
-2: |             locate
```

< starts at commit level 0, *locate* starts at commit level -2, and the REXX program starts at commit level -1. The dispatcher sets the current commit level to -2 and starts *locate*, which checks its secondary input stream to make sure it is not connected. On seeing that it is not connected, *locate* commits to level 0:

Figure 29. Commit Levels after First Commit

```
 0: | <           locate
-1: |                          yourprog
```

Your program is then started. Presumably it reads an input record; this causes an implied commit to level 0. All stages are now at commit level 0 and can call the dispatcher to transport data.

Recall that the dispatcher maintains an aggregate return code as stages return. When the commit level for a pipeline specification is raised, the stages that are waiting to commit at this new level receive the aggregate return code as the return code on their COMMIT pipeline commands, allowing them to ensure that all is still well. If the aggregate return code is zero, then the stages that start at the new commit level are, in fact, started (that is, made dispatchable); they are abandoned if the aggregate return code is non-zero.

To illustrate a stage that detects an error and exits instead of committing, consider this pipeline:

```
pipe < input file | l: locate /pipe/ | l: | rexx myprog
```

locate is dispatched at commit level -2 as before; when it now finds that its secondary input stream is connected, it issues a message and terminates with a non-zero return code. This causes the remaining stages to be abandoned.

Clearly, two stages in a pipeline specification can pass data between themselves only when they have reached the same commit level; the pipeline will stall if a program tries to read or write a stream that is connected to a stage that is suspended waiting to commit. *CMS Pipelines* has adopted the convention of passing data at commit level 0. Most built-in programs start at commit level 0; those that start at a negative commit level commit to level zero before transmitting data; they exit without passing data and without causing permanent changes if the COMMIT return code is non-zero.

Now consider a REXX filter that issues a subroutine pipeline. Assume that the REXX filter (the caller) is still at commit level -1 when it issues the CALLPIPE pipeline command to run the subroutine pipeline. The caller must now wait for the subroutine to complete, but if the subroutine is connected to the caller's pipeline, then the pipeline set would appear to be deadlocked, the calling pipeline being on commit level -1 and the subroutine trying to commit to level 0. Thus, to avoid a stall (which is explained in the next section) when a

subroutine pipeline increases its commit level (for instance to zero to transport data), it is ensured that the stage that created the subroutine is at the new commit level for the subroutine, or higher. If not, the caller commits to the higher level (while remaining suspended in the CALLPIPE pipeline command); this can cause the grandfather to commit, and so on.

In contrast, the other ways in which one can issue a pipeline specification are not susceptible to such a deadlock. Although a pipeline set that invokes another pipeline specification with the PIPE command or the *runpipe* built-in program become suspended while the new pipeline specification runs (with *runpipe* the two take turns at running as the new pipeline specification produces messages—PIPE suspends the issuing stage until the pipeline specification has run to completion), there is no danger of a deadlock between the added pipeline specification and the issuing stage, because the two pipeline specifications are not connected to one another. Pipeline specifications issued with the ADDPIPE pipeline command do not *co-ordinate their commit* with the issuing pipeline specification; though they may be connected to the pipeline set that issued the ADDPIPE pipeline command, the stage that adds the pipeline specification continues to run and can commit to level zero in its own good time. Thus PIPE, *runpipe*, and ADDPIPE need not interact with the commit level of the stage that issues them.

REXX pipeline programs that cause permanent change should first use the COMMIT pipeline command to ensure that all is still well:

```
/* LOCATE REXX */
'streamstate input 1'                /* Ensure this is not connected */
if RC>=0 & RC<12
   then do; Say 'Secondary input stream is connected'; exit 36; end
'commit 0'
if RC¬=0 then exit 0
/* Now process data */
```

This program uses the STREAMSTATE pipeline command to determine whether the secondary input stream is connected or not. It issues a message and exits with return code 36, while still at commit level -1, if the stream is connected. The program is then prepared to process data; it commits to zero and terminates “quietly” if the return code on the COMMIT pipeline command is non-zero. (It gives return code zero because it discovered no error itself.)

When a stage commits to zero it is assured that all stages are prepared to process data; it can raise its commit level to 1 to determine whether all data were processed correctly. Consider a REXX program to send a job to z/OS:

```
/* TOMVS REXX */
address command 'CP TAG DEV 00D CPHMVS1 SYSTEM 0'
address command 'CP SPOOL 00D TO RSCS CLASS A NOHOLD NOCONT PURGE'
'callpipe *: | punch | *:'            /* Punch file and pass it on   */
r=RC                                  /* Save return code            */
If RC=0                               /* OK So far?                  */
   Then                               /* Yes:                        */
      Do
         'sever output'               /* Disconnect output           */
         'commit 1'                   /* Now commit to 1             */
      End
If RC/=0                            /* Trouble with punch or others? */
   Then 'callpipe CP SPOOL 00D PURGE' /* Flush the file              */
   Else address command 'CP CLOSE 00D'/* Send file to RSCS           */
exit r
```

The subroutine pipeline fails if SPOOL is full, but what if a stage to read a disk file fails, possibly with the dreaded “Error 3 reading file”? Such an error does not affect the return

code from the subroutine, but it will cause a non-zero return code from COMMIT 1, and thus the SPOOL file can be purged rather than being transmitted as an incomplete job.

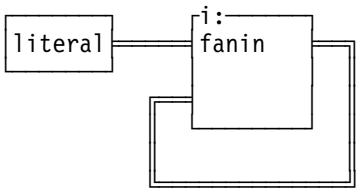
You might wonder why one could not simply test the return code from the complete pipeline in whatever program set it up and purge the SPOOL file there if the pipeline as a whole failed. First, this means that the pipeline cannot be issued directly from the terminal; but second, and more importantly, all EXECs that use TOMVS would then have to understand about SPOOL files. When commit levels are used as they are here, only one REXX program needs to know about SPOOL files and how to purge them.

The examples in this section have, naturally, emphasised commit levels; thus it may be helpful to point out that a REXX pipeline programmer can ignore commit levels most of the time; *CMS Pipelines* takes care of the cases where no explicit commit is performed: an automatic COMMIT to zero is performed when a REXX program issues a PEEKTO, a READTO, an OUTPUT, or a SELECT ANYINPUT pipeline command without having performed an explicit commit operation. As always, however, the programmer is in complete control; the program can issue the NOCOMMIT pipeline command to indicate that it will manage its commit level and wishes to read or write at commit level -1 (presumably on a stream that is connected with an ADDPIPE prefix connector).

Stall

A pipeline is stalled when no stage can run and no stages are waiting for external events, but not all stages have terminated. A stall can occur only if at least one stage of a pipeline specification has secondary streams. A stall is easily provoked; for example, this two-stage pipeline stalls:

Figure 30. Stalled Pipeline, Contrived



The primary output stream from *fanin* is fed back to *fanin*'s secondary input stream. The corresponding command and response are:

Figure 31. How to Stall

```
 pipe literal abc | i: fanin | i:
►Pipelines stalled.
►... Issued from stage 2 of pipeline 1.
►... Running "fanin".
►Ready(-4095);
```

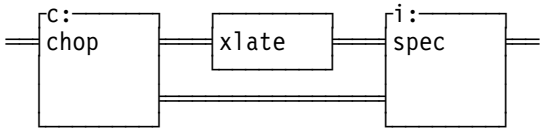
The label reference at the end of the pipeline specification defines *fanin*'s secondary streams; its primary output stream is connected to its secondary input stream.

fanin works by first passing all records on the primary input stream to the primary output stream; it then passes all records on the secondary input stream to the primary output stream, and so on. In this example, it peeks at the input record on the primary input stream and writes it to the primary output stream. The dispatcher blocks *fanin* (makes it non-dispatchable) because it is waiting for an output record to be consumed. But it is the very same stage that must consume the record, and (being non-dispatchable) *fanin* cannot possibly consume an input record to satisfy its own write. The pipeline dispatcher now tries to find a stage to run to consume the record, but cannot find one; the pipeline is

stalled. To continue, the pipeline dispatcher then severs all connections and makes all stages dispatchable with return code -4095.

In real life it may be more difficult to diagnose the cause for a stall. One might try this subroutine pipeline to uppercase the word in the left margin of each input record:

Figure 32. Stall with Chop



An attempt could be[1]:

Figure 33. The Stalling Pipeline with Chop

```
/* BADUPW REXX -- *BAD* Uppercase word in left margin */
signal on novalue
trace off
'callpipe (nomsg 6 end ?)',
   '|*:',
   '|c: chop blank',               /* Truncate after the first word */
   '|   xlate upper',                /* make it uppercase           */
   '|i: spec 1-* 1 select 1 1-* next',/* rebuild record             */
   '|*:',
   '?c:',                           /* Rest of record              */
   '|i:'                            /* Into SPEC                   */
exit RC
```

chop writes the beginning of each record up to but not including the first blank to the primary output stream; it then selects the secondary output stream and writes the remainder of the record including the first blank. Either output record can be null. *spec* can have multiple input streams. The word after the SELECT keyword specifies which input stream is to be used for subsequent specification items. When *spec* uses multiple streams, they are all peeked before the output record is built. As used in this example, *spec* peeks a record on the primary input stream, then selects the secondary input stream and peeks a record. It then copies the record from the primary input stream into its buffer, appends the record from the secondary input stream, and writes the output record.

This is what happens:

Figure 34. Stall with CHOP and SPEC

```
 pipe (nomsg 6) literal abc def | badupw | console
►Pipelines stalled.
►... Running "console".
►Ready(-4095);
```

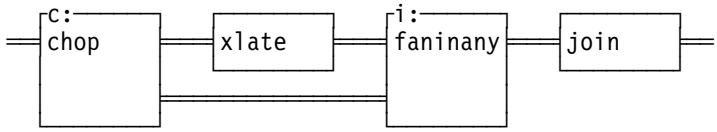
The problem is that *chop* produces a record on two output streams one at a time, but *spec* requires both of them to be available at the same time, and *xlate* does not delay the record.

There is a specific cure to this problem where a record is split and needs to be joined later, and a general one. The specific cure is to use *faninany* to get the parts of the original

[1] The option NOMSG disables the message “Issued from ...”.

record back into a single stream; *join* can then be used to put the original record back together again, rather than *spec*:

Figure 35. Special-case Cure for Stall

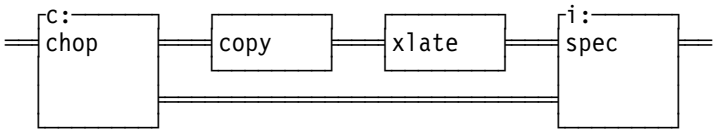



```
/* UPW1 REXX -- Special-case non-stalling */
signal on novalue
trace off
'callpipe (end ?)',
   '|*:',
   '| c: chop blank',                 /* Take label                 */
   '|    xlate upper',                /* Uppercase it               */
   '| i: faninany',                  /* Merge labels and rest      */
   '|    join',                       /* Rebuild original record    */
   '|*:',
   '? c:',                           /* Rest of record...          */
   '| i:'                            /* ... bypasses xlate         */
exit RC
```

```
 pipe literal abc def | upw1 | console
►ABC def
►Ready;
```

The general method to break a stall caused by a stage being unable to produce several output records at the same time is to insert *copy* stages in the critical paths (so the program in Figure 19 on page 21 is after all not completely redundant):

Figure 36. General Case Cure for Stall




```
/* UPW2 REXX -- Uppercase word in left margin */
signal on novalue
trace off
'callpipe (end ?)',
   '|*:',
   '| c: chop blank',                 /* Get label                  */
   '| copy',                         /* One-stage elastic          */
   '| xlate upper',                  /* Uppercase it               */
   '| i: spec 1-* 1 select 1 1-* next',              /* Join pieces */
   '|*:',
   '? c:',                           /* The rest of the record     */
   '| i:'                            /* ... is passed to spec      */
exit RC
```

```
 pipe literal abc def | upw2 | console
►ABC def
►Ready;
```

Because *copy* performs a consuming read before it writes the output record, *chop* is unblocked and thus is able to write to its secondary output stream in time for *spec* to peek both input records.

Although only a multistream pipeline can stall, this is not to say that even all single-stream pipelines eventually complete. A single straight pipeline can have a stage that goes into a loop, producing no output and consuming no input, in which case the pipeline will never complete. The immediate command HX can break such a loop. When the pipeline contains REXX programs, the immediate command HI may be able to terminate those stages and then cause the pipeline to stop.

A pipeline can also contain stages that wait on external events that never occur, for example an infinite delay. There are also device drivers (*starmsg* and *immcmd*) for which end-of-file is never reflected from the host interface. The immediate command PIPMOD STOP can be used to terminate such stages.

In Retrospect

Over the years, function has been added to *CMS Pipelines* to tackle problems that had appeared to be unsolvable the previous year. Much prototyping has been done to determine the best way to implement new concepts; old concepts that turned out to be in the way of progress were ruthlessly expunged, once even from code that had shipped to customers. It always turned out that to solve a "hard" problem one needed a general solution, sometimes requiring concepts that were "orthogonal" to the then current implementation. It often turned out that these general solutions could cope with problems that had not even been thought of when the original pain was treated:

- Multistream pipelines were added to support the *update* program. It was soon realised that this provided a powerful way for selection stages to go about their tasks, leading to what is now known as decoding networks, and the ability to reason about the relative order of records as they flow through a multistream network.
- Dynamic reconfiguration "fixed" the problem of nested include files and then turned out to provide a general mechanism by which a program can react to its input data by *functional programming* instead of procedural code.
- A syntax check implemented in the scanner rather than in the actual stage prevented embarrassment in situations where, for instance, *console* would want a record, only to discover that the following stage had failed. Having a program descriptor motivated a change to allocate work areas for all stages in contiguous storage before checking syntax; this removed most storage management issues from the individual programs, and greatly simplified the ports to z/OS and GCS.
- The ability to determine the status of a stream took care of the embarrassment when the output from a device driver that consumes the resource it provides access to (such as *xedit*) is severed because a partitioning stage has consumed all the records it wants. Such device drivers now try to read from the host interface only when the following stage is waiting for a record.
- Commit levels brought syntax checking to REXX programs, as well as allowing built-in programs that use multiple streams to ensure that their topology is acceptable before committing to transport data. The support for commit levels is so new that it is not yet completely explored.
- The notion of summarily stopable stages was introduced to make storage management more reliable, because the pipeline dispatcher would take care of releasing buffers. However, this was soon extended to allow faster propagation of end-of-file backwards through filters and selection stages.

Except for the non-trivial problems of high-level language support and one remaining nagging problem with the support for commit level, there seems to be little more needed in the core of *CMS Pipelines*; in fact, it has a few as yet unexplored features that were put there quietly with the future in mind.

But, then again, *CMS Pipelines* has looked complete every step of the way. The plumbing community will no doubt continue to find areas that need improvement and to make its needs known. And answering these needs will no doubt bring unanticipated benefits.

One big challenge will be to run pipelines across platforms and on different types of systems, possibly even interconnecting with POSIX files and pipes.

Appendix

We describe a few terms in general usage that have a specific meaning in the context of *CMS Pipelines*.

Blocking

This term has two usages:

1. Records are blocked (in the OS sense) when several logical records are written as a physical block (hence the *block* built-in program, which does precisely this).
2. The pipeline dispatcher *blocks* a stage when the dispatcher cannot perform a requested function immediately; some other stage must run, for example, in order to produce a record for the blocked stage to consume.

Bytes and Characters

These two terms are used interchangeably for the unit of information inside *CMS Pipelines* (eight bits). Usage is that characters are displayed on a screen or a printed page, never bytes.

Records and Lines

A record is the unit of information passed between stages by the pipeline dispatcher. The terms records and lines are used interchangeably, with a tendency towards lines for those records that originated on the terminal or contain text; binary data are usually referred to as records.

A record is comprised of zero or more bytes. A record containing zero bytes is a *null record*.

Columns and Positions

CMS Pipelines refers to positions in a record relative to the first or last byte. Thus the first byte is in position (or column) one, the last byte is column -1, and so on. *Input ranges* can be specified as column ranges, word ranges, field ranges, or substrings of such ranges. The facilities are mindbending.

File

In general, a file is an ordered collection of records. That is, there is a first record, a second record, and so on. Except for the relative ordering implied by the sequence of records, the file needs not be sorted in any way.

A *CMS file* is a named collection of records stored on a CMS minidisk or in the Shared File System (SFS).

SPOOL files, on the other hand, are managed by CP. There are several types of SPOOL files; the most common ones are reader files, print files, and punch files.

The concept of a file, in general, is vague.

It is often convenient to speak of the *input file* for a filter, meaning the collection of records that will be (or could be) presented to the filter on its input stream. It is understood that the origin of this collection of records needs not be any of the file objects described previously in this section. The records could, for instance, come from a set of variables in a REXX or EXEC2 environment; they could be the result of an SQL query; or they could be generated internally in the pipeline. Note that such a file does not exist as an object in time or space.

The motivation to call such a collection of records a file, even though no tangible file exists, is this: Often one tests a filter by reading a test case from a disk file, passing it through the filter, and writing the result to disk. In this case the input and output files do exist, though they are not being read and written directly by the stage being tested.

Thus, it is important to remember that when *file* is used to designate the collection of records going into (or out of) a stage, it never refers to the CMS file that is being read seven stages earlier in the pipeline, nor does it refer to the SPOOL file that will be written five stages downstream; it is merely a way to refer to the input (or output) records collectively.

End-of-file

In common usage, end-of-file is received when a device reaches the end of the set of records comprising the file it is reading.

For *CMS Pipelines*, end-of-file means that no further data can flow through a stream because the stream has been *severed;* that is, the stream is no longer connected to another stage. A stage may explicitly sever a stream, but in most cases the output streams for a stage are severed by the pipeline dispatcher when the stage has run to completion and has returned to the pipeline dispatcher.

CMS Pipelines extends the traditional concept of end-of-file to include not only end-of-file on input (such as reaching the end of a deck of cards in a card reader) but also end-of-file on output. (Think, for example, of writing to a tape that has reached the end of the reel.)

End-of-file is also reflected to a stage when it attempts an output operation to a stream that is not connected. This means either that the stage is the last of a pipeline (and thus the stream was never connected) or that the stage that was connected to the stream has terminated or has severed its corresponding input stream.

Buffering

Many *CMS Pipelines* built-in programs process an input record and generate an output record without reference to any other records. *sort*, on the other hand, must of necessity read all input records before it can write any output record. Thus, *sort* is said to *buffer* the entire file. Other programs (for instance *take* LAST) buffer a subset of the file.

In OS parlance, on the other hand, a buffer is a storage area into which the access method stores a record or from which it writes a record. The *buffer* built-in program uses a buffer in this sense of the word to hold the entire file; other built-in programs may buffer a record at a time.