z/VM

IBM

CMS Pipelines User's Guide

version 6 release 1

z/VM

CMS Pipelines User's Guide

version 6 release 1

> **Note:**
> Before using this information and the product it supports, read the information in "Notices" on page 285.

Contents

Figures

ix

Tables

About This Document

This document describes how to use IBM® z/VM® CMS Pipelines. CMS Pipelines provides a rich and efficient set of functions that can be used to solve complex problems without requiring a program, and it lets you write REXX execs without device interface dependencies.

Intended Audience

This document is for anyone who wants to learn how to use CMS Pipelines and has not used CMS Pipelines before or has had some experience with CMS Pipelines and wants to gain more knowledge.

Before using this document, you should be familiar with CMS. You should know how to enter CMS commands. It is also helpful, though not necessary, to know how to write simple REXX execs. You do not need to have system programming experience.

Where to Find More Information

For more information about z/VM, see the "Bibliography" on page 291 for a list of related publications.

Links to Other Online Documents

If you are viewing the Adobe® Portable Document Format (PDF) version of this document, it might contain links to other documents. A link to another document is based on the name of the requested PDF file. The name of the PDF file for an IBM document is unique and identifies the edition. The links provided in this document are for the editions (PDF names) that were current when the PDF file for this document was generated. However, newer editions of some documents (with different PDF names) might exist. A link from this document to another document works only when both documents reside in the same directory.

How to Send Your Comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or give us any other feedback that you might have.

Use one of the following methods to send us your comments:
1. Send an e-mail to mhvrcfs@us.ibm.com
2. Visit the z/VM reader's comments Web page at www.ibm.com/systems/z/os/zvm/zvmforms/webqs.html
3. Mail the comments to the following address:
   > IBM Corporation
   > Attention: MHVRCFS Reader Comments
   > Department H6MA, Mail Station P181
   > 2455 South Road
   > Poughkeepsie, NY 12601-5400
   > U.S.A.
4. Fax the comments to us as follows:
   > From the United States and Canada: 1+845+432-9405
   > From all other countries: Your international access code +1+845+432-9405

Include the following information:
- Your name and address
- Your e-mail address
- Your telephone or fax number
- The publication title and order number:
  > **z/VM V6R1 CMS Pipelines User's Guide**
  > **SC24-6170-00**
- The topic and page number related to your comment
- The text of your comment

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you submit to IBM.

If You Have a Technical Problem

Do not use the feedback methods listed above. Instead, do one of the following:
- Contact your IBM service representative.
- Contact IBM technical support.
- Visit the z/VM support Web page at www.vm.ibm.com/service/
- Visit the IBM mainframes support Web page at www.ibm.com/systems/support/z/

Chapter 1. Pipeline Basics

CMS Pipelines lets you solve big problems by combining small programs. It lets you do work that would otherwise require someone to write a new program. Often you get the result you need with a single CMS command. That command is PIPE.

The PIPE command accepts *stages* as operands. Many stages are included with CMS Pipelines. Some stages read data from system sources, such as disk files, tape files, and the results of z/VM® commands. Other stages filter and refine that data in some way. You can combine many stages within a single PIPE command to create the results you need.

Here is an example of a simple PIPE command. It counts the number of words in your ALL NOTEBOOK file and writes the result to your terminal. A word is anything surrounded by blanks:

```
pipe < all notebook | count words | console
```

Three stages are combined to do the work. The < (read file) stage reads the file, the COUNT stage counts the words, and the CONSOLE stage displays the count at your terminal.

Anyone familiar with CMS commands can use CMS Pipelines. You do not need to be a programmer. If you can describe what you want to do, chances are that you can use CMS Pipelines to do it. This book describes how to use CMS Pipelines. Read this chapter to learn pipeline concepts and to see what pipelines can do. Then use the rest of the book as you need it. The chapters can be read in any order.

If you are a programmer, CMS Pipelines can save you some coding. Pipelines can be used in execs to replace device-dependent code (such as EXECIO). Often, several lines of code can be replaced with a single PIPE command. You might also consider writing your own stages. These *user-written* stages are programs that read from and write to a pipeline. User-written stages are device-independent and can be easily used by others.

The Significant Difference

Two important characteristics of CMS Pipelines distinguish it from other z/VM facilities:

- CMS Pipelines lets you combine programs so that the output of one program serves as the input to the next.
- CMS Pipelines includes many programs, referred to as *built-in stages*, that are ready for you to combine in pipelines.

What Is a Pipeline?

CMS pipelines are like the pipelines used in plumbing. Instead of water flowing through pipes, however, data flows through programs. Data enters the pipe from a device (such as a terminal or a disk), flows through the pipeline, and exits to another device. (See Figure 1 on page 2.)
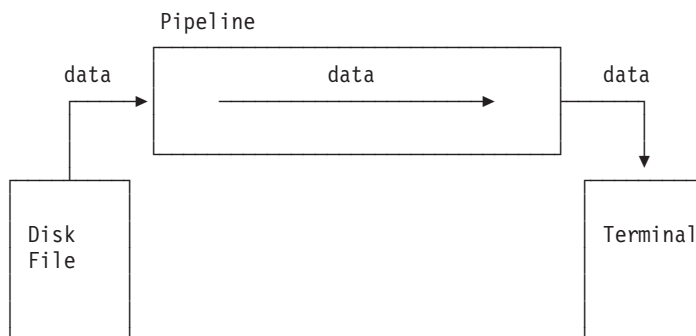
Figure 1. Data Flowing through a Pipeline

Programs, like pieces of pipe, can be fit together to solve complex problems. Each program, or *stage*, in the pipeline changes the data that flows through it. As data flows through the stages it is transformed, step-by-step, into the results you need. The data flows from left to right. (See Figure 2.)
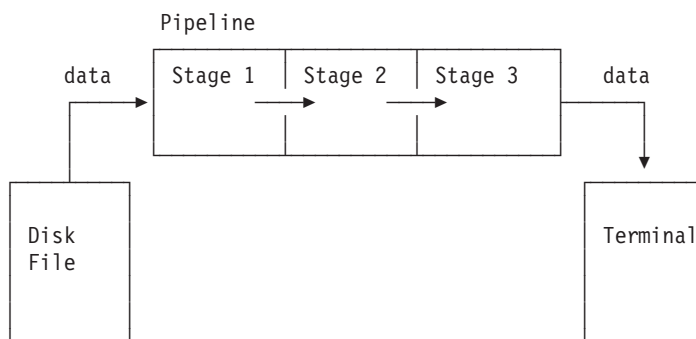
Figure 2. Stages within a Pipeline

CMS Pipelines includes many stages that you can use in your pipelines. Some stages move data into and out of the pipeline. Others filter or transform data within it. These stages may be sufficient for creating pipelines that meet your needs. If not, you can write your own stages.

User-written stages are REXX or Assembler programs that read data from the pipeline, work on the data, and place the data back in the pipeline. You can use user-written stages and built-in stages in the same pipeline. Because data is read from and written to the pipeline, stages are independent. Writing a stage is like creating a new water valve—because the pipe fittings are standard, others can use the valve in their plumbing.

Stages

In a pipeline, the output of one stage is the input to the next. The data itself is in the form of discrete records; that is, it is records that flow through the pipeline, not a continuous stream of bytes. A record is simply a string of characters—perhaps a line of a CMS file or a line entered at the terminal. Imagine a stage as a small factory through which a conveyor moves records. Records enter the stage on the left, and leave on the right. Figure 3 on page 3 shows input records before

processing on the left. The stage reads the records and processes them. The resultant output records, written by the stage, are shown on the right.

```
                        ┌──────────┐
                        │ Stage    │
┌─────────────────┐     │          │     ┌─────────────────┐
│ Input Record 1  │     │          │     │ Output Record 1 │
│ Input Record 2  │ ──▶ │          │ ──▶ │ Output Record 2 │
│ Input Record 3  │     │          │     │ Output Record 3 │
└─────────────────┘     │          │     └─────────────────┘
                        └──────────┘
```
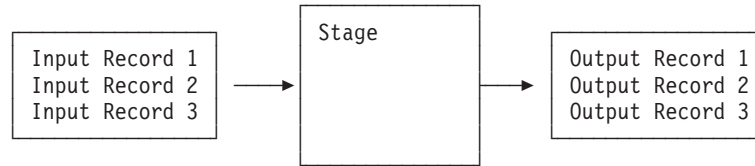
Figure 3. Records Flowing through a Stage

While within the stage, the records can be modified, discarded, or split apart. Practically anything can happen to them. Precisely what happens depends on the stage that is being used. Many stages write one output record for each input record. Some, however, do not.

Figure 4 shows a stage consisting of a CHOP stage. CHOP truncates records at a specified length. In the example, each record is truncated to a length of 5 characters. Like many stages, CHOP writes one output record for every input record.

```
Input records         ┌──────────┐      Output record
                      │ CHOP 5   │
┌──────────────┐      │ Stage    │      ┌──────────┐
│ BOB SMITH    │      │          │      │ BOB S    │
│ SUE JONES    │ ──▶  │          │ ──▶  │ SUE J    │
└──────────────┘      │          │      └──────────┘
                      │          │
                      └──────────┘
```
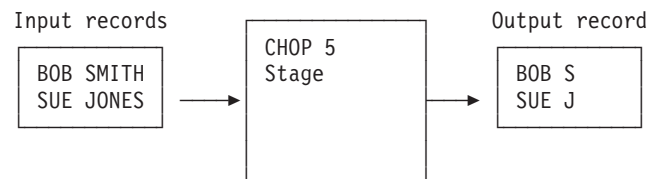
Figure 4. Records Flowing through a CHOP Stage

Figure 5 shows a stage consisting of a COUNT WORDS stage. Two records flow into the stage, but only one flows out. That single record contains the count of the number of words on all the records flowing into the stage.

```
Input records         ┌────────────┐    Output records
                      │ COUNT WORDS│
┌──────────────┐      │ Stage      │    ┌──────────┐
│ BOB SMITH    │      │            │    │ 4        │
│ SUE JONES    │ ──▶  │            │ ──▶└──────────┘
└──────────────┘      │            │
                      │            │
                      └────────────┘
```
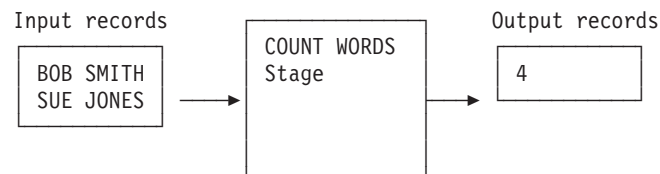
Figure 5. Records Flowing through a COUNT Stage

Figure 6 on page 4 shows another example. In this case the stage is LOCATE /BOB/. Again, two records flow into the stage. LOCATE looks at the content of each incoming record. If it contains the string BOB, LOCATE lets the record through.

```
Input records                                      Output record
                        ┌──────────────────┐
┌──────────────┐        │ LOCATE /BOB/     │      ┌──────────────┐
│ BOB SMITH    │        │ Stage            │      │ BOB SMITH    │
│ SUE JONES    │───────▶│                  │─────▶│              │
└──────────────┘        │                  │      └──────────────┘
                        └──────────────────┘
```
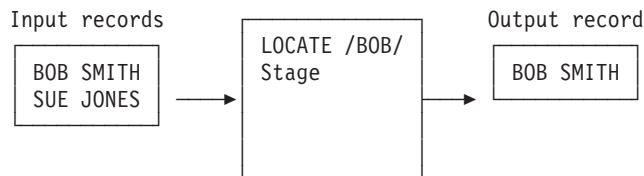
Figure 6. Records Flowing through a LOCATE Stage

The records entering a stage are called its *input stream*. The records leaving a stage are called its *output stream*. In the example in Figure 6, LOCATE reads all records from its input stream, but writes only the records containing BOB to its output stream.

Stages can use more than one input stream or output stream. You can use these *secondary streams* to write complex multistream pipelines. Multistream pipelines are described in Chapter 6, "Multistream Pipelines," on page 107. Until that chapter, let's work with simpler pipelines that use only one input stream and one output stream.

Figure 7 shows how records flow through several stages. The output of the LOCATE stage becomes the input to the COUNT stage.

```
                 ┌──────────────┐                              ┌──────────────┐
                 │ LOCATE /BOB/ │                              │ COUNT WORDS  │
                 │ Stage        │                              │ Stage        │
┌──────────────┐ │              │ ┌──────────────┐             │              │ ┌──────────────┐
│ BOB SMITH    │ │              │ │ BOB SMITH    │             │              │ │ 2            │
│ SUE JONES    │▶│              │▶│              │────────────▶│              │▶│              │
└──────────────┘ │              │ └──────────────┘             │              │ └──────────────┘
                 └──────────────┘                              └──────────────┘
```

Figure 7. Records Flowing through Multiple Stages

The LOCATE stage reads both records from its input stream (SUE JONES and BOB SMITH). It writes only the record containing BOB SMITH to its output stream. The COUNT stage reads records from its input stream. There is only one record: BOB SMITH. COUNT tallies the number of words in that record and writes a single record to its output stream. That record contains the number 2, which is the number of words in the record read by the COUNT stage.

The PIPE Command

To run a pipeline, use the CMS PIPE command. Like other CMS commands, PIPE can be entered from the CMS command line or from an exec. PIPE accepts one or more pipelines as operands. In the first few chapters of this book, the PIPE command operands consist of only a single pipeline. Multiple pipelines are described in Chapter 6, "Multistream Pipelines," on page 107.

In a pipeline, stages are separated by a character called a *stage separator*:

```
pipe stage_1 | stage_2 | ... | stage_n
```

Do not place stage separators after the last stage.

For the default stage separator, the PIPE command expects the character X'4F'. You must determine which key on your terminal generates the character X'4F'. It is a solid vertical bar (|) on American and English 3270 terminals. In some countries, this character is displayed as an exclamation mark (!). Some workstation terminal emulator programs map the solid vertical bar to the split vertical bar (¦). The solid vertical bar is the logical-or operator in PL/I and REXX programs. In a pipeline, it indicates where one stage ends and another one begins. If you aren't sure what character to use, create and run the exec in Figure 8.

```
/* STAGESEP EXEC */
say 'The stage separator is:' '4f'x'.'
exit
```

Figure 8. Finding the Stage Separator

There is no limit (other than available space on the command line) to the number of stages you can specify. For your first pipeline, try this PIPE command:

```
pipe < profile exec | count lines | console
```

If you do not have a PROFILE EXEC, substitute the name of any existing file. Be careful to leave a space after <. The number of lines in your PROFILE EXEC is displayed. If you make a mistake typing the command, an error message is displayed. Just type the command again, correcting the mistake. Figure 9 shows a map of the above pipeline.

```
Stage 1              Stage 2              Stage 3

┌──────────────┐     ┌──────────────┐     ┌──────────────┐
│ < profile exec│ ──► │ count lines  │ ──► │ console      │
└──────────────┘     └──────────────┘     └──────────────┘
       ▲                                          │
       │                                          ▼
┌──────────────┐                          ┌──────────────┐
│ PROFILE      │                          │ Terminal     │
│ EXEC         │                          │              │
│ file         │                          │              │
└──────────────┘                          └──────────────┘
```
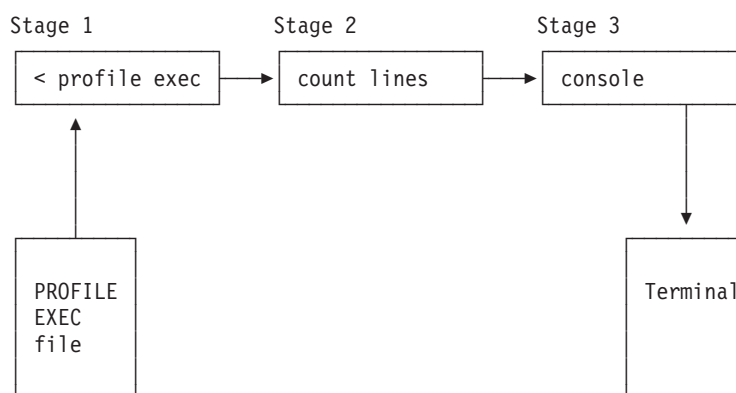
Figure 9. Map of Your First Pipeline

The example contains three stages. Each stage consists of a stage plus operands:

< profile exec
> The < (read file) stage reads a file and writes all of the file records to its output stream. As used here, the < stage reads your PROFILE EXEC and writes each record to its output stream.

count lines
> The COUNT stage counts items on the records it reads from its input stream. COUNT has operands that let you tell it what to count (such as bytes, words, or the records themselves). As used here, the LINES operand causes COUNT to tally the count of the input records. The records that COUNT reads are the records written by the < stage. (That is, the output of < becomes the input to COUNT.) So, COUNT tallies the count of records in

your PROFILE EXEC. Then COUNT writes a single record containing the tally to its output stream. COUNT writes only one record—it does not write the records from the PROFILE EXEC to its output stream.

`console`
> The CONSOLE stage either reads from your console or writes to it, depending on its position in the pipeline. As used here, the COUNT stage reads records from its input stream and displays them on the console. Only one record is in its input stream. That record, written by COUNT, contains the count of the lines in your PROFILE EXEC.

Notice that both < and CONSOLE work with devices. In the example, < reads a disk file and CONSOLE writes to the console. Stages that convey data between the pipeline and the outside world are called *device drivers*.

One advantage of CMS Pipelines is that you can use a different device by changing only one stage. To change the above example to write the count to a file instead of the console, just substitute a > stage for CONSOLE. Here's how:

```
pipe < profile exec | count lines | > yourfile data a
```

The > (write file) stage writes all records in its input stream to a file. Specify the file name as an operand. Remember to leave a space after the > symbol. The > stage creates a new file or replaces an existing file named YOURFILE DATA A. A file mode must be specified, as shown.

Device Drivers

Device drivers are stages that interact with devices or other system resources. The simplest pipelines consist of two device drivers. Data read from one device moves through the pipeline to the other device. For example, to copy data from a file to your terminal, enter the following command (change TEST DATA to the name of an existing file):

```
pipe < test data | console
```

The results are like those of the CMS TYPE command. Figure 10 shows a map of this pipeline.
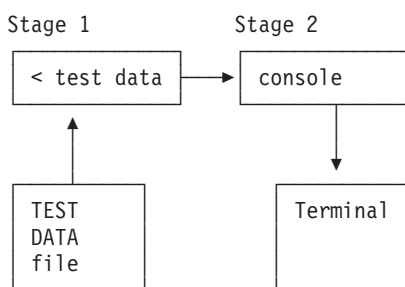
Figure 10. Map of a Pipeline with Two Device Drivers

The < stage reads the file TEST DATA and writes each record to its output stream. The output of the < stage is connected to the input of CONSOLE. CONSOLE reads the records from its input stream and displays them on the screen.

CMS Pipelines includes many device-driver stages. They work with tapes, printers, disk files, XEDIT data, the console, the reader, the program stack, REXX variables, and the system environments. Although not all of these are true devices, stages that work with them are called device drivers. Commonly used device drivers are described in Chapter 4, "Device Drivers," on page 59.

The device drivers used most often in this book are <, >, >>, CONSOLE, CP, CMS, and LITERAL. You have already seen some of these. Let's look at some examples of the others. The following pipeline uses the CMS device driver:

```
pipe cms listfile * * a | > yourlist data a
Ready;
```

The CMS device driver passes the LISTFILE command to CMS for execution and writes the results of LISTFILE to its output stream. (The results are not displayed on your terminal.) The second stage is a > device driver. The > device driver reads records from its input stream and writes them to the file YOURLIST DATA A.

The CP device driver works in a similar fashion. Use it to capture the responses of CP commands. In this example, the CP stage passes the string QUERY USERS to CP. The results are passed to the next stage, which writes them to a file named USERS DATA A.

```
pipe cp query users | > users data a
Ready;
```

LITERAL is a very useful device driver. It writes a string to its output stream. In this PIPE command, LITERAL writes the string Testing 1, 2, 3 to its output stream. The CONSOLE stage reads that record and displays it:

```
pipe literal Testing 1, 2, 3 | console
Testing 1, 2, 3
Ready;
```

How would you write the same string to a file? Substitute a > stage for CONSOLE.

The >> (append file) device driver adds records to the end of an existing file, or creates a file if it does not exist. The following PIPE command adds the record The End to the file USERS DATA A:

```
pipe literal The End | >> users data a
Ready;
```

Filters

Device drivers let you get data in and out of a pipeline. *Filters* are stages that work on data already in the pipeline. The COUNT stage used in the first example pipeline is a filter. It counts every record that flows into it from its input stream. Then it writes one record containing that count to its output stream. The LOCATE stage is also a filter. It examines the records from its input stream, looking for those that match a specified string. If the record matches, LOCATE writes the record to its output stream. LOCATE discards records that do not match. (See Figure 5 on page 3 and Figure 6 on page 4.)

Filters can do any function imaginable. Many filters are built into CMS Pipelines, but you can also code your own using the REXX language.

The filters supplied with CMS Pipelines do many functions of general use. For example, they select records based on the content of the record or on the position of the record in the stream flowing through the pipeline. They change or rearrange

records as they pass through. They can even sort records. Commonly used filters are described in Chapter 2, "Filters," on page 17.

Note: Stages are grouped into the categories *filter*, *device driver*, *host command interface*, or *other* for convenience. To the PIPE command, they are all just stages that happen to do different kinds of functions.

Specifying PIPE Options

In addition to an operand (consisting of one or more pipelines), the PIPE command accepts one or more *options*. PIPE options reassign the stage separator, trace execution of the pipeline, control the level of messages displayed, and so on. A complete list of the PIPE options is in the *z/VM: CMS Pipelines Reference*.

This book describes several PIPE options as the need arises. To show you how to specify a PIPE option, let's use the STAGESEP option. STAGESEP assigns the stage separator to a different character. By default, the stage separator is a vertical bar (|). To use a question mark as the stage separator, for example, specify `stagesep ?` as shown in Figure 11.

```
pipe (stgsubsep ?) literal one two three ? count words ? console
3
```

Figure 11. Specifying an Option on the PIPE Command

As shown, PIPE options should be enclosed within parentheses after the PIPE keyword. When specifying more than one option, separate them with blanks. The first stage of the pipeline begins immediately after the options.

Understanding Pipelines

New pipeline users often think that each stage of a pipeline processes all the records before writing results to the next stage. This is not true. Most stages process only one record at a time. CMS Pipelines controls when the stages run. It knows which stages have a record to process and which do not. The order in which stages run is unpredictable.

Because CMS Pipelines usually lets each stage process only one record at a time, only several records are in the pipeline at any moment—minimal virtual storage is used. This is an important characteristic of pipelines. It means that if you process a file containing one million records, you do not need to worry about having enough virtual storage to hold all those records.

There are times, however, when all the records are held in virtual storage. Some stages need to read all the records before they can do their work. For example, the SORT stage cannot sort the records in the pipeline until it has read all the records. After it sorts the records, it writes those records, one at a time, to its output stream. But, these filters are exceptions. They are said to *buffer the records*. Both this book and the *z/VM: CMS Pipelines Reference* indicate when a stage buffers records.

Pipelines in Execs

The PIPE command can be run from within an exec, just like any other CMS command. This section describes several different uses of pipelines within execs.

Preserving a Pipeline

One reason to use a pipeline within an exec is to save it so you can run it again. This simple REXX exec contains a comment, a PIPE command, and an EXIT instruction:

```
/* WORDS EXEC counts the number of words in PROFILE EXEC.  */
'pipe < profile exec | count words | console'
exit
```

The PIPE command reads the file named PROFILE EXEC, counts the number of words, and displays that number. PIPE is a CMS command, so it should be enclosed in single or double quotation marks as shown. To run the pipeline, run the exec as usual (by typing `WORDS`).

A more useful exec would let you specify a different file identifier:

```
/* WORDS EXEC counts the number of words.  */
parse arg fileid
'pipe <' fileid '| count words | console'
exit
```

The exec reads an argument and substitutes that argument for the variable `fileid` in the PIPE command, just as it would with any other CMS command in an exec.

Continuing Pipelines on Several Exec Lines

With experience, you will write longer pipelines. Rather than string out the PIPE command on a single line, use REXX continuation characters to split it onto several lines, as in the following example. The comma (,) is the REXX continuation character.

```
/* WORDS EXEC counts the number of words.  */
parse arg fileid
'pipe <' fileid,
  '| count words',
  '| console'
```

The lines in the example are indented to improve readability.

To continue a REXX string, enclose the string in single quotation marks and put the comma after the ending single quotation mark. In the example, `fileid` is not enclosed in quotation marks because it is a REXX variable (not part of a string).

REXX replaces the continuation character with a blank when it interprets the lines. If you do not want REXX to put a blank between the lines when it interprets them, use the REXX concatenation symbol (||):

```
/* Continuation without an intervening blank */
'pipe',
  'literal Hello'||,
  '| console'
```

When REXX interprets the lines, there is no blank between `Hello` and the following stage separator:

```
'pipe literal Hello| console'
```

Because trailing blanks are significant to some stages, it is important to remember how to use the concatenation symbol.

See *z/VM: REXX/VM Reference* for more about the REXX continuation character.

Using Pipelines As Part of an Exec

Pipelines can also be used to do tasks within a larger exec. Several device drivers are provided that read and write exec variables from a pipeline. After the data is in the pipeline, you can use filter stages to change that data.

Most experienced exec writers start using pipelines by substituting a PIPE command for the CMS EXECIO command. For example, the following exec fragment uses the EXECIO command to read the file TEST DATA into a stem variable (LINES.):

```
   :
'EXECIO * DISKR TEST DATA A (FINIS STEM LINES.'
   :
```

To do that with a pipeline, use:

```
   :
'pipe < test data a | stem lines.'
   :
```

STEM is a device driver that puts the contents of the pipeline into a stemmed array. (STEM is described in "STEM Stage" on page 65.)

Often the next step in using pipelines is moving other exec functions into the pipeline. Usually CMS Pipelines can do the work more efficiently. Continuing the above example, suppose that the stemmed variables are sorted after being set by EXECIO:

```
   :
'EXECIO * DISKR TEST DATA A (FINIS STEM LINES.'
call mysort                     /* Call a subroutine to sort LINES.n */
   :
```

Rather than use interpreted REXX instructions to sort the stemmed array, you can sort it in the PIPE command:

```
   :
'pipe < test data a | sort | stem lines.'
   :
```

The SORT stage sorts the records. (SORT is described in "Sorting Records (SORT)" on page 49.)

Another popular use of pipelines in execs is to simplify the code needed to get command results. For example, the following code fragment lists files on a given file mode and invokes the MYFILE EXEC for each file in the list:

```
/* Code fragment */
   :
address command
'MAKEBUF'                            /* Get a new stack            */
theirs=queued()                      /* Learn what is already there */
'LISTFILE * *' fm '(STACK FTYPE'     /* Issue a LISTFILE command    */
do queued()-theirs                   /* Loop through our entries    */
   pull afn aft                      /* Pull one off the stack      */
   'EXEC MYEXEC' afn aft             /* Build and execute a command */
end
'DROPBUF'                            /* Discard the stack           */
   :
```

This single PIPE command does the same thing:

```
/* Code fragment using pipelines */
  :
  :
'pipe',
   'cms LISTFILE * *' fm '(FTYPE',           /* Issue a LISTFILE command */
   '| specs /EXEC MYEXEC/ 1 1-* nextword',   /* Build a MYEXEC command   */
   '| cms',                                  /* Pass it to CMS           */
   '| console'                               /* Display any responses    */
  :
```

The first stage is a CMS stage. The CMS stage passes its operand to CMS for execution. In this example, the operand is a LISTFILE command. CMS writes the response lines from LISTFILE to its output stream. For each file listed, the SPECS stage builds a record containing the words EXEC MYEXEC followed by the file name and file type. SPECS writes these records to its output stream. Do not worry about understanding SPECS now. It is described in "SPECS Stage" on page 35. The second CMS stage reads the records written by SPECS and passes them to CMS. CONSOLE displays any messages or responses produced by MYEXEC.

When writing execs, treat PIPE as you would treat any other CMS command. There is nothing wrong with entering many PIPE commands in a single exec. It is also valid to call an exec from a pipeline even if that exec contains other PIPE commands. For example, suppose an exec named TEST contains a PIPE command. It is valid to call TEST from another PIPE command, as follows:

```
pipe cms exec test | console
```

Return Codes

Each stage of the pipeline gives a return code when it ends, but PIPE returns only one return code. PIPE returns the worst return code from all the stages in the pipeline. (Any negative return code is worse than any positive return code.)

You can put options following the PIPE command to display a message with each return code from each stage and to list stages that return with a nonzero return code; see the PIPE command description in *z/VM: CMS Pipelines Reference*.

When the PIPE command is entered at the terminal, the return code is displayed as part of the Ready; message after the command ends (just as it is for any other CMS command). In execs, the return code is stored in the variable RC. The following example shows a test for a nonzero return code after the PIPE command:

```
/* Code fragment using pipelines */
  :
  :
'pipe',
   'cms LISTFILE * *' fm '(FTYPE',           /* Issue a LISTFILE command */
   '| specs /EXEC MYEXEC/ 1 1-* nextword',   /* Build a MYEXEC command   */
   '| cms',                                  /* Pass it to CMS           */
   '| console'                               /* Display any responses    */
if rc¬=0 then                                /* Test for return code     */
   say 'Return code=' rc 'from PIPE.'
  :
```

User-Written Stages

There are two kinds of stages: built-in stages, and user-written stages. Built-in stages are supplied with CMS Pipelines. User-written stages are written by you (or some other user).

User-written stages are programs written in REXX or Assembler language. These REXX or Assembler programs have input and output streams. A typical user-written stage reads a record from its input stream, modifies that record, and then writes the record to its output stream.

CMS Pipelines provides *pipeline subcommands* and *assembler macros* that you can use in your stage to interact with the calling pipeline. For example, the READTO subcommand reads a record from the input stream, while the OUTPUT subcommand writes a record to the output stream. Pipeline subcommands are described in Chapter 5, "Writing Stages," on page 75 and in Chapter 6, "Multistream Pipelines," on page 107. Pipeline Assembler macros are described in Chapter 5, "Writing Stages," on page 75.

One pipeline subcommand, named CALLPIPE, lets you write *subroutine pipelines* within user-written stages. Subroutine pipelines consist of sections of pipelines. When a CALLPIPE subcommand is executed, the section of pipeline is, in effect, inserted into the pipeline that called the user-written stage. Subroutine pipelines let you preserve often-used sequences of stages so that they can be easily reused in other pipelines.

Because any stage is used by specifying its name, those you write can be easily shared with others. Users of your stages need to know only what the stages do to records flowing through them. They do not need to set up complicated parameter lists. Records go in and records come out.

Reading Interactive Dialogs

This book contains many examples that show commands entered at the terminal and the system's responses. Commands and data entered by a user are shown in **highlighted text** while system responses are not. For example:

```
tell yourid hi
 16:42:30  * MSG FROM YOURID  : hi
Ready;
```

How to Read Syntax Diagrams

This document uses diagrams (often called *railroad tracks*) to show the syntax of external interfaces.

To read a syntax diagram, follow the path of the line. Read from left to right and top to bottom.

- The ►►── symbol indicates the beginning of the syntax diagram.
- The ──► symbol, at the end of a line, indicates that the syntax diagram is continued on the next line.
- The ►── symbol, at the beginning of a line, indicates that the syntax diagram is continued from the previous line.
- The ──►◄ symbol indicates the end of the syntax diagram.

Within the syntax diagram, items on the line are required, items below the line are optional, and items above the line are defaults. See the following examples.

Syntax Diagram Convention	Example

Keywords and Constants:

A keyword or constant appears in uppercase letters. In this example, you must specify the item KEYWORD as shown.

In most cases, you can specify a keyword or constant in uppercase letters, lowercase letters, or any combination. However, some applications may have additional conventions for using all-uppercase or all-lowercase.

```
►►──KEYWORD──────────────────────►◄
```

Abbreviations:

Uppercase letters denote the shortest acceptable abbreviation of an item, and lowercase letters denote the part that can be omitted. If an item appears entirely in uppercase letters, it cannot be abbreviated.

In this example, you can specify KEYWO, KEYWOR, or KEYWORD.

```
►►──KEYWOrd──────────────────────►◄
```

Symbols:

You must specify these symbols exactly as they appear in the syntax diagram.

*	Asterisk
:	Colon
,	Comma
=	Equal Sign
-	Hyphen
()	Parentheses
.	Period

Variables:

A variable appears in highlighted lowercase, usually italics.

In this example, *var_name* represents a variable that you must specify following KEYWORD.

```
►►──KEYWOrd──var_name─────────────►◄
```

Repetitions:

An arrow returning to the left means that the item can be repeated.

A character within the arrow means that you must separate each repetition of the item with that character.

A number (1) by the arrow references a syntax note at the bottom of the diagram. The syntax note tells you how many times the item can be repeated.
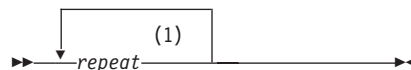
Syntax notes may also be used to explain other special aspects of the syntax.

```
      ┌──────────┐
►►────▼──repeat──┴────────────────►◄
```

```
      ┌────,─────┐
►►────▼──repeat──┴────────────────►◄
```

```
      ┌──────(1)─┐
►►────▼──repeat──┴────────────────►◄
```

Notes:

1   Specify *repeat* up to 5 times.

Required Item or Choice:

When an item is on the line, it is required. In this example, you must specify A.

When two or more items are in a stack and one of them is on the line, you must specify one item. In this example, you must choose A, B, or C.

```
►►──A─────────────────────────────►◄
```

```
►►─┬──A──┬────────────────────────►◄
   ├──B──┤
   └──C──┘
```
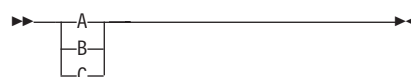
Syntax Diagram Convention	Example

Optional Item or Choice:

When an item is below the line, it is optional. In this example, you can choose A or nothing at all.
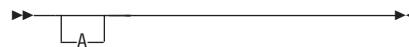
When two or more items are in a stack below the line, all of them are optional. In this example, you can choose A, B, C, or nothing at all.
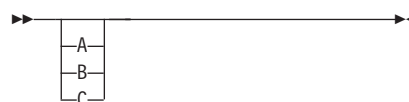
Defaults:

When an item is above the line, it is the default. The system will use the default unless you override it. You can override the default by specifying an option from the stack below the line.
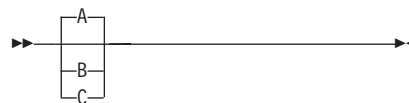
In this example, A is the default. You can override A by choosing B or C.

Repeatable Choice:

A stack of items followed by an arrow returning to the left means that you can select more than one item or, in some cases, repeat a single item.
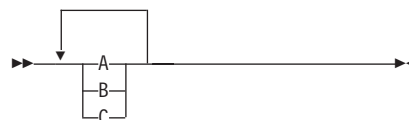
In this example, you can choose any combination of A, B, or C.

Syntax Fragment:

Some diagrams, because of their length, must fragment the syntax. The fragment name appears between vertical bars in the diagram. The expanded fragment appears in the diagram after a heading with the same fragment name.
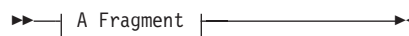
In this example, the fragment is named "A Fragment."

A Fragment:

Message and Response Notation

This document might include examples of messages or responses. Although most examples are shown exactly as they would appear, some content may depend on the specific situation. The following notation is used to show variable, optional, or alternative content:

xxx   Highlighted text (usually italics) indicates a variable that represents the data that will be displayed.

[ ]   Brackets enclose an optional item that might be displayed.

{ }   Braces enclose alternative items, one of which will be displayed.

I   The vertical bar separates items within brackets or braces.

...   The ellipsis indicates that the preceding item might be repeated. A vertical ellipsis indicates that the preceding line, or a variation of that line, might be repeated.

Pipeline Help

There are three ways to get help information about CMS Pipelines. You can use the z/VM HELP Facility, or you can use the HELP or AHELP stages of CMS Pipelines. AHELP is provided by the author and may offer advanced assistance. All three ways are described in the following sections.

Using the Online HELP Facility

You can receive online information about the commands described in this book using the z/VM HELP Facility. For example, to display a menu of the CMS Pipelines stages, pipeline subcommands, and assembler macros, enter:

```
help pipe menu
```

To display information about a specific stage or pipeline subcommand (the BETWEEN stage in this example), enter:

```
help pipe between
```

To display information about the PIPE command, enter:

```
help pipe
```

To display a list of CMS Pipelines tasks, enter:

```
help pipe task
```

You can also display information about a message by entering one of the following commands:

```
help msgid  or  help msg msgid
```

For example, to display information about message FPL001I, you can enter one of the following commands:

```
help fpl001i  or  help msg fpl001i
```

For more information about using the HELP Facility, see the *z/VM: CMS User's Guide*. To display the main HELP Task Menu, enter:

```
help
```

For more information about the HELP command, see the *z/VM: CMS Commands and Utilities Reference* or enter:

```
help cms help
```

Using the HELP Stage

CMS Pipelines includes a HELP stage that displays information about CMS Pipelines messages, stages, and pipeline subcommands. To use it, enter a one-stage PIPE command:

```
pipe help literal
```

The above example gets help information for the LITERAL stage. The word `literal` is an operand of the HELP stage. To get help on any pipeline message, stage, or pipeline subcommand, type its name as an operand of HELP.

Note: Using PIPE HELP provides the same online information provided by the z/VM HELP Facility.

CMS Pipelines remembers messages it has issued. To get help on the message issued most recently, specify `0` as the operand on the HELP stage:

```
pipe help 0
```

If you omit the operand, it defaults to 0. The following PIPE command yields the same result:

```
pipe help
```

CMS Pipelines remembers the ten messages issued before the last one. You can get help for any of these messages without knowing its message number. For instance, to get help for the next to last message, enter:

```
pipe help 1
```

Refer to the *z/VM: CMS Pipelines Reference* or the *CMS/TSO Pipelines: Author's Edition* for more information about the HELP stage.

Using the AHELP Stage

CMS Pipelines includes an AHELP stage that displays author information about CMS Pipelines messages, stages, and pipeline subcommands in the same manner as the HELP stage. AHELP often provides more advanced assistance than the HELP stage. The function and syntax of the AHELP stage is identical to that of the HELP stage.

Refer to the *z/VM: CMS Pipelines Reference* for more information about the AHELP stage.

Migration Information

If you have been using the CMS Pipelines Programming RPQ, refer to Appendix C, "Migrating to CMS Pipelines," on page 267. That appendix summarizes the differences between the CMS Pipelines and the Programming RPQ.

Reference Book

The *z/VM: CMS Pipelines Reference* and the *CMS/TSO Pipelines: Author's Edition* describe the PIPE command, CMS Pipelines stages and subcommands. All of the descriptions contain examples. These books briefly describe some commonly used stages and subcommands. If you are designing an application that will use CMS Pipelines, browse through these books to find other topics that may do exactly what you need.

Chapter 2. Filters

CMS Pipelines has many *filter* stages. A filter reads data from its input stream, does some work using that data, and writes the results to its output stream. The difference between a filter and a device driver is that a filter does not interact with devices or other system resources, as device drivers do.

The results that filters write can be far different from the data read. The CHANGE stage, for example, writes a record to its output stream for every record it reads. LOCATE, on the other hand, reads all the records, but writes only those that match a search string. COUNT reads every record in its input stream, but writes only a single record to its output stream. By stringing filters together, you can transform raw data into useful results.

This chapter describes some commonly used filters:
- Filters that select records by content
- Filters that select records by position
- Filters that change records
- Miscellaneous filters:
  - A filter for duplicating records
  - A filter for counting records, words, and characters
  - A filter for sorting records
  - A filter for buffering records.

Some of the filters are similar in function and format to XEDIT subcommands. These similarities are intentional and are intended to help you learn CMS Pipelines quickly.

Selecting Records by Content

Several stages select pipeline records. That is, they read all records in the pipeline, but write only those that meet some selection criteria. This section describes filters that select records based on the content of the record itself. Those filters include:
- LOCATE and NLOCATE
- FIND and NFIND
- TOLABEL and FRLABEL
- UNIQUE.

All of these filters are *case sensitive*. In string comparisons, the words "Apple" and "apple" are not considered equal. The CASEI stage can be used in combination with these filters to do *case insensitive* comparisons.

Looking Everywhere in the Record (LOCATE, NLOCATE)

The LOCATE stage selects records having a particular string; NLOCATE selects records not containing a particular string. By default, both filters look everywhere in the record for the string you supply. You can specify input ranges to limit the scope of the search.

17

LOCATE Stage

The LOCATE stage writes only the records containing a specific string. Suppose you have a file named WINTER THOUGHTS that contains the following records:

```
I like winter.
Winter is cold.
WINTER FUN
winter sun
```

Figure 12 shows an example of LOCATE. The < stage reads the WINTER THOUGHTS file and writes the records to its output stream. The next stage, LOCATE, examines the records, looking for the string winter. If the record contains the string, LOCATE writes it to its output stream. If it does not, LOCATE discards it. The next stage, CONSOLE, displays the records that LOCATE writes.

```
pipe < winter thoughts | locate /winter/ | console
I like winter.
winter sun
Ready;
```

Figure 12. LOCATE Stage Example: Locating Records Containing the String

Because LOCATE is case sensitive, it selects only the first and the last lines of this file. LOCATE is looking for lowercase winter, but the second and third lines contain Winter and WINTER.

What would happen if you looked for summer? Figure 13 shows the result.

```
pipe < winter thoughts | locate /summer/ | console
Ready;
```

Figure 13. LOCATE Stage Example: Not Locating Records Containing the String

Nothing in the file WINTER THOUGHTS matches summer so LOCATE does not write anything to its output stream. Consequently, CONSOLE has nothing to display. It is perfectly acceptable for LOCATE (or any other filter that selects records) not to find anything. CONSOLE does not give an error when there aren't any records in its input stream. The pipeline remains intact—no error has occurred.

Before continuing with the next example of LOCATE, it is necessary to understand a bit more about the LITERAL stage. The next few examples and, in fact, many of the examples in the rest of the chapter use the LITERAL stage. LITERAL provides an easy way to put data in a pipeline so that you can see what a filter does.

When LITERAL is first in a pipeline, it simply writes a record to its output stream. That record contains whatever you type as the LITERAL operand. When LITERAL is not first, it writes a record containing the operand, and then copies any records in its input stream to its output stream. This is a very important characteristic of LITERAL, and one that is easily forgotten. Remember: LITERAL writes before copying. There is more about LITERAL in "Writing Literal Strings to a Pipeline (LITERAL)" on page 60.

Look again at the example of LOCATE in Figure 12. Notice that slashes (/) are used in LOCATE to delimit the string you are searching for. You can use any character to delimit the string that is not itself in the string. To select records containing x/y for

example, you might use commas as delimiters, as shown in Figure 14.

```
pipe literal z=x/y | locate ,x/y, | console
z=x/y
Ready;
```

Figure 14. LOCATE Stage Example: Using Delimiters

Note: When entering a PIPE command next to a file in the list displayed by commands like FILELIST and RDRLIST, do not use the slash (/) as the delimiter. The slash has a special meaning to these commands.

Using Input Ranges with LOCATE: It is possible to limit the search area inspected by LOCATE. Simply enter an input range before the string operand. An input range defines a particular location of the record on which LOCATE operates. The first example in Figure 15 uses words as its input range, displaying records with the letter o in the second word. A word, by default, is defined as characters delimited by a blank. The second example defines the input range as columns, looking for o anywhere in columns 1 through 7.

```
pipe literal red shoe|literal orange hat|locate w2 /o/| console
red shoe
Ready;
pipe literal red shoe|literal orange hat|locate 1-7 /o/|console
orange hat
red shoe
Ready;
```

Figure 15. LOCATE Stage Example: Using an Input Range

Notice in the second PIPE command example above that orange hat is displayed before red shoe. Given the order of the LITERAL stages in the pipeline, this is not what you might expect. To understand what happened, remember the rule that LITERAL writes before copying. In the example, the first LITERAL stage writes a record containing red shoe to its output stream. The second LITERAL stage writes a record containing its operand (orange hat) to its output stream, and *then* copies any records in its input stream to its output stream. So the orange hat record travels through the pipeline *before* the red shoe record.

See the *z/VM: CMS Pipelines Reference* for a complete description of input ranges under the LOCATE stage.

The LOCATE stage does not let you specify multiple strings. Instead, use multiple LOCATE stages. For example, to locate records in NOVEL SCRIPT that have both the strings mercurial and saturnine, you would enter:

```
pipe < novel script | locate /mercurial/ | locate /saturnine/ | console
Mary was mercurial.  Sam, saturnine.  Yet on this day ten years ago they wed.
Ready;
```

Figure 16. LOCATE Stage Example: Locating Multiple Strings

If you want to locate records that have *either* `saturnine` or `mercurial`, you will need
to use a multistream pipeline. (See Figure 172 on page 118.)

Selecting Records By Length: You can select records that have some minimum
length by using LOCATE without a string. Instead, just specify a column. Figure 17
selects records having a length of 2 or more bytes.

```
pipe literal a| locate 2 | console
Ready;
pipe literal ab| locate 2 | console
ab
Ready;
```

Figure 17. LOCATE Stage Example: Specifying Data Length

Be careful about trailing blanks when you use LITERAL. Notice that a stage
separator immediately follows the strings in Figure 17—there are no trailing blanks.
Try putting a blank between `a` and the stage separator. LITERAL puts any trailing
blanks on records it writes. The blank is counted as a character, which means the
entire string is 2 characters long. LOCATE will then select the string.

Selecting Entire Records: There are instances where you will want to select
entire records, for instance when you want to remove all blank lines from a file. The
example in this section uses a file named BLNKNSTF SCRIPT that contains a
blank line:

```
now is the time
for your heart to skip a beat

and robins to herald the coming of spring
```

Figure 18 reads in your file, uses the STRIP stage to change all blank lines to null
(zero length) records, then uses LOCATE with no operands to remove all those
records in the file with a length of 0, and displays the remaining records as output
to the terminal.

```
pipe < blknstf script a | strip | locate | console
now is the time
for your heart to skip a beat
and robins to herald the coming of spring
Ready;
```

Figure 18. LOCATE Stage Example: Removing Blank Lines from File

NLOCATE Stage

NLOCATE does the opposite of the LOCATE stage. It writes all records that do not
contain the string specified as the operand. Figure 19 on page 21 shows an
example of NLOCATE that reads the file WINTER THOUGHTS. Suppose WINTER
THOUGHTS contains the following records:

```
I like winter.
Winter is cold.
WINTER FUN
winter sun
```

```
pipe < winter thoughts | nlocate /winter/ | console
Winter is cold.
WINTER FUN
Ready;
```

Figure 19. NLOCATE Stage Example: Locating Records

Except for the use of NLOCATE instead of LOCATE, the pipeline is identical with Figure 12 on page 18. As you would expect, the results are exactly opposite. The first and last lines are not echoed because the string `winter` precisely matches the NLOCATE argument. The second and third lines are echoed because `Winter` and `WINTER` do not match the NLOCATE argument.

You can limit the range of columns inspected by NLOCATE. Specify a column range before the string operand (just like LOCATE).

You can select short records with NLOCATE. Figure 20 shows how to select all records having a length less than 5.

```
pipe literal too long| literal okay| nlocate 5 | console
okay
Ready;
```

Figure 20. NLOCATE Stage Example: Specifying Data Length

The record `okay` is selected because it has fewer than 5 characters.

Looking at the Beginning of a Record (FIND, NFIND, TOLABEL, FRLABEL)

CMS Pipelines includes stages that look only at the beginning of a record. This section describes four of them: FIND, NFIND, TOLABEL, and FRLABEL. FIND and NFIND select records that begin with (or do not begin with, for NFIND) a particular string. TOLABEL selects records before a record beginning with a particular string. FRLABEL selects records after a record beginning with a particular string.

Note: The STRFIND, STRNFIND, STRTOLABEL and STRFRLABEL stages can also be used. These stages affect records that begin with a specified string of characters.

FIND and NFIND Stages

These stages select records according to whether the leading characters match the argument string.

```
pipe literal abc | literal def | find ab| console
abc
Ready;
```

Figure 21. FIND Stage Example: Finding Records Containing the String

If you do not care what characters certain columns hold, use a blank. Blanks in the argument string indicate columns whose contents are arbitrary, *not* columns containing only a blank character. (This differs from the way LOCATE works—to

LOCATE, a blank is a blank.) Figure 22 shows an example that selects records beginning with `the most`, where any character can be between `the` and `most`.

```
pipe literal thermostat | literal the most interesting| find the most| console
the most interesting
thermostat
Ready;
```

Figure 22. FIND Stage Example: Using Arbitrary Characters in the String

How do you find a blank? Use an underscore (_) in the argument string to indicate that a column must have a blank. See how the results change in Figure 23.

```
pipe literal thermostat | literal the most interesting| find the_most| console
the most interesting
Ready;
```

Figure 23. FIND Stage Example: Specifying a Blank in the String

Because an underscore indicates a required blank, you cannot search for underscores directly. Instead, use XLATE to transpose underscore and some other character before the FIND or NFIND stage and restore them afterward. See "Translating Characters (XLATE)" on page 27 for a description of the XLATE stage.

Be careful with blanks before the stage separator. If there are one or more blanks between the string you want to find and the stage separator, these positions must be present in the input record even if their contents are ignored. This is why Figure 24 shows responses you might not expect. The input record is only one character long, but the argument to FIND (and NFIND) is two bytes.

```
pipe literal a| find a | console
Ready;
pipe literal a| nfind a | console
a
Ready;
```

Figure 24. FIND and NFIND Stages Examples

NFIND is the converse of FIND; it selects records that do not match the argument.

You can use FIND with a suitable number of blanks to select records of a certain size (or longer), but it is simpler to use LOCATE with a column number to do this. In a similar way, select short records with NLOCATE.

TOLABEL and FRLABEL Stages

TOLABEL copies records to its output stream until it meets a record that begins with the argument string. TOLABEL does not write the matching record to its output stream.

FRLABEL skips records until one is met with the required beginning; the record with the matching string and the remaining records are copied to the output. It doesn't matter what is in the rest of the file, even if there is another matching record.

Figure 25 shows the examples of FRLABEL and TOLABEL selecting records from an F-format file called TEST FILE.

```
pipe < test file | console
Text before START
START
Text in the middle
END
Text after END.
Ready;
pipe < test file | frlabel START | console
START
Text in the middle
END
Text after END.
Ready;
pipe < test file | tolabel END | console
Text before START
START
Text in the middle
Ready;
```

Figure 25. FRLABEL and TOLABEL Stages Examples

FRLABEL and TOLABEL are often used in combination. For instance, Figure 26 shows how to select records from the label START and to (but not including) the label END.

```
pipe < test file | frlabel START | tolabel END | console
START
Text in the middle
Ready;
```

Figure 26. FRLABEL and TOLABEL Stages Example

Note the blank after START and END. In this example, the blank after each argument string will select the same records as it would if each argument string did not have a trailing blank. Your results could vary from what is shown in Figure 26 if the pipeline reads in a V-format file rather than an F-format file and the lines in the file did not have trailing blanks. With them the pipeline would stop, for instance, without any records being matched.

Looking at the End of a Record

There is no stage to select records that end with some particular string, but several filters can be combined to do it. The trick is to copy the end of each record to the beginning of each record. For that, you need to use the SPECS stage, which has not yet been discussed. Refer to Figure 88 on page 47 to see an example of looking at the end of a record.

Discarding Duplicate Records (UNIQUE)

There are two stages that get rid of duplicate records: SORT (with the UNIQUE operand specified) and UNIQUE. They produce different results. Use UNIQUE when the records are:

• Already sorted and you want to remove duplicates

- Not sorted and you want to remove duplicates that happen to be adjacent to each other.

Use SORT UNIQUE when you have unsorted records and you want to discard duplicates while sorting those records. SORT UNIQUE is described in "Discarding Duplicates When Sorting" on page 51.

The UNIQUE stage gets rid of duplicate records that happen to be next to each other. It does not compare each record with all other records. If the records are already sorted, all duplicates are next to each other. Using UNIQUE in this case causes all duplicates to be discarded.

If the records are not sorted, duplicates may not be next to each other, so UNIQUE may not remove them all.

UNIQUE compares each input record with the next one. By default it discards a record that has the same contents as the following one. That is, series of identical records are replaced with the last occurrence of the record.

Suppose you have a file named WEATHER INFO that contains the following records:

```
It is raining today.
It is raining today.
It is raining today.
Next week may be better.
Always be an optimist.
Always be an optimist.
Always be an optimist.
It is raining today.
```

The records are not sorted, so not all duplicates are adjacent. UNIQUE will not remove all duplicates (Figure 27).

```
pipe < weather info | unique | console
It is raining today.
Next week may be better.
Always be an optimist.
It is raining today.
Ready;
```

Figure 27. UNIQUE Stage Example

Discarding Unique Records (UNIQUE MULTIPLE)

UNIQUE MULTIPLE discards records that do not have adjacent duplicates. It keeps records that occur at least twice together. DUPLF EXEC, shown in Figure 28 on page 25, uses UNIQUE MULTIPLE. It creates a list of files that exist on two minidisks.

```
/* DUPLF EXEC -- Find files that are on two minidisks and report */
/*               the first of each                               */

arg fm1 fm2 .
if fm2='' then exit 24                 /* Incomplete parameters?  */

address command
'PIPE',
   'literal LISTFILE * *' fm2,         /* Second listfile command */
   '| cms LISTFILE * *' fm1,           /* Execute both LISTFILEs   */
   '| sort',                           /* Put duplicates together */
   '| unique 1.17 multiple',           /* Select only duplicates  */
   '| unique 1.17 first',              /* Take only the first     */
   '| console'                         /* Display them            */
exit rc
```

Figure 28. UNIQUE MULTIPLE Stage Example (DUPLF EXEC Contents)

DUPLF shows a useful characteristic of the CMS stage: it passes records in its input stream to CMS. The CMS stage, you will recall, passes its operand to CMS for execution. In DUPLF, the operand is a LISTFILE command. CMS writes any responses from the command to its output stream. After processing its operand, CMS reads its input stream. It passes each record in its input stream to CMS for execution. CMS writes the responses from the commands to its output stream.

Assuming there are records in the pipeline for the CMS stage to read, you can omit the operand. In this case, the input records are passed to CMS for execution and CMS traps any responses.

In DUPLF, two LISTFILE commands are executed. The first one executed is the operand of the CMS stage. The second one is the LISTFILE that LITERAL writes to its output stream. The results are sorted to put duplicates together. Then UNIQUE MULTIPLE is executed. A column range of 1.17 is specified on UNIQUE MULTIPLE to restrict the test to the file name and file type. (The column range 1.17 specifies a field that begins in column 1 and is 17 characters long.) Now the pipeline contains records having duplicate file names and file types.

A second UNIQUE stage eliminates duplicate lines. The result, displayed by CONSOLE, is one line for each file on both minidisks. Figure 29 shows a sample run.

```
duplf a b
DSMUTTOC SCRIPT   A1
TMF      SAVED    A1
Ready;
```

Figure 29. UNIQUE MULTIPLE Stage Example (DUPLF EXEC Results)

Selecting Records by Position (TAKE, DROP)

TAKE and DROP make it easy to select records based on their positions in the input stream. The examples in this section use a file named STRING LIST that contains a list of stringed instruments:

```
harp
lyre
lute
```

```
viola
violin
bouzouki
oud
guitar
mandolin
harpsichord
```

The TAKE filter picks the first or last *n* records (where *n* is zero, or a positive number, or * for all) for output to the pipeline. By default, TAKE selects the first records. Figure 30, shown next, demonstrates taking the first 3 records.

```
pipe < string list | take 3 | console
harp
lyre
lute
Ready;
```

Figure 30. TAKE Stage Example

To take the last records, specify the LAST operand on TAKE. Figure 31 shows how to take the last 4 records.

```
pipe < string list | take last 4 | console
oud
guitar
mandolin
harpsichord
Ready;
```

Figure 31. TAKE LAST Stage Example

The DROP stage is the converse of TAKE. It lets you discard the first or last *n* records (where *n* is zero, or a positive number, or * for all). Figure 32, shown next, drops the first two records from the file STRING LIST.

```
pipe < string list | drop 2 | console
lute
viola
violin
bouzouki
oud
guitar
mandolin
harpsichord
Ready;
```

Figure 32. DROP Stage Example

To drop the last records in the pipeline, use the LAST operand. Figure 33, shown next, shows how to drop the last three records of the STRING LIST file.

```
pipe < string list | drop last 3 | console
harp
lyre
lute
viola
violin
bouzouki
oud
Ready;
```

Figure 33. DROP LAST Stage Example

By combining TAKE and DROP, you can get records in the middle. One use of this would be removing top and bottom margins from formatted text files. Figure 34 shows how to get the fifth and sixth lines of the STRING LIST file.

```
pipe < string list | drop 4 | take 2 | console
violin
bouzouki
Ready;
```

Figure 34. TAKE and DROP Stage Example

Changing Records

CMS Pipelines includes stages that change the records passing through them. You can:
- Translate characters to uppercase or to lowercase
- Substitute one character for another
- Split and join records
- Expand and truncate records
- Remove leading or trailing characters from records
- Replace a string with another string
- Rearrange the contents of a record.

REXX programmers will notice that many of these stages have counterparts in the REXX language. The difference is that these stages work on all data flowing through the stage, while in REXX they operate on a single expression.

Translating Characters (XLATE)

XLATE translates data passing through the pipeline on a character-by-character basis. You can translate characters to lowercase or to uppercase. You can also replace all occurrences of one character with another. Figure 35 shows how to translate characters to uppercase:

```
pipe literal I'm NOT mad about it. | xlate upper | console
I'M NOT MAD ABOUT IT.
Ready;
```

Figure 35. XLATE Stage Example: Translating a String to Uppercase

Translating a string to lowercase is just as easy, as Figure 36 on page 28 shows.

```
pipe literal I'm NOT mad about it. | xlate lower | console
i'm not mad about it.
Ready;
```

Figure 36. XLATE Stage Example: Translating a String to Lowercase

To limit the translation to certain columns, specify a column range after the XLATE. (See Figure 37.)

```
pipe literal Play piano, not forte. | xlate 17.5 upper | console
Play piano, not FORTE.
Ready;
```

Figure 37. XLATE Stage Example: Using a Column Range

The 17.5 defines a range of 5 columns starting on column 17. (Specifying 17-21 would yield the same result.)

With the UPPER and LOWER operands, many characters have been translated with a single XLATE stage. It is also possible to translate individual characters. Figure 38 shows how. Instead of specifying UPPER and LOWER operands, specify pairs of characters after XLATE. The example in Figure 38 changes all e's to X's.

```
pipe literal Extra eggplants free. | xlate e X | console
Extra Xggplants frXX.
Ready;
```

Figure 38. XLATE Stage Example: Translating Individual Characters

Notice that the capital E in `Extra` was not changed. XLATE is case sensitive. To change both small and capital E's to X's, specify two pairs of characters after XLATE (Figure 39).

```
pipe literal Extra eggplants free. | xlate e X E X | console
Xxtra Xggplants frXX.
Ready;
```

Figure 39. XLATE Stage Example: Translating Multiple Characters

You can specify many pairs of characters after XLATE. The characters are not limited to those in the alphabet. You can also specify numbers, punctuation, and hexadecimal values.

When you are specifying numbers, you may also need to specify a column range. Otherwise, XLATE might mistake your first pair of numbers for a range of characters. Figure 40 on page 29 shows a column range (1-*) on the XLATE stage, followed by a number. The string `1-*` indicates a range from column 1 through the last column. This means that the complete record should be translated. The string `1-*` is specified to ensure that CMS Pipelines parses the stage properly.

```
pipe literal 370 | xlate 1-* 3 E 7 S 0 A | console
ESA
Ready;
```

Figure 40. XLATE Stage Example: Specifying a Range

Instead of specifying characters directly, you can use hexadecimal values. This is useful when you want to specify characters that you cannot type on your keyboard. It is also useful when you want to specify a blank, which is the hexadecimal value X'40'. Specify hexadecimal values by typing the two characters that make up the byte. Figure 41 changes all parentheses to blanks.

```
pipe literal x(x)x | xlate 1-* ( 40 ) 40 | console
x x x
Ready;
```

Figure 41. XLATE Stage Example: Specifying Hexadecimal Values

You can also specify a range of characters (as opposed to a column range) to be translated instead of individual characters. For example, suppose you want to translate the numbers 3 through 5 to equal signs. One way to do it is by typing each character:

```
...| xlate 1-* 3 = 4 = 5 = | ...
```

Figure 42 shows another way. It uses a character range to do the same translation. The second PIPE command in Figure 42 also shows a character range. It removes punctuation from a record by translating the punctuation to blanks. Hexadecimal values are used in the second example.

```
pipe literal 123456789 | xlate 1-* 3-5 = | console
12===6789
Ready;
pipe literal (In parentheses.) | xlate 1-* 41-7f 40 | console
 In parentheses
Ready;
```

Figure 42. XLATE Stage Example: Using Ranges of Characters

You can specify ranges for both the input and the output. Figure 43 translates characters 1 through 9 into the corresponding letters A through I.

```
pipe literal 123 12389 | xlate 1-* 1-9 A-I | console
ABC ABCHI
Ready;
```

Figure 43. XLATE Stage Example: Using Ranges for Input and Output

To translate all but a few of the characters in a range, specify the range of characters and also the characters you wish to omit. Before XLATE does any translation, it determines whether any character is specified more than once in the operands. XLATE uses only the last occurrence when translating the data. Figure 44 on page 30 shows an example. The characters from c to g are translated

to equal signs, except for the character e. The character e is translated to itself.

```
pipe literal abcdefghi | xlate c-g = e e  | console
ab==e==hi
Ready;
```

Figure 44. XLATE Stage Example: Overriding a Character Range

You can use the same technique to augment the built-in translations such as uppercase or lowercase.

Splitting and Joining (SPLIT, JOIN)

The SPLIT and JOIN stages split and join records. Other filters are available that block and deblock data in other formats, including some formats supported by MVS access methods. For more information see, Chapter 9, "Blocking and Deblocking," on page 181.

SPLIT Stage

By default, SPLIT creates an output record for each blank-delimited word in its input records. (See Figure 45.)

```
pipe literal A phrase with five words | split | console
A
phrase
with
five
words
Ready;
```

Figure 45. SPLIT Stage Example

SPLIT has operands to define other kinds of splitting. See the *z/VM: CMS Pipelines Reference* for a complete description of SPLIT.

JOIN Stage

JOIN creates a single record from one or more input records. By default, JOIN puts together pairs of input records. (See Figure 46.)

```
pipe literal A phrase with five words | split | join | console
Aphrase
withfive
words
Ready;
```

Figure 46. JOIN Stage Example: Joining Pairs of Records

LITERAL puts a record into the pipeline. SPLIT puts each of the five words on a separate record. Then JOIN combines pairs of records. Because there weren't an even number of input records, JOIN puts words by itself in an output record. Notice that JOIN puts records together without intervening blanks. To put a blank between the joined records, specify / / as shown in Figure 47 on page 31.

```
pipe literal A phrase with five words | split | join / / | console
A phrase
with five
words
Ready;
```

Figure 47. JOIN Stage Example: Putting a Space between Joined Records

Between the delimiters (/) you can put any string you want to insert between the joined records. In Figure 48, dashes (--) are inserted.

```
pipe literal A phrase with five words | split | join /--/ | console
A--phase
with--five
words
Ready;
```

Figure 48. JOIN Stage Example: Putting Strings between Joined Records

As usual, you can use any character not in the string itself as the delimiter. Figure 49 shows question marks (?) used as delimiters.

```
pipe literal A phrase with five words | split | join ?--? | console
A--phase
with--five
words
Ready;
```

Figure 49. JOIN Stage Example: Using String Delimiters

You can also join more than two records with JOIN by telling it the number of records to put together. Put the number after the JOIN keyword before any delimited string. The number itself is the number of records to be joined to the first one. If you want to join every three records, specify 2 *not* 3. Figure 50 joins every three input records into one output record.

```
pipe literal A phrase with five words | split | join 2 / / | console
A phrase with
five words
Ready;
```

Figure 50. JOIN Stage Example: Joining More than Two Input Records

Because the number of input records is not evenly divisible by three, JOIN puts the last two input records on a single output record.

To put all input records in a single output record, specify an asterisk (*) instead of a number. Figure 51 on page 32 splits apart the record written by LITERAL and puts it back together again.

```
pipe literal Break up and make up | split | join * / / | console
Break up and make up
Ready;
```

Figure 51. JOIN Stage Example: Joining All Pipeline Records

Padding and Chopping (PAD, CHOP)

You can pad (expand) or chop (truncate) records so they have a desired length. Often PAD and CHOP are combined to create a particular output format.

CHOP Stage

CHOP truncates each record after a column. Specify the column number after CHOP. (See Figure 52.)

```
pipe literal She loves me; she loves me not. | chop 12 | console
She loves me
Ready;
```

Figure 52. CHOP Stage Example

PAD Stage

PAD fills each record to the specified length with a pad character (the default is a blank). You can request the pad character to be filled on the right or on the left.

Figure 53 chops the record at column 12 and then pads the record to column 20 with question marks (?). The pad character must follow the column number.

```
pipe literal She loves me; she loves me not. | chop 12 | pad 20 ? | console
She loves me????????
Ready;
```

Figure 53. PAD Stage Example

By default, PAD adds pad characters to the right side of the string. To add them to the left, type `left` after `pad` as shown in Figure 54.

```
pipe literal She loves me; she loves me not.| chop 12 | pad left 20 . |console
........She loves me
Ready;
```

Figure 54. PAD Stage Example: Padding on the Left

Figure 55 on page 33 combines CHOP and PAD to create records with fixed lengths (here with a length of 10). The TEST DATA file being read into the pipeline is a V-format file.

```
pipe < test data | console
Short
This record will be truncated
Ready;
pipe < test data | pad 10 ? | chop 10 | console
Short?????
This recor
Ready;
```

Figure 55. PAD and CHOP Stages Example

In this example, PAD extends short records to 10 characters with question marks
(?) on the right. CHOP truncates records that are longer than 10 characters.

Combining PAD and CHOP to create fixed records is useful when you want to
create an F-format file. See Chapter 4, "Device Drivers," on page 59 for more about
creating F-format files.

Removing Leading or Trailing Characters (STRIP)

Use the STRIP stage to remove blanks from the beginning and the end of records.
Figure 56 shows a simple STRIP example.

```
pipe literal     Hello?      | strip | console
Hello?
Ready;
```

Figure 56. STRIP Stage Example

STRIP can also remove only leading or trailing blanks, as shown in Figure 57.

```
pipe literal c| literal   b  | literal a| join *| console
a  b  c
Ready;
pipe literal c| literal   b  | literal a| strip leading| join *| console
ab  c
Ready;
pipe literal c| literal   b  | literal a| strip trailing| join *| console
a  bc
Ready;
pipe literal c| literal   b  | literal a| strip| join *| console
abc
Ready;
```

Figure 57. STRIP Stage Example: Stripping Leading or Trailing Characters

You can also use STRIP to remove characters other than blank. Use the STRING
operand to identify the string to be stripped.

```
pipe literal 0000120| strip leading string /0/| console
120
Ready;
pipe literal bread meat bread| strip string /bread/| console
 meat
Ready;
pipe literal bread meat bread lettuce bread| strip string /bread/| console
 meat bread lettuce
Ready;
```

Figure 58. STRIP Stage Example: Stripping Nonblank Characters

Changing and Rearranging Contents (CHANGE, SPECS)

The CHANGE and SPECS stages edit the contents of records passing through the pipeline. CHANGE replaces one group of characters with another. SPECS can rearrange record contents, add literal strings and record numbers, and convert fields from one format to another (for example, from character to unpacked hexadecimal).

CHANGE Stage

The CHANGE stage replaces one character or group of characters with another. It is similar to the XEDIT CHANGE subcommand. For example, suppose you want to change the name John to Martin in a file named STORY SCRIPT. You might enter:

```
pipe < story script | change /John/Martin/ | console
```

The CHANGE stage replaces the string `John` with the string `Martin` wherever it occurs in every record in the pipeline. Note that `Martin` is 6 characters while `John` is 4. The strings do not have to be the same length.

Be careful when using CHANGE. The above CHANGE stage would change:

```
John Smith and Bob Johnson ate johnnycake.
```

to:

```
Martin Smith and Bob Martinson ate johnnycake.
```

Both `John` and `Johnson` were changed because there isn't a blank after `John` in the CHANGE stage. This may or may not be what you intended. The word `johnnycake` was not altered because CHANGE is case sensitive.

The default is to change all occurrences in every record. You can limit the number of substitutions per record by writing a number after the delimited strings. For example, write `1` after the delimited strings to change only the first occurrence in every record.

You can also limit CHANGE by specifying a range of columns. In Figure 59 on page 35, CHANGE searches for `John` only in the first four columns of each record. Consequently, `Johnson` is not changed.

```
pipe < story script | change 1-4 /John/Martin/ | console
Martin Smith and Bob Johnson ate johnnycake.
Ready;
```

Figure 59. CHANGE Stage Example: Using Column Ranges

The column range tells CHANGE where to look. It does not cause CHANGE to limit replacements to the first four columns. `Martin` still replaced `John` even though `Martin` goes beyond the fourth column.

Figure 60 shows two column ranges. CHANGE looks for `John` in columns 1 through 4 and in columns 20 through 23. Use parentheses as shown when entering more than one column range. This time `Johnson` is changed because it begins in column 20:

```
pipe < story script | change (1-4 20-23) /John/Martin/ | console
Martin Smith and Bob Martinson ate johnnycake.
Ready;
```

Figure 60. CHANGE Stage Example: Using Several Column Ranges

CHANGE looks in all column ranges before changing the record. So, the fact that `Martin` is longer than `John` does not prevent CHANGE from finding the second `John` starting in column 20. In the output, `Martinson` begins in column 22.

If you are working with structured data and want to avoid changing the length of the record, simply make the search or replacement strings the same length:

```
pipe < yourid netlog | change /MIKE    /BARBARA / | console
```

You can specify up to 10 column ranges. If you specify more than one, delimit each column range with at least one blank and enclose the set in parentheses. Make sure the ranges are in ascending order and do not overlap.

SPECS Stage

The SPECS stage rearranges contents of records. It is one of the most versatile stages. This section covers many SPECS operands, but it does not cover all of them. After you have gained some experience with SPECS, refer to the *z/VM: CMS Pipelines Reference* to see what other functions it offers.

SPECS creates output records by piecing together data from various sources. An output record might contain snippets from an input record, a literal string that you supply, and a record number that SPECS itself generates. For each piece of data you want in the output records, you must tell SPECS where it is to get the data and where it should be placed.

For example, suppose your input records contain names and addresses. Each record contains a surname in columns 20 through 40, inclusive. The rest of each input record contains the first name and the address of the person. For each input record read, you want to create an output record containing only the surname. What's more, you want that surname to start in column 1 of the output record.

Here is the SPECS stage to do it:

```
...| specs 20-40 1 |...
```

The first SPECS operand (`20-40`) tells SPECS where to get the data: from columns 20 through 40 of each input record. The second SPECS operand (`1`) tells SPECS where to put that data in each output record: starting at column 1.

Now suppose you also want to extract the zip code from each input record and put it after the surname in each output record. You need to add a few more operands to SPECS. Assuming that the zip code is in columns 70 through 79 of each input record, here is a SPECS stage to do it:

```
...| specs 20-40 1 70-79 22 |...
```

The operand `70-79` tells SPECS to get a second data item from columns 70 through 79 of each input record. The operand `22` tells SPECS to put that item in each output record starting at column 22 (immediately after the 21-column surname).

The above two examples illustrate two very important aspects of SPECS:

- A group of operands identify and control each data item. The last example above has two groups.
- Within a group, the operand that identifies the input item precedes the operand that identifies where it is to be placed in the output record. These operands are referred to as the *input* and *output* operands.

To describe SPECS, we'll first show you various input operands. That is, we'll show you how to tell SPECS where to get data. Then we'll show you various ways to tell SPECS where to put that data in the output record. In all cases, the input and output operands must be in the following order:

input output

Input Operands—Columns Numbers and Column Ranges: You have already seen isolated SPECS stages that put pieces of input records in the output records. Now let's look at a complete PIPE command in Figure 61.

```
pipe literal abcdefghij | specs 3-5 1 | console
cde
Ready;
```

Figure 61. SPECS Stage Example: Using Numbers to Identify Input

The SPECS operands indicate that columns 3 through 5 of each input record should be written to each output record starting at column 1. As a result, the string `cde` is extracted from the sole input record and is written in columns 1 through 3 of the output record, which CONSOLE then displays.

In the example, none of the other letters in the input record are in the output record. The output records start out empty. The only data in the output records is what you tell SPECS to put in them.

To put additional data from each input record into each output record, use additional groups of operands, as shown in Figure 62 on page 37.

```
pipe literal abcdefghij | specs 3-5 1 1 5 | console
cde a
Ready;
```

Figure 62. SPECS Stage Example: Using More than One Group of Numbers

The operand 1 tells SPECS that you want the data in column 1 of the input record. The operand 5 tells SPECS to put that data in column 5 of the output record.

By default, SPECS fills unassigned columns of the output records with blanks. In the example, SPECS puts data in the output record in columns 1 through 3 and in column 5, but not in column 4. SPECS puts a blank in column 4 for you. The output record length is determined by the last assigned column. In the example, the output record has a length of 5 characters.

You can use any of the usual formats for specifying column ranges. Figure 63 uses the operand 4.5 to identify a column range of 5 characters starting with column 4. It also uses the operand 4-6 to identify columns 4 through 6.

```
pipe literal 000Television | specs 4.5 5 4-6 1 | console
Tel Telev
Ready;
```

Figure 63. SPECS Stage Example: Using Various Column Ranges

Figure 63 also shows that you do not need to build the output record in any particular order. First SPECS puts data starting in column 5, then SPECS puts data starting in column 1. There isn't anything wrong with doing this. It is also valid to refer to columns of the input record more than once. Notice that SPECS refers twice to columns 4, 5, and 6.

An asterisk (*) in the second position of a column range means end of record, just as it does on other stages. The example in Figure 64 uses an asterisk in the column range. It removes characters from columns 1 through 8 of the input record. All characters from column 9 to the end of the input record are copied to the output record starting at column 1.

```
pipe literal 12345678An input record | specs 9-* 1 | console
An input record
Ready;
```

Figure 64. SPECS Stage Example: Specifying the End of the Record

Using numbers to identify columns or column ranges on input records is useful when the data is structured. In inventory records, for example, all part numbers are likely to occur within a specific column range. However, not all data is structured in this way. For processing other forms of data, the WORDS operand is often more useful.

Input Operands—WORDS: The WORDS operand of SPECS lets you select data from input records by word number (or word range) rather than column number (or column range). To SPECS, a *word* is any delimited string of characters. A word

cannot be null. The default delimiter is a blank. To change the delimiter character use the WORDSEPARATOR operand. (WORDSEP and WS are abbreviations for this operand).

Figure 65 shows two examples of WORDS. In both examples, the third word of the input record is put in the output record starting at column 1. The CONSOLE stages display the record. Note that each example, because it is not specified otherwise, uses the default word delimiter of blank.

```
pipe literal See Joe compute. | specs words 3 1 | console
compute.
Ready;
pipe literal See Joe compute. | specs w3 1 | console
compute.
Ready;
```

Figure 65. SPECS Stage Example: Using the WORDS Operand

In the first example, the input operand `words 3` identifies the third word in the input record. Even though the input is expressed in words, the output is expressed in columns. The output operand `1` tells SPECS that you want that word placed in the output record starting in *column* 1. In the second example, the input operand `w3` also indicates the third word. W is an abbreviation for WORDS, and the space between WORDS and the number is optional.

You can also specify a range of words. Indicate word ranges in the same way that you have been indicating column ranges. Figure 66 shows two examples.

```
pipe literal fish dog cat horse | specs word2-3 1  | console
dog cat
Ready;
pipe literal fish dog cat horse | specs word 2-* 1  | console
dog cat horse
Ready;
```

Figure 66. SPECS Stage Example: Specifying Word Ranges

The input operand `word2-3` means words 2 through 3, inclusive (`dog cat` in the example). The input operand `word 2-*` means from word 2 through the end of the record (`dog cat horse`).

Input Operands—FIELDS: The FIELDS operand of SPECS lets you select data from input records by field number (or field range) rather than word or column number (or word or column range). To SPECS, a *field* is any delimited string of characters. A field can be null. The default field delimiter is the tab character, or X'05'. To change the field delimiter character use the FIELDSEPARATOR operand. (FIELDSEP and FS are abbreviations for this operand).

Figure 67 on page 39 shows an example using FIELDSEParator and FIELDS. In this example, the input data contains two fields separated by a dash (-): The CONSOLE stage displays the record.

```
pipe literal torn-asunder| specs fieldsep - f1 1 f2 10 | console
torn      asunder
Ready;
```

Figure 67. SPECS Stage Example: Using the FIELDS Operand

The operand `fieldsep` - identifies the dash as the character delimiting the fields. Operand `f1 1` specifies that the first field in the input record, *torn*, is placed in column 1 of the output record, and operand `f2 10` places the second field, *asunder*, in column 10 of the output record. F is an abbreviation for FIELDS, and the space between F and the number is optional.

You can also specify a range of fields. Indicate field ranges in the same way that you have been indicating column or word ranges. Figure 68 shows two examples.

```
pipe literal fish hook worm water | specs fs blank field2-3 1  | console
hook worm
Ready;
pipe literal AB?CD?EF?GH?IJ?KL?MN | specs fs ? fields -3;-2 5 | console
    IJ?KL
Ready;
```

Figure 68. SPECS Stage Example: Specifying Field Ranges

In the first example, the FIELDSEPARATOR operand `fs blank` defines fields as separated by a blank character. The input operand `field2-3 1` selects fields 2 through 3, inclusive (hook and worm in the example), and places them in the output record starting in column 1. In the second example, the FIELDSEPARATOR operand `fs ?` defines fields as separated by a question mark (?). The operand `fields-3;-2 5` specifies the third-from-last field and the second-from-last field as the input locations to be placed starting in the fifth column of the output record.

You can mix references to columns, to words, and to fields in a single SPECS stage. Use whatever format is most appropriate for the task at hand. There are also ways to specify data that is not from the input records. The remaining sections on input operands discuss how.

Input Operands—Literal Strings: The preceding examples all show how data from the input records can be placed in the output records. Data can come from sources other than the input records. One of those sources is a literal string specified on the SPECS stage itself.

To specify a literal on SPECS, use a delimited string as the input operand instead of a column or word reference to the input record. Delimit the string the same way you delimit strings on other stages (such as LOCATE). Use any character that is not in the string itself and that does not have a special meaning (such as a stage separator). SPECS puts the string on every output record. For example, Figure 69 on page 40 puts the words `Space Captain` in the output record.

```
pipe literal Bob | specs /Space Captain/ 1 1-* 15 | console
Space Captain Bob
Ready;
```

Figure 69. SPECS Stage Example: Specifying A Literal String

As before, read the SPECS arguments in groups:

```
specs   /Space Captain/ 1     1-* 15
        ─────────────────     ──────
                │                │
              Group 1          Group 2
```
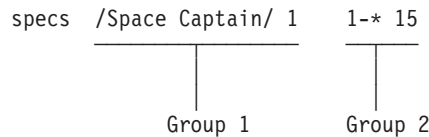
The input operand in the first group tells SPECS that the data to be put on the output record is the string `Space Captain`. Notice that slashes (/) delimit the string. The output operand `1` of that group puts the literal in the output record starting at column 1. The second operand group puts the entire input record in the output record starting at column 15, which happens to be after the `Space Captain` literal string.

So far, the LITERAL stage has been used to put a record in the SPECS examples, but usually the input stream contains many records. SPECS puts the literal string `Space Captain` on each. For example, suppose you have a file named SPACE CADETS listing the following names:

```
Bob
Mark
Joe
Mary
Sue
```

To make all of these people Space Captains, you could enter the following command. In this case, question marks (?) are the delimiters.

```
pipe < space cadets | specs ?Space Captain? 1 1-* 15 | console
Space Captain Bob
Space Captain Mark
Space Captain Joe
Space Captain Mary
Space Captain Sue
Ready;
```

You are not required to use any of the input record in the output record. In Figure 70, for example, SPECS reads two records from its input stream. Data from these input records is not copied to the output records. Instead, only the string `fruit salad` is put in each output record.

```
pipe literal banana | literal melon | specs /fruit salad/ 1 | console
fruit salad
fruit salad
Ready;
```

Figure 70. SPECS Stage Example: Not Using Data from the Input Records

Usually you will use literal strings with regular characters. Occasionally you might need to use hexadecimal characters (perhaps because you cannot type the desired characters on your terminal). To do it, introduce the hexadecimal string as an operand on SPECS, prefix the string of hexadecimal digits with the character x or h,

make sure you have an even number of digits, and end the string with a blank. Because a blank marks the end of the hexadecimal string, there can be no blanks in the hexadecimal string itself.

Figure 71 shows how to put a vertical bar (|) in the output record without it being interpreted as a stage separator when the pipeline is processed. The contents of the input record are put after the vertical bar.

```
pipe literal a line | specs x4f 1 1-* 2 | console
|a line
Ready;
```

Figure 71. SPECS Stage Example: Using a Hexadecimal Literal

So far you have been shown how to use data from input records and literal strings. The last input operand to discuss puts a record number on each output record.

Input Operands—RECNO: To put record numbers in the SPECS output records, use RECNO as an input operand. RECNO causes SPECS to generate a record number in a 10-character field. The number is right-justified in the field and is padded on the left with blanks. The first record has number one (1). The counter is incremented by one for each output record.

Figure 72 shows an example in which the 10-character record number is positioned at column 1 of the output record (`RECNO 1`). The record number is padded on the left with 9 blanks. The input record is put immediately after the record number (`1-* 11`).

```
pipe literal line2|literal line1|specs recno 1 1-* 11 | console
         1line1
         2line2
Ready;
```

Figure 72. SPECS Stage Example: Using the RECNO Operand

Output Operands—Columns and Column Ranges: The next few sections describe various operands you can use to position data in output records. You have already seen how to use column numbers. Figure 73 shows another example.

```
pipe literal Start me at column 15 | specs 1-* 15 | console
              Start me at column 15
Ready;
```

Figure 73. SPECS Stage Example: Using Column Numbers for Output

Figure 74 on page 42 shows what happens when you miscalculate. In this case, `Jones` is copied to columns 1 through 5, and `Raymond` is copied to column 5. `Raymond` overlays the `s` in `Jones`. Because you might want to do something like this intentionally, no error message is produced.

```
pipe literal Raymond Jones | specs 9-13 1 1-8 5 | console
JoneRaymond
Ready;
```

Figure 74. SPECS Stage Example: Overlaying Data

You can also specify a column range for the output operand. In this case, the input field is truncated or padded on the right to fill the output range.

Output Operands—NEXT: Often you will want to put data on the output record in the next available column. Use the NEXT output operand to do it. When NEXT is used, you do not have to count columns.

In Figure 75, the operands `recno 1` put the record number at the beginning of the record. The operands `1-* next` put the contents of the input record in the output record at the next available space. In this case, the next available space is immediately after the record number.

```
pipe literal My record | specs recno 1 1-* next | console
        1My record
Ready;
```

Figure 75. SPECS Stage Example: Using the NEXT Operand

Output Operands—NEXTWORD: NEXTWORD is similar to NEXT. The difference is that NEXTWORD puts a blank before the input data.

NEXTWORD does not put in a blank if there isn't yet data in the output record. Instead, NEXTWORD copies the input in column 1. Figure 76 shows an example.

```
pipe literal flip flop | specs word2 nextword word1 nextword | console
flop flip
Ready;
```

Figure 76. SPECS Stage Example: Using the NEXTWORD Operand

Alignment Operands—LEFT, RIGHT, CENTER: The preceding sections described various input operands and output operands. Another kind of operand is the *alignment operand*. It aligns data within the output record. The alignment operand follows the input and output operand pair, as follows:

```
►►──inputoutput──────────────────────────────────────────────►◄
              └─alignment─┘
```

In the following example, the string `My Summer Vacation` is centered in a range of 80 columns.

```
pipe literal My Summer Vacation | specs 1-* 1-80 center | console
                                  My Summer Vacation
Ready;
```

Figure 77. SPECS Stage Example: Aligning Data

When aligning data, SPECS strips the input field of leading and trailing blanks and aligns what remains of the input field in the output field, truncated or padded as necessary.

Figure 78 shows how to align lines on the right. The output field is from column 1 for 50 columns.

```
pipe literal shorter|literal a long line|specs 1-* 1.50 right| console
                                        a long line
                                            shorter
Ready;
```

Figure 78. SPECS Stage Example: Aligning Data to the Right

By default, data is aligned to the left when a column range is specified for the output operand. Figure 79 shows an example in which the LEFT operand is specified.

```
pipe literal Aligned left | specs 1-* 1.50 left | console
Aligned left
Ready;
```

Figure 79. SPECS Stage Example: Aligning Data to the Left

Conversion Operands: Another kind of SPECS operand is the *conversion operand*. The conversion operand causes SPECS to convert data from one format to another. You can, for example, convert a character input item to hexadecimal, and have the resultant hexadecimal value placed in the output record.

A conversion operand for a data item is specified between the input and output operands for that item. Thus, you now have four kinds of operands that can be specified for a single data item. The order of operands for a given item must be as follows:

```
►►─input─┬──────────────┬─output─┬──────────────┬──────────────────────►◄
         └─conversion─┘          └─alignment─┘
```
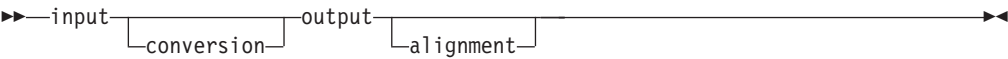
The input and output operands must always be specified. The conversion and alignment operands are optional. If desired, a conversion operand and an alignment operand can be specified for a single data item.

Figure 80 on page 44 shows the first eight bytes of a packed file in hexadecimal. Two output data items are specified. The group of operands for the first item is `1.4 c2x 1`. The group of operands for the second item is `5.4 c2x 10`. The operand `c2x` is the conversion operand for both items.

```
pipe < proc copy | take 1 |specs 1.4 c2x 1 5.4 c2x 10 | console
00140C6F 00000050
Ready;
```

Figure 80. SPECS Stage Example: Converting Data

In SPECS, the operands `1.4 c2x 1` indicate that the first four bytes of the input should be copied to column 1 of the output after being converted from character to hexadecimal (c2x). The string `5.4 c2x 10` converts the next four bytes and positions them at column 10 of the output record.

Figure 81 shows several other conversions. The conversion operand C2B converts data from character to binary. B2C reverses the conversion. X2C converts from hexadecimal to character—it requires an even number of hexadecimal characters.

```
pipe literal 911| specs 1-* c2b 1 | console
111110011111000111110001
Ready;
pipe literal 911| specs 1-* c2b 1 | specs 1-* b2c 1 | console
911
Ready;
pipe literal F9F1F1| specs 1-* x2c 1 | console
911
Ready;
```

Figure 81. SPECS Stage Example: Additional Conversions

The *z/VM: CMS Pipelines Reference* contains more information about using the many conversion operands.

Advanced Uses of SPECS

This section describes several advanced uses of SPECS. You may want to skip this section for now and refer back to it after you have some experience with SPECS. The following sections describe how to combine several input records with SPECS, how to write multiple output records, and how to use relative column references.

Combining Input Records: SPECS lets you process several input records at a time. This is often useful when you want to process groups of related input records. For example, suppose you are processing input records that are consistently grouped as follows:

```
Record 1 of the group contains a name
Record 2 of the group contains a street address
Record 3 of the group contains the a city, state, and zip code
```

Now suppose that for each group of records you want to write one output record that contains the state followed by the name. To do it, you would need to get the name from the first record, skip the second record, and get the state from the third. You can do it with the READ operand of SPECS.

The READ operand causes SPECS to read the next record from the input stream without writing a record to the output stream. Look at the example in Figure 82 on page 45.

```
pipe < address data | console
Smith, Joseph
3211 Titan Drive
Lake Town        NY   11011
Jones, Susan
525 Main Street
Scranton         PA   20192
Ready;
pipe < address data | specs 1-* 4 read read 20.2 1 | console
NY Smith, Joseph
PA Jones, Susan
Ready;
```

Figure 82. SPECS Stage Example: Using the READ Operand

The first PIPE command displays the contents of the file ADDRESS DATA. There are two addresses. Each address takes three records. The second PIPE command displays the desired results.

Let's analyze the SPECS stage. The first group of operands `1-* 4` takes the entire input record, which contains the name, and puts it in the output record starting at column 4. That is all you need to do with the first input record, so you specify a READ operand to read the next input record, which contains the street address. You do not want to do anything with the street address, so you specify a second READ operand.

The operands following the second `read` in SPECS now refer to the third input record. From this third input record, select the state abbreviation. The state abbreviation always starts in column 20 of the input record and is two characters. The operand group `20.2 1` puts the state abbreviation into the first and second columns of the output record.

Notice that you are still working with the same output record even though you have read three input records. After the state is put in the output record, SPECS writes the single output record to its output stream. Then the whole process repeats for the next three input records.

Writing Multiple Output Records: The WRITE operand causes SPECS to write an output record without reading a new input record. It is the converse of READ.

Figure 83 shows an example that produces two output records for every input record read.

```
pipe < winner file | console
NY Smith, Joseph
PA Jones, Susan
Ready;
pipe < winner file | specs word1 1 /state:/ nextword write 4-* 4 | console
NY state:
   Smith, Joseph
PA state:
   Jones, Susan
Ready;
```

Figure 83. SPECS Stage Example: Using the WRITE Operand

Two data items are specified for the first output record: the state, which is taken from the input record, and a literal string. The operands `word1 1` put the state abbreviation in the output record. The operands `/state:/ nextword` put a literal string in the output record after the state.

These two groups of operands build the first output record. To write it, the WRITE operand is specified next. The operands following `write` build the second output record, which starts out empty, just as the first one did. Those operands, `4-* 4`, put the name portion of the input record into the second output record. A second WRITE should not be specified at the end.

You can use both the READ operand and the WRITE operand in a SPECS stage.

Using Relative Column References: SPECS lets you refer to input columns by relative position. For example, when you specify ranges (such as 1-7), the numbers are relative to the beginning of the record. You can also use negative numbers to refer to columns relative to the end of the record. SPECS and ZONE are examples of filters with this facility.

For example, suppose the pipeline contains records of varying lengths. How can you have SPECS write only the last column to the output record? It is not possible with what has been discussed so far. Everything so far has been relative to the beginning of the record. Because the lengths of the records differ, no single column number will give the last column for all input records.

Instead, you need to refer to the last column by giving some number relative to the end of the record. To do so, use a negative column number. When negative column numbers are used in a column range, they must be separated by a semicolon (;). The usual hyphen (-) or period (.) cannot be used. The example in Figure 84 shows a SPECS stage that displays the last column of each record.

```
pipe literal ABCDE| literal abc| specs -1;-1 1 | console
c
E
Ready;
```

Figure 84. SPECS Stage Example 1: Using Negative Relative Column Numbers

The argument pair `-1;-1 1` means that the first column relative to the end of the input record should be copied to column 1 of the output record. The input range of `-1;-1` is a range that refers to a single column. Think of the columns as being numbered backward:

```
ABCDE  <--record
54321  <--column numbers relative to the end of the record

abc    <--record
321    <--column numbers relative to the end of the record
```

Figure 85 on page 47 shows a similar example. The third column relative to the end of the input record is put in the output record at column 5.

```
pipe literal ABCDE| literal abc| specs -3;-3 5 | console
    a
    C
Ready;
```

Figure 85. SPECS Stage Example 2: Using Negative Relative Column Numbers

Suppose you want to see the last two columns. The input range should then be -2;-1. Figure 86 shows the result.

```
pipe literal ABCDE| literal abc| specs -2;-1 1 | console
bc
DE
Ready;
```

Figure 86. SPECS Stage Example 3: Using Negative Relative Column Numbers

You cannot reverse the order of the numbers in SPECS (in the preceding example, specs -1;-2 1). This would make the beginning column of the range to the right of the ending column.

Figure 87 shows what happens when you use a negative column number that is too high. The entire record is returned. When the column number is too high in a positive column range, the same result occurs.

```
pipe literal ABCDE| literal abc| specs -600;-1 1 | console
abc
ABCDE
Ready;
```

Figure 87. SPECS Stage Example: Specifying Range Beyond the Input Record

Figure 88 shows you how to filter records by looking at the ends of the records. The example finds all records ending in x. It is assumed that the file INPUT FILE contains variable-length records.

```
pipe < input file | specs -1;-1 1 1-* next | find x| specs 2-* 1 | console
```

Figure 88. SPECS Stage Example: Looking at the End of a Record

The first SPECS stage copies the last column of each input record to column one of the output record. It also copies the entire input record to the same output record. After the end of the record is moved to the beginning, you can use FIND to select those beginning with x. The second SPECS stage removes the first column of the selected lines, restoring the original contents.

Miscellaneous Filters

This section describes several miscellaneous filters:

- DUPLICATE—duplicates input records.
- COUNT—counts characters, words, and records.
- SORT—sorts records.
- BUFFER—reads all input records before writing them.

Duplicating Records (DUPLICATE)

DUPLICATE makes copies of input records. It reads an input record and writes that record one or more times to its output stream. For DUPLICATE's operand, specify the number of additional copies desired.

Figure 89 makes 2 additional copies of each input record.

```
pipe literal Are we almost there? | literal Dad | duplicate 2 | console
Dad
Dad
Dad
Are we almost there?
Are we almost there?
Are we almost there?
Ready;
```

Figure 89. DUPLICATE Stage Example

Counting Characters, Words, and Records (COUNT)

The COUNT stage counts characters (bytes), words, or records in its input stream. It writes a single record containing the count to its output.

Figure 90 shows an example in which three PIPE commands are entered. The first counts the number of bytes in the file ALL NOTEBOOK, the second counts the number of words, and the third counts the number of lines.

```
pipe < all notebook | count bytes | console
2456
Ready;
pipe < all notebook | count words | console
347
Ready;
pipe < all notebook | count lines | console
67
Ready;
```

Figure 90. COUNT Stage Examples

When counting words, the COUNT stage considers any blank-delimited string to be a word. To COUNT, the TEST DATA file in Figure 91 on page 49 contains 10 words.

```
========================================
Don't worry about me--I can take
care of myself.
```

Figure 91. COUNT Stage Example: Counting Words

The string of equal signs (=) counts as one word, and the string `me--I` also counts as one word (not two).

COUNT lets you count several things at a time, as shown in Figure 92. The operand CHARS is a synonym for BYTES.

```
pipe < all notebook | count chars words lines | console
2456 347 67
Ready;
```

Figure 92. COUNT Stage Example: Counting Several Items

When you specify more than one item, COUNT always returns results in this order: characters, words, lines. The order in which you specify the operands does not change the order of the results (Figure 93).

```
pipe < all notebook | count lines chars | console
2456 67
Ready;
```

Figure 93. The Order of COUNT Results

COUNT is also useful in multistream pipelines. See "COUNT, Revisited" on page 124 for more information.

Sorting Records (SORT)

The SORT stage orders pipeline records. SORT *buffers* the pipeline records; that is, it reads all its input records before writing output records. The sorting is done in virtual storage. SORT gives a message and a nonzero return code if the input is too large to fit in storage.

For example, the following pipeline sorts the names of all files in your search order having a file type of SCRIPT. The names of only the first 10 files are displayed.

```
pipe cms listfile * script * | sort | take 10 | console
ASYNCMS  SCRIPT   A1
BOGUS    SCRIPT   A1
CHKAUTH  SCRIPT   A1
CHKFILE  SCRIPT   A1
COPY     SCRIPT   A1
FGETMSG  SCRIPT   A1
FPACK    SCRIPT   D1
GENERIC  SCRIPT   A1
LOCOR    SCRIPT   A1
LOGREC   SCRIPT   A1
Ready;
```

Figure 94. SORT Stage Example

By default, records are sorted in ascending order. (You can use the ASCENDING operand if you wish.) To sort records in descending order, specify the DESCENDING operand, which is abbreviated to DESC in Figure 95.

```
pipe cms listfile * script * | sort desc | take 10 | console
YRDFTHLP SCRIPT   D1
YEARDFT  SCRIPT   D1
YEAR     SCRIPT   D1
REXXSEND SCRIPT   D1
REXXNOTE SCRIPT   D1
REVOUT   SCRIPT   A1
REQUEST  SCRIPT   A1
QBRAPA   SCRIPT   P1
LOGREC   SCRIPT   A1
LOCOR    SCRIPT   A1
Ready;
```

Figure 95. SORT DESCENDING Stage Example

Using Column Ranges When Sorting

You can also specify column ranges to be used in the sort. The records are sorted by the contents of the column ranges. For example, Figure 96 sorts the names of the files on file mode A by file type, and displays the first ten file names. (The file types in a LISTFILE response start at column 10 and can be up to 8 characters long.)

```
pipe cms listfile * * a | sort 10-17 | take 10 | console
NOT      AUTHORIZ A1
TEMP     DATA     A1
TEST     DATA     A1
TEST1    DATA     A1
TEXT     DATA     A1
DIRADD   EXEC     A1
DIRCOUNT EXEC     A1
DISKSPAC EXEC     A1
DUAL     EXEC     A1
FEXAM    EXEC     A1
Ready;
```

Figure 96. SORT Stage Example: Using a Column Range

To sort in descending order when a column range is used, use DESCENDING after the column range. (See Figure 97.)

```
pipe cms listfile * * b | sort 10-17 descending | take 4 | console
MYBOOK   SCRIPT   B1
ROLLUP   MODULE   B1
DMSC5MST LIST3820 B1
TEST     DATA     B1
Ready;
```

Figure 97. SORT DESCENDING Stage Example: Using a Column Range

Discarding Duplicates When Sorting

Use SORT UNIQUE to discard duplicate records during a sort. Figure 98 shows how to display a list of unique words in a file.

```
pipe < input file | split | sort unique | console
```

Figure 98. SORT UNIQUE Stage Example

The < stage reads the file INPUT FILE. SPLIT puts each blank-delimited word on a separate record. The records are then sorted in alphabetic order. The UNIQUE operand on SORT causes duplicate records to be discarded. Finally, the unique words are displayed on the console.

Counting and Discarding Duplicates While Sorting

The COUNT operand of SORT counts the number of duplicates of each record. It discards duplicates, but adds a count of the number of duplicates to the beginning of each remaining record. The count is right-justified in columns 1 through 10 of each output record. The original input record follows, beginning in column 11.

Figure 99 shows two PIPE commands. The first command shows the contents of the file that is sorted in the second PIPE command.

```
pipe < sample data | console
one
two
two
three
three
three
four
four
four
four
Ready;
pipe < sample data | sort count | specs 1-10 1 11-* nextword | console
         4 four
         1 one
         3 three
         2 two
Ready;
```

Figure 99. SORT COUNT Stage Example: Counting and Discarding Duplicates

Buffering Records (BUFFER)

The SORT stage buffers records in the course of its processing. There are other times you might want to buffer records yourself. To do so, use the BUFFER stage.

BUFFER holds all the records until it has read the last input record. Then BUFFER writes the records to the next stage. Use BUFFER any time the records must be delayed until all input is read.

One such time is when you want to read lines of input from the terminal and write the lines to the program stack. The lines might then be processed by an exec. The STACK stage is a device driver that you can use to read from or write to the program stack.

In the PIPE command in Figure 100, the CONSOLE stage reads records entered at the terminal. The BUFFER stage holds all the records until it reads the last input record. The last input record is the final record the user types before pressing enter twice. Once BUFFER has read all the input records, it then writes the records to the STACK stage.

```
pipe console | buffer | stack
```

Figure 100. BUFFER Stage Example: Stacking Terminal Input Lines

If you remove the BUFFER stage from the PIPE command in Figure 100, the CONSOLE stage reads an input record and writes it immediately to the STACK stage which places the record on the program stack. Because CONSOLE reads not only from the terminal but also from the program stack, the command loops.

BUFFER is also useful when a multistream pipeline stalls. See "Pipeline Stalls" on page 131 for more information.

Chapter 3. Host Command Interfaces

In CMS Pipelines, host command interfaces are stages within a pipeline that run host environment commands. For instance, the CMS command *query disk* displays information about the disks your user ID has accessed. CMS writes the response to your terminal screen when you issue the command directly to CMS, or the response can be captured and processed in a pipeline with a host command interface stage.

There are several stages that can receive a response from a CMS or CP host command or subcommand and can write it to the pipeline, or not. The CP stage sends commands to CP and writes the response to the pipeline. The CMS and COMMAND stages send commands to CMS and intercept the response normally sent to the terminal. This allows you to control command responses and messages that may be issued. For instance, you may choose to suppress error messages while issuing a CP command. The SUBCOM stage passes subcommands to a specified subcommand environment. The STARMONITOR stage connects to the CP *MONITOR system service and writes the data it receives into the pipeline. The STARSYS stage connects using the Inter User Communication Vehicle (IUCV) to a two-way system service like *ACCOUNT and writes the system information it retrieves into the pipeline. The STARMSG stage connects to the CP message system service to intercept console output. A description of STARMSG may be found in Chapter 7, "Event-Driven Pipelines," on page 155.

This chapter introduces these CMS Pipeline's host command interfaces and describes how to use them. Syntax and reference information for all of the host commands are documented in the *z/VM: CMS Pipelines Reference*.

Working with CMS and CP Commands

CMS, COMMAND, and CP are three useful stages that run CMS and CP host environment commands. Instead of displaying the command response, however, they put it in the pipeline. From there, you can use another stage to process the data as you require.

CMS Stage

The CMS stage runs CMS commands and writes the response to its output stream. For instance, suppose you want to save the output of a CMS QUERY DISK command in a file named SPACE DATA A. You would enter:

```
pipe cms query disk | > space data a
```

This pipeline has two stages:

cms query disk
> The CMS stage runs the QUERY DISK command. Instead of displaying the QUERY DISK response, CMS writes the response to its output stream.

> space data a
> This stage writes whatever is in its input stream to the file SPACE DATA A. (The > stage writes data from the pipeline to a file. This kind of stage is called a device driver, and will be described in detail later.)

If you want to have the command output displayed on the terminal *and* saved in a file, add a CONSOLE stage to your pipeline:

```
pipe cms query disk | > space data a | console
```

If the supplied string is not recognized as a CMS command, it is passed to CP. In this case, however, CP writes the response directly to your terminal; the CMS stage does not write the response to its output stream. To put the response of a CP command in a pipeline, use the CP stage instead of CMS.

To run a CMS command from a pipeline *without* having the response records written to the pipeline, use the SUBCOM stage. (See page 55 for a description of SUBCOM.) When processed by SUBCOM, CMS commands are executed as though they were entered from the command line or with a REXX ADDRESS CMS instruction.

COMMAND Stage

The COMMAND stage also issues CMS commands. It passes the command to CMS for execution as if the command were invoked using ADDRESS COMMAND from REXX/VM. You should specify the command in uppercase unless you wish to execute the command with mixed case names or operands.

For instance, suppose you want to save the output of a CMS QUERY DISK command in a file named SPACE DATA A. Using COMMAND, you would enter:

```
pipe command QUERY DISK | > space data a
```

QUERY DISK is executed without the search for execs or CP commands.

CP Stage

To put the response from a CP command in a pipeline, use the CP host command interface. The CP stage passes the specified string directly to CP:

```
pipe cp query users | > users data a
```

The CP stage executes the CP QUERY USERS command and writes the response to its output stream. The > stage puts the data in the file USERS DATA A.

Before the CP stage passes a command to CP, it examines the first word of the command. If the first word contains lowercase letters, the CP stage translates the entire command to uppercase, because CP expects command names and most options in uppercase. If you want the CP stage to pass a command to CP without translating it, write the first word of the command in uppercase.

For example, suppose you want to send a message to user BILL, but you want the message text to be sent as-is:

```
pipe cp MSG BILL Is the product name ChocoMilk or Chocomilk?
```

Putting VM Command Results in REXX Variables

REXX programmers can use the PIPE command to put CP and CMS command responses directly into stemmed arrays, as shown in Figure 101 on page 55:

```
/* Put command response into a stemmed array.          */
'pipe cms listfile * script a',     /* Execute a LISTFILE    */
  '| stem fname.'                    /* Put results in FNAME. */
if rc=0 then
   do i=1 to fname.0
    /* Other file processing.                            */
   end
```

Figure 101. STEMMED ARRAY: Placing Host Command Responses in

The list of files is assigned to the `fname.` stem variable. The variable `fname.0` contains the number of lines the STEM stage reads from its input stream. Note that PIPE is a CMS command, so it should be enclosed by single quotation marks. (More about using PIPE in execs is described later in Chapter 4, "Device Drivers," on page 59.)

Executing Pipeline Records as Commands

The CP, CMS, and COMMAND host command interfaces read input from their input stream as long as they are not the first stage in a pipeline. CP passes its input records to CP for execution. CMS and COMMAND pass their input records to CMS for execution. When the commands to be run are in the input streams, you do not need to specify operands on these stages. If you specify an operand, the command specified as an operand is run *before* the commands read from the input stream.

Figure 102 shows an example in which the CMS stage executes two commands. One of the commands is specified as an operand (`tell * This one first.`). It is executed first. The other command is issued in the pipeline by the LITERAL stage.

```
pipe literal tell * This is second. | cms tell * This one first.
09:50:03  * MSG FROM YOURID  : This one first.
09:50:03  * MSG FROM YOURID  : This is second.
Ready;
```

Figure 102. Executing Multiple Commands

Using Subcommand Environments (SUBCOM)

There can be several subcommand environments active in your session. The XEDIT system editor, for example, sets up a subcommand environment named XEDIT to processes XEDIT subcommands from macros. The CMS subcommand environment processes CMS commands with the full command resolution, as if the command had been typed on your terminal. User-written applications may set up their own subcommand environments. To send commands to these environments from a pipeline, use the SUBCOM stage.

SUBCOM requires at least one argument: the name of the environment you want to send commands to. Following that, you can specify the command to be passed. If there are records in the pipeline, SUBCOM sends them to the specified environment (just as the CMS stage does). SUBCOM also copies each record in its primary input stream to its primary output stream.

SUBCOM does not intercept the output of the commands it sends to subcommand environments. If, for example, you want to run a CMS command without intercepting the output, use `SUBCOM CMS command-string`.

The XEDIT macro in Figure 103 writes information about a file as an XEDIT message. To do this without a pipeline you would need to stack the result of a CMS LISTFILE command, read it, and generate an XEDIT message.

```
/* State a file and show the result as an XEDIT message */
parse upper arg file
if words(file) < 3 then do
   say 'Must specify file name, file type, and file mode'
   exit 1
end

address command
'PIPE',
   'state' file,                        /* Look for file          */
   '| specs /msg / 1 1-* next',         /* Build command          */
   '| literal emsg File' file 'not found', /* Prefix a not found msg */
   '| take last',                       /* Take the last msg      */
   '| subcom xedit'                     /* Execute it             */
exit rc
```

Figure 103. STATE XEDIT: Writing Information about a File

This example also shows a technique for being prepared to display an error message if a stage produces no output. Here, LITERAL generates the error message and inserts it in front of the line generated by the STATE stage, if any. TAKE LAST then selects the response if there is one; only when STATE writes nothing (the file is not found) is the literal line retained.

Connecting with CP System Services

Use stages that connect with CP system services to respond to messages received by the pipeline, or to write records received during the connection.

STARMONITOR Stage

The STARMONITOR stage lets you write lines from the CP *MONITOR system service. Before using STARMONITOR, you need to understand CP's system services. It is described in the *z/VM: CP Programming Services* book. It is also necessary to:

- Use the IUCV directory control statement to authorize yourself to connect to the *MONITOR system service,
- Attach the monitor segment to the virtual machine using the CMS SEGMENT LOAD command, and
- Enable the monitor domains you wish to process using the CP MONITOR command

before issuing STARMONITOR.

STARMONITOR requires you to name the monitor shared segment to be used by specifying a *segment* operand. Make sure you use the same segment name that you specified on the SEGMENT LOAD command which attached the monitor segment to the virtual machine. STARMONITOR, by default, will connect to *Monitor in shared mode, and will collect event and sample data. To connect to *Monitor in shared mode, and to collect only event data, specify EVENTS as the operand to STARMONITOR. To connect to *Monitor in shared mode, and to collect only sample data, specify SAMPLES as the operand to STARMONITOR. To connect to *Monitor with exclusive use of the specified monitor segment, specify EXCLUSIVE as the

operand to STARMONITOR. To suppress records from one or more of the monitor record domains, specify SUPPRESS *hex* as the operand to STARMONITOR. However, enabling the monitor domains selectively will give you a better performance time than using the SUPPRESS operand.

The STARMONITOR stage writes lines it receives from the CP *MONITOR system service as logical records to its primary output stream. Each record begins with the 20-byte prefix defined for monitor records in the MRRECHDR structure. For more information about the structure of monitor records, see the MONITOR LIST1403 file containing the monitor records that was loaded onto your base CP object disk (194) at the time z/VM was installed on your system.

You can use the STARMONITOR stage in a REXX exec to create a log file of the virtual disks in storage created on a system. To accomplish this, the exec collects a subset of the Address Space Created monitor records (domain 3 record 12) and writes the user ID and virtual device number (vdev) of the created virtual disk in storage to an output file. Refer to the example exec that creates this log file in the *z/VM: CMS Pipelines Reference*.

Chapter 4. Device Drivers

In CMS Pipelines, device drivers are stages that move data between your pipeline and the outside world. In this case, the outside world consists of devices (virtual and real) and other system resources (such as files maintained in storage by XEDIT).

Most device drivers can be anywhere in the pipeline. Some must be first, however, and others cannot be first. Be careful when using device drivers that can be placed anywhere in a pipeline (such as, XEDIT, STEM, VAR, and FILEFAST). When first in a pipeline, these device drivers read from the system resource. When used anywhere else in the pipeline, they write to the system resource, often replacing existing data. You can overwrite or destroy data when you misplace these device drivers.

This chapter describes some of CMS Pipeline's device drivers. All of the device drivers are documented in the *z/VM: CMS Pipelines Reference*.

Working with the Terminal (CONSOLE)

The CONSOLE stage reads from the terminal and writes to it. CONSOLE senses where it is in the pipeline. When it is the first stage, it reads from the terminal and writes the records to its primary output stream. When it is in any other stage, it reads records from its primary input stream and writes them to the terminal. CONSOLE also copies the records it reads or writes to the following stage.

Many other device drivers work the same way. That is, they read from or write to the pipeline depending on their position in the pipeline. When device drivers write to a host interface, they always write their output records to their output streams as well, so that the records can be passed to the next stage.

In Figure 104, CONSOLE is not the first stage, so it writes to the terminal.

```
pipe literal Hello out there | console
Hello out there
Ready;
```

Figure 104. CONSOLE Stage Example 1

CONSOLE also writes its input records to the following stage. In the example in Figure 105, the string `Hello out there` is displayed on the terminal and is also written to the file CONSOLE LOG A.

```
pipe literal Hello out there | console | > console log a
Hello out there
Ready;
```

Figure 105. CONSOLE Stage Example 2

When CONSOLE is the first stage, it reads lines from the console and writes them to its output stream. Every time you type data on the CMS command line and press the Enter key, CONSOLE writes that record to its output stream. To end CONSOLE, press Enter without typing anything.

Figure 106 shows an example in which the records entered at the terminal are written to the file TYPETO FILE A. The blank line after `to a file.` is a null line the user entered to end the CONSOLE stage. In other words, the user pressed the Enter key without first typing anything.

```
pipe console | > typeto file a
You can use console to type
to a file.

Ready;
```

Figure 106. CONSOLE Stage Example: Typing to a File

Writing Literal Strings to a Pipeline (LITERAL)

The LITERAL stage is not like the CONSOLE device driver. It doesn't work with a real device. Instead, it lets you write a string to the pipeline. The string written is the string you specify as the operand, including any leading or trailing blanks.

In Figure 107, LITERAL writes the string `Hello, World.` to its primary output stream. CONSOLE, which is the next stage, displays the string on your terminal.

```
pipe literal Hello, World.|console
Hello, World.
Ready;
```

Figure 107. LITERAL Stage Example 1

After writing the operand to its primary output stream, LITERAL copies any records in its primary input stream to its primary output stream. Therefore, you can use LITERAL to put a header on your output.

Figure 108 shows how to combine CMS and LITERAL stages to write a heading for the output of a CMS command.

```
pipe cms query accessed | literal My accessed disks and directories: | console
My accessed disks and directories:
Mode  Stat  Files  Vdev  Label/Directory
A     R/W      47   191  BAR191
C     R/O    1152   19C  ESA19C
G     R/O    4729   19F  NUGOOD
S     R/O     351   190  CMS11
Y/S   R/O     378   19E  19ESP4
Ready;
```

Figure 108. CMS, LITERAL, and CONSOLE Stages Example

The CMS stage writes the response from the QUERY ACCESSED command to its primary output stream a record at a time. Before LITERAL processes its input, it

writes its operand to its primary output stream. In this example, LITERAL writes `My accessed disks and directories:` to its primary output stream. Then it copies its primary input stream (the results of the QUERY ACCESSED command) to its primary output stream. The CONSOLE stage writes the records in its primary input stream (the header followed by the command results) to the screen.

Note that the end of the literal string is the last character before the stage separator (|). In Figure 109 the output on the display has two trailing blank characters (though you normally do not see them on the screen). This is not significant in this case but it can be important when records are modified in the pipeline. The first space after `LITERAL` is not part of the literal string. Any additional spaces, however, are included at the left of the record written.

```
pipe literal ... world.|literal Hello...  | console
Hello...
... world.
Ready;
```

Figure 109. LITERAL Stage Example 2

In Figure 109, the second LITERAL stage writes its string (`Hello...`) to its output stream before copying the records from its input stream (`...world.`). Because LITERAL works this way, header records are often added to data near the end of the pipeline, not near the beginning as one might expect.

Working with CMS Files

CMS Pipelines provides many stages for working with CMS files. They are:
- < (read file)—reads a CMS file and writes the records to its output stream.
- > (write file)—reads records from its input stream and writes them to a file (replacing any existing file).
- >> (append file)—reads records from its input stream and writes them to a file (appending any existing file).
- FILEFAST—when used as the first stage, reads a CMS file and writes the records to its primary output stream. Otherwise, FILEFAST reads records from its primary input stream and writes them to a file (appending any existing file) and to its primary output stream, if connected.
- FILESLOW—when used as the first stage, reads a CMS file beginning at a specified record number and writes the records to its primary output stream. Otherwise, FILESLOW reads records from its primary input stream and writes them to a file (appending any existing file) starting at a specified record number. FILESLOW also writes the records to its primary output stream, if connected.
- FILEBACK—reads a file backward (that is, from its last record to its first) and writes those records to its output stream.
- FILERAND—reads specific records or a range of records from a file and writes those records to its output stream.
- FILEUPDATE—replaces specific file records.

This chapter describes <, >, >>, and FILEFAST. The other stages are described in the *z/VM: CMS Pipelines Reference*.

CMS Pipelines works with files that reside on minidisks or in SFS directories. The stages need file mode letters, so the minidisk or directory must be accessed.

The < Stage

The < (read file) stage reads a CMS file and writes the records to its output stream. The < stage must be used as the first stage of a pipeline. Specify a file identifier as the operand.

Figure 110 reads a file named DAY LIST and writes the records to its output stream. The CONSOLE stage displays the records.

```
pipe < day list | console
Morose Monday
Tranquil Tuesday
Wonderful Wednesday
Tumultuous Thursday
Fabulous Friday
Spectacular Saturday
Sedate Sunday
Ready;
```

Figure 110. < Stage Example

When using the < stage, do not forget to leave a blank between < and the file identifier.

The file mode is optional for the < stage. If the file mode is omitted, the < stage looks for the file in virtual storage (as loaded by the EXECLOAD command). If the file is not there, < looks for it on your accessed disks or directories using the usual CMS search order.

The < stage reads both F-format and V-format files.

The > Stage

The > (write file) stage reads records from its input stream and writes those records to a file. If the file exists, it is replaced. If the file does not exist, it is created. Specify the file identifier as an operand—you must specify a file mode. A > stage cannot be the first stage of a pipeline.

Like the CONSOLE stage, the > stage copies its input stream to its output stream for use by any following stage. All output device drivers work this way. In fact, both the following examples yield the same results:

```
pipe cms query disk | > space data a | console
```

```
pipe cms query disk | console | > space data a
```

The order of the > and CONSOLE stages is switched. CONSOLE displays the QUERY DISK response on your terminal, but also writes the records to its output stream. The input stream of the > stage is connected to the output stream of CONSOLE. So, the > stage reads the response records and writes them to the SPACE DATA A file.

In Figure 111 on page 63, the < stage reads the file DAY LIST, writing the records to its output stream. Then the > stage reads those records from its input stream and writes them to the file NEWDAY LIST A.

```
pipe < day list | > newday list a
Ready;
```

Figure 111. > Stage Example

By default, the > stage creates a V-format file. (You can specify the operand VARIABLE after the file identifier.) Use the FIXED operand to create a file having F-format records. Type the desired record length after FIXED, as shown in this example:

```
pipe literal Peppers | pad 80 | chop 80 | > garden list a fixed 80
```

The `FIXED 80` operand causes the file GARDEN LIST A to be created as an F-format file with a record length of 80. When FIXED is used, all input records to the > stage must be the same length (that is, the length specified after the FIXED operand). Any record that is longer or shorter causes an error.

You can omit the record length from FIXED. In this case, the length of the first input record determines the record length. If all the records do not have the same length, an error results.

To force records to have a specific length, use the PAD and CHOP filters as shown in the preceding example. PAD ensures that the record is at least 80 bytes long; CHOP truncates any records longer than 80 bytes. CHOP is superfluous in the above example because you know the record `Peppers` is less than 80 bytes long.

The >> Stage

The >> (append file) stage reads records from its input stream and writes those records to a file. If the file already exists, the >> stage appends the records to the file. If the file does not exist, the >> stage creates it. Specify the file identifier as an operand—you must specify a file mode. A >> stage cannot be the first stage of a pipeline.

The following exec fragment tracks the use of an exec. It writes the name of the exec, the date, and the time to a log file:

```
:
:
/* Append a record to the log file      */
'pipe',
   'literal MYEXEC run on' date() time(),
   '| >> myexec log a variable'
:
:
```

When the >> stage creates a file, it creates a V-format file by default. You can omit keyword VARIABLE from the example above and get the same result. To create an F-format file, specify the FIXED operand just as you do for the > stage. If you use the FIXED operand, make sure all of the records to be written are the same length.

If you are appending a file that has fixed-length records, the records you append must be the same length as those in the file. Use CHOP and PAD stages to fix the lengths of pipeline records. (See page 63 for examples using the FIXED operand.)

The FILEFAST Stage

The FILEFAST stage performs different functions depending on its position in the pipeline. When used as the first stage, it performs a function similar to the < stage. That is, it reads a file and writes the records to its output stream. Figure 112 on page 64

page 64 shows an example.

```
pipe filefast day list | console
Morose Monday
Tranquil Tuesday
Wonderful Wednesday
Tumultuous Thursday
Fabulous Friday
Spectacular Saturday
Sedate Sunday
Ready;
```

Figure 112. FILEFAST Stage Example

There are some differences between the FILEFAST stage and the < stage, as follows:

- The FILEFAST stage will *not* display an error message if the file to be read does not exist. The < stage does display an error message.
- The FILEFAST stage does not look in virtual storage for the file. It looks only through the CMS search order. In contrast, the < stage does look for a file loaded by EXECLOAD before looking for it in the CMS search order, if the file mode is omitted.

When FILEFAST is not the first stage, its function is identical with that of the >> stage.

Unless you want to exploit the differences outlined above, it is recommended that you use < and >> instead of FILEFAST.

Getting Facts about Files (STATE, STATEW)

To get facts about several files, use the CMS stage to run a LISTFILE command (see Figure 28 on page 25 for an example). Use the STATE or STATEW stages when you are interested in a single file or the first file with a particular name and type.

The STATE and STATEW device drivers get information about a file and write a line in the same format as the response from a CMS LISTFILE command with the DATE option. Use STATE or STATEW instead of CMS LISTFILE when you wish to find the first file of a certain type, or obtain the date of a file or a list of files.

STATE looks on all accessed file modes, while STATEW looks on only file modes that are accessed read/write.

The format of the input lines (and the operand string, if present) is three words separated by blanks (or *tokens*): a file name, a file type, and a file mode. Use an asterisk (*) for the file name or the file type (or both) to find the first occurrence of a certain type of file. This asterisk, or wildcard character, is supported by both the CMS STATE and CMS STATEW commands, but will give you a different result than if you used it on the CMS LISTFILE command. STATE and STATEW return information for only the first file that matches the argument, not *all* files.

STATE and STATEW work like the CMS and CP stages; they process any operand first, then process the records from their input streams. STATE and STATEW expect the records to contain file identifiers. One file identifier should be on each input

record. Figure 113 shows an example.

```
pipe literal test * * | state profile * * | console
PROFILE  EXEC    A1 V        72          65          2  5/14/91 15:42:49
TEST     D       A1 V        80          40          2  8/29/90 13:47:15
Ready;
```

Figure 113. STATE Stage Example

In execs, use STATE and VAR to put file information into stem variables that can be used in the body of the exec. (See Figure 114.)

STATE writes one output record for each file that it finds. It does not write an output record for files it cannot find. In contrast, the CMS command LISTFILE writes a line for each file matching the argument pattern; it can write several lines for a single pattern.

```
/* State subroutine                                              */
state: procedure
parse arg fn ft .

'pipe state' fn ft '*',    /* Check for existence                    */
   '| specs 1-22 1',       /* Put file ID and record format in output record */
      '28.7  next ',       /* Put the record length next             */
      '37-44 next ',       /* Put the number of records next         */
      '56-*  next',        /* Put date and time next                 */
   '| var state'           /* Put it in variable STATE               */
If RC/=0
   Then call err RC, 'File' fn ft '* not found'
return right(state, 80)
```

Figure 114. STATE and VAR Stages Example

Packing and Unpacking Files

CMS Pipelines lets you read CMS files that are packed. It also lets you create packed files. To learn how, refer to "Packed Format (PACK, UNPACK)" on page 189.

Accessing Exec Variables

Several device drivers read and write variables in a REXX or EXEC 2 program. You can read and write a single variable or a stemmed array.

STEM Stage

STEM lets a pipeline use stem variables in REXX programs. When first in a pipeline, STEM reads the contents of variables and writes them to its primary output stream. In other positions it reads records from its primary input stream and writes them to REXX stem variables, as well as to its primary output stream.

The argument to STEM is the part of the stemmed variable up to and including the last period (.) in the variable name you wish to reference. The name must include the period, for instance:

ARRAY.NAME.

When STEM is the first stage, it uses the variable *stem*.0 to determine how many records to write to its output stream. In Figure 115, STEM looks at `inval.0` to determine how many variables to read. Because `inval.0` is set to 3, STEM reads the variables `inval.1`, `inval.2`, and `inval.3`. It writes the contents of these variables to its output stream (one record per variable). CONSOLE displays the records.

```
/* STEMFRST EXEC -- STEM as the first stage */
inval.0 = 3
inval.1 = 'red'
inval.2 = 'white'
inval.3 = 'blue'
inval.4 = 'green'
'pipe',
   'stem inval.',
   '| console'
exit rc
```

Figure 115. STEMFRST EXEC: Using STEM Stage to Read REXX Variables

The following example show the results of executing STEMFRST. Because `inval.0` is 3, not 4, `green` is not displayed on the terminal.

```
stemfrst
red
white
blue
Ready;
```

When STEM is not first, it reads its input records and writes them to the specified stem variable. STEM also sets *stem*.0 to the number of variables set. In Figure 116, for example, the variable `outval.0` is set to 3.

```
/* STEMMID EXEC -- STEM in a stage other than the first */
'pipe',
   'literal blue',
   '| literal white',
   '| literal red',
   '| stem outval.'
do i=0 to outval.0
   say 'OUTVAL.'i "=" outval.i
end
exit rc
```

Figure 116. STEMMID EXEC: Using STEM Stage to Write REXX Variables

The following example show the results of executing STEMMID. STEM writes the three records in its input stream to the stem variable `outval.`.

```
stemmid
OUTVAL.0 = 3
OUTVAL.1 = red
OUTVAL.2 = white
OUTVAL.3 = blue
Ready;
```

Often you will use STEM when you want to use the REXX language to do some processing that is not easily done in CMS Pipelines. You use STEM to temporarily leave CMS Pipelines. In Figure 117 on page 67, for example, stages read a file and

preprocess the data (using < and LOCATE). STEM puts the preprocessed records in a stemmed variable so that they can be counted. If more than 20 records are counted, STEM is used again to gather the records, which are then written to a file.

Note that REXX determines how to end CMS Pipelines processing.

```
/* Find 'the the'                                        */
'pipe',
   '< mybook script',      /* Read a file                */
   '| locate /the the/',   /* Locate records with 'the the' */
   '| stem line.'          /* Put records in a stem variable */
if line.0 > 20 then        /* Now let's use REXX for a while */
   'pipe',                 /* More than 20 errors...     */
     'stem line.',         /* Get variables              */
     '| > error file a'    /* Write to file              */
else
   'pipe',                 /* Less than 20 errors...     */
     'stem line.',         /* Get variables              */
     '| console'           /* Display them               */
```

Figure 117. STEM Stage Example: Using REXX with CMS Pipelines

By using STEM, you can take advantage of the strengths of both CMS Pipelines and REXX. In a single exec, you can switch between CMS Pipelines and REXX as needed, using whatever is most effective for solving the problem at hand.

Let's take another example that is somewhat less contrived. Figure 118 shows an exec that counts the number of files on all accessed minidisks. The response to QUERY DISK is processed to extract the number of files on each minidisk and store these numbers in variables $num\_files.1$, $num\_files.2$, and so on. $num\_files.0$ is set to the number of the last line stored; it is set to zero when there is no input to STEM.

```
/* COUNTFIL EXEC -- Count the files on all accessed minidisks */
'pipe',
   'command QUERY DISK',   /* Issue command              */
   '| drop 1',             /* Drop title                 */
   '| specs 37.8 1',       /* Take number of files       */
   '| stem num_files.'     /* Set variable               */

 total_files = 0
 do i = 1 to num_files.0
    total_files = total_files + num_files.i
 end
 say 'You have' total_files 'files across your',
     num_files.0 'minidisks.'
```

Figure 118. COUNTFIL EXEC: Using STEM Stage

Our final STEM example is in Figure 119 on page 68. The exec in the example searches for the last available disk mode letter (Z, X, ...).

```
/* FINDFM EXEC -- Find the last available mode */
'pipe',
   'command QUERY DISK',
   '| drop 1',
   '| specs 13.1 1.2',
   '| join *',
   '| stem modes.'

allmodes='ZYXWVUTSRQPONMLKJIHGFEDCBA'
p=verify(allmodes, modes.1)
if p=0
   Then call err 36, 'No available mode for access'

mode=substr(allmodes,p,1)
say 'The last available mode is 'substr(allmodes,p,1)
```

Figure 119. FINDFM EXEC: Using STEM Stage

The pipeline queries the accessed disks, drops the header line and selects the mode letter from column 13. Then it puts all records together into a single record.

So STEM gets one record which it assigns to the variable modes.1. It also sets modes.0 to one so you know how many stemmed variables were set (like the XEDIT EXTRACT subcommand).

The rest of the exec fragment shows how to find the first mode that is not in the list of accessed modes and how to obtain that mode. The list of mode letters is reversed to make them appear in the order you wish to look for a letter. As used here, the REXX function VERIFY returns the position of the first character in the first argument that is not in the second one (or zero if all characters of the first argument are present in the second one). Because the second argument is the list of accessed mode letters, and the first argument is the list of all possible modes in reverse order, you get the index to the last mode letter that is not accessed, or zero if all 26 disks are accessed.

Even though only one pipeline record is being saved in the example, STEM is used instead of VAR. VAR drops the REXX variable if there isn't a record in the input stream, but STEM does not. To avoid the risk of having the variable dropped, STEM is used.

VAR Stage

When used first in a pipeline, VAR reads the value of a single REXX variable and writes it to its output stream. When used elsewhere, VAR reads its first input record and writes it to the specified exec variable. VAR also copies its input stream to its output stream (including the first record.)

When VAR is not first in the pipeline, it drops the variable if it gets no input. (When a REXX variable is dropped, it becomes unassigned—it is restored to its original uninitialized state.)

DISKSPAC EXEC (Figure 120 on page 69) shows the use of VAR. DISKSPAC displays the percentage of minidisk space in use for a given file mode letter. When entered for a file mode associated with an SFS directory, DISKSPAC displays a message. (The concept of *percentage in-use* does not apply to a single SFS directory.)

```
/* DISKSPAC EXEC -- Display the percentage of in-use disk space      */
signal on novalue
parse arg fm
if fm='' then fm='A'                   /* No file mode?  Use A        */

'pipe',
   'cms query disk' fm,                /* Issue QUERY DISK            */
   '| drop 1',                         /* Drop the title line         */
   '| specs 57-58',                    /* Get the value in cols 57 & 58 */
   '| var diskspace'                   /* Put data line in VAR        */

if datatype(diskspace) ¬= 'NUM'        /* Check if data is a number   */
  then say 'SFS Directory'             /* None for SFS directory      */
  else say diskspace                   /* Okay, display it            */
exit rc

novalue:
   say 'File mode not accessed'
   exit
```

Figure 120. DISKSPAC EXEC: Using VAR Stage

In the DISKSPAC EXEC, the stages of a PIPE command issue a QUERY DISK command, select the data record from the response, and write that data record to the variable `diskspace`. Regular REXX instructions parse the response and display the appropriate message.

REXX interprets the entire PIPE command before executing it. If you use a variable in an expression in the PIPE command, REXX resolves the variable before executing the PIPE command. Therefore, if you change the value of the variable in the PIPE command, the value is changed *after* the PIPE command is interpreted. The following example fails because `fid` is null at the time that REXX substitutes the variables in the pipeline.

```
/* Exec fails because FID is null */
fid=''
'pipe',
   'literal MYFILE DATA A',    /* Put the file ID in the pipeline */
   '| var fid',                /* Assign it to variable FID       */
   '| drop 1',                 /* Drop the record                 */
   '| append <' fid,           /* Read the file (FAILS)           */
   '| console'
```

Working from XEDIT

CMS Pipelines provides device drivers that work with XEDIT. The XMSG stage issues XEDIT messages. The XEDIT stage accesses XEDIT files. XEDIT must already be running before you can use the XMSG or XEDIT device drivers. (You cannot use the CMS stage to execute the CMS XEDIT command and then use the XMSG or XEDIT device drivers later in the pipeline.)

Issuing XEDIT Messages (XMSG)

The XMSG stage issues XEDIT messages. To try it, edit a file and enter a PIPE command from the XEDIT command line.

```
====> pipe literal Hello XEDIT user. | xmsg
```

The string `Hello XEDIT user.` should be displayed as an XEDIT message.

Accessing XEDIT Files (XEDIT)

The XEDIT stage reads lines from files in the active XEDIT file ring and write lines to them. XEDIT must already be running before you can do this. The operands file name, file type, and file mode are optional to identify the file you want to access. XEDIT reads from or writes to the current file if you do not specify the file.

Reading from XEDIT

The XEDIT device driver reads from the file when it is the first stage in the pipeline. Reading starts at the current line and respects the scope and range settings. You always get the complete line irrespective of the verify setting. The current line pointer is set after the last line read, or at the end of the file if reading continues to end-of-file. (Reading stops before end-of-file if, for instance, a TAKE filter is put after XEDIT.)

Try this example. While editing a file with more than nine lines, enter the following command on the XEDIT command line.

```
====> pipe xedit | take 9 | console
```

You should see nine lines of the current file displayed on the console in line mode (or in the CMS output window).

You must put the current line pointer at the top of the file to be sure you read the entire file.

If the current line is at the top of the file, you can count the number of lines in your XEDIT file using:

```
====> pipe xedit | count lines | xmsg
```

Writing to XEDIT

You can also write to the copy of the file XEDIT holds in storage, but be careful not to overwrite lines you wish to keep. XEDIT writes the line at the current line pointer, advancing the pointer after each record is written. Records are added to the file when the current line pointer is after the last line. You cannot insert lines between already existing lines with the XEDIT stage (although you can use SUBCOM XEDIT to issue the XEDIT INPUT subcommand to insert lines).

Another thing to consider is the record length. If the file is F-format, the records written must have the same length as the record length of the file; they are not padded or truncated automatically. For V-format files you have more leeway. Lines longer than the truncation column are processed according to the spill setting; records *are* added when the spill function is activated. Records longer than the file's truncation column are truncated without a diagnostic message if SPILL is off.

If you start with a new file (no records in it) then everything the XEDIT stage writes is added to the end of the empty file. Note that the file must be in the XEDIT ring before you start writing to it. Use the XEDIT subcommand of XEDIT (*not* the XEDIT stage of CMS Pipelines) to create a file. If you want to be sure the file is empty when you start, use the name of a file that does not exist.

It is a good idea to code the file name, file type, and file mode as XEDIT stage operands so you are sure that data is added to the correct file. This is particularly important in XEDIT macros when you do not know which files are in the ring.

Figure 121 shows a simple XEDIT macro named TRAILER. It adds a record to the end of the file being edited.

```
/* TRAILER XEDIT -- Put trailer in file.                   */
'command locate *'   /* Go to the bottom of the file       */
'pipe',
   'literal .* File updated by' userid() 'on' date(),
   '│ xedit'
exit rc
```

Figure 121. TRAILER XEDIT: Putting a Trailer Record in a New File

Figure 122 shows another example. The example, SNIP XEDIT, copies lines from the file being edited to the end of a file named SNIP SCRIPT. After SNIP XEDIT ends, the file SNIP SCRIPT is the active editing session. To keep the example simple, we assume the files are V-format and we do not check for possible truncation (see preceding text).

```
/* SNIP XEDIT -- Append lines to SNIP SCRIPT                       */
parse arg num .                       /* Read number of lines        */
if num='' then num=1                  /* Use 1 if nothing is specified */
'extract /fname/ftype/fmode'          /* Get the ID of the file      */
'command xedit snip script a'         /* Edit SNIP SCRIPT            */
'command locate *'                        /* Go to bottom of file     */

'pipe',
   'xedit' fname.1 ftype.1 fmode.1,  /* Read the original file      */
   '│ take' num,                      /* Take specified number       */
   '│ xedit snip script a'            /* Add records to SNIP         */
```

Figure 122. SNIP XEDIT: Appending Text to Another File

Combining Records from Device Drivers

In this section we'll be discussing two device drivers: APPEND and PREFACE. These device drivers do not interact with a real device. Instead, they let you run other device drivers.

This may not seem very useful until you consider that only one device driver can be first in a pipeline. If you want another device driver to behave as it does when it is the first stage, use APPEND or PREFACE.

APPEND Stage

APPEND copies all records in its primary input stream to its primary output stream. Then APPEND runs a stage or subroutine pipeline specified as an operand, writing the resultant records to its primary output stream. Thus, the APPEND stage appends the output of a stage or subroutine pipeline to the records that are already in the pipeline.

The stage that you specify as an APPEND operand can be any device driver or host command interface that can be first in a pipeline. In Figure 123 on page 72, the output of a CMS LISTFILE command is added to the pipeline after the output of CMS QUERY DISK.

```
pipe cms query disk | append cms listfile * script a | console
LABEL  VDEV M  STAT  CYL TYPE BLKSIZE  FILES  BLKS USED-(%) BLKS LEFT  BLK TOTAL
BAR191 191  A   R/W    3 3380 4096       78         98-22       352        450
ESA19C 19C  C   R/O  100 3380 4096     1011       3440-23     11560      15000
NUGOOD 19F  G   R/O  150 3380 4096     4736      19522-87      2978      22500
CMS11  190  S   R/O   86 3380 4096      350       9197-71      3703      12900
19ESP4 19E  Y/S R/O  200 3380 4096      832      13531-45     16469      30000
EMPLOYEE SCRIPT    A1
LEGUMES  SCRIPT    A1
LOWER    SCRIPT    A1
RECORDS  SCRIPT    A1
SAMPLE   SCRIPT    A1
SNIP     SCRIPT    A1
VMLETTER SCRIPT    A1
Ready;
```

Figure 123. APPEND Stage Example

Figure 124 shows how to use APPEND to write the contents of two files as a single file:

```
pipe < file1 script a | append < file2 script a | > big script a
Ready;
```

Figure 124. Appending Two Files

The < stage writes the contents of the file FILE1 SCRIPT A to its output stream. APPEND copies these records from its input stream to its output stream. Then APPEND runs the second < stage, writing the resultant records to its output stream. Therefore, the records of FILE1 SCRIPT A precede the records of FILE2 SCRIPT A in the pipeline. Finally, the > stage reads the records from its input stream and writes them to BIG SCRIPT A. BIG SCRIPT A contains the records of FILE1 SCRIPT followed by those of FILE2 SCRIPT.

This example shows another way to combine two literals:

```
pipe literal Hello...|append literal ... world.| console
Hello...
... world.
Ready;
```

PREFACE Stage

PREFACE runs a stage or subroutine pipeline specified as an operand, writing the resultant records to its primary output stream. Then it copies all records in its primary input stream to its primary output steam. Thus, the PREFACE stage prefaces the records in the pipeline with the output of a stage or subroutine pipeline.

Like APPEND, the stage that you specify as a PREFACE operand can be any device driver or host command interface that can be first in a pipeline. For example, to insert the output of a CMS LISTFILE command before the output of CMS QUERY DISK:

```
pipe cms query disk | preface cms listfile * script a | console
EMPLOYEE SCRIPT   A1
LEGUMES  SCRIPT   A1
LOWER    SCRIPT   A1
RECORDS  SCRIPT   A1
SAMPLE   SCRIPT   A1
SNIP     SCRIPT   A1
VMLETTER SCRIPT   A1
LABEL  VDEV M  STAT  CYL TYPE BLKSIZE  FILES  BLKS USED-(%) BLKS LEFT  BLK TOTAL
BAR191 191  A   R/W    3 3380 4096        78      98-22        352        450
ESA19C 19C  C   R/O  100 3380 4096      1011    3440-23      11560      15000
NUGOOD 19F  G   R/O  150 3380 4096      4736   19522-87       2978      22500
CMS11  190  S   R/O   86 3380 4096       350    9197-71       3703      12900
19ESP4 19E  Y/S R/O  200 3380 4096       832   13531-45      16469      30000
Ready;
```

Figure 125. PREFACE Stage Example

One final observation: there is never any need to specify a LITERAL stage as a
PREFACE operand. Specifying the LITERAL stage directly yields the same results.

Chapter 5. Writing Stages

> This chapter contains Programming Interface and
> Associated Guidance Information.

When you need to do something that can't be done with built-in stages, it's time to write your own stage. While stages that you write can do any function you want, most of them will filter pipeline data in some way. User-written stages are used in pipelines the same way that built-in stages are used—their names are specified in stages. They are indistinguishable from built-in stages.

User-written stages are written in the REXX or Assembler language. In some ways, writing a stage is easier than writing a regular exec. In regular execs you must write all the code for the I/O devices. You need to know what system interface works with the device and how to use that interface in an exec. When writing a stage that filters data, you don't have to worry about devices. The stage reads records from its input stream and writes records to its output stream.

Reading and writing pipeline records insulates your stage from others. Stages do not call and pass data directly to each other. They work only with pipeline records. This independence not only makes them easy to write, it makes them easy to reuse—your stages can be used in any pipeline.

Keep your stages small and simple. They will be easier to write, easier to test, and less likely to have errors. Moreover, others will be able to use your stages in ways you haven't considered.

Note: Keep in mind that a sequence of several built-in stages could run faster than a user-written stage that performs the same function.

Stage Concepts

Writing a stage is similar to writing any other REXX-language or Assembler-language program. Your use of the REXX or Assembler language is not restricted. Specific points about writing stages in the REXX or Assembler language are described next.

REXX Stages

Stages written in the REXX language can use all REXX keyword instructions and functions, and you can execute CMS or CP commands from within the stage.

Assembler Stages

Assembler user-written stages provide increased performance over REXX user-written stages, especially when used to process large amounts of data. You must write a stage in Assembler if it uses interfaces that are not offered in REXX, or if it must start at a commit level below -127. CMS Pipelines offers *pipeline Assembler macros* that are the building blocks for writing user-written stages in Assembler. Writing a stage in Assembler is similar to writing a relocatable and re-entrant Assembler-language program. The pipeline concepts that apply to writing stages in REXX also apply to writing stages in Assembler.

Interaction with CMS Pipelines

The significant difference between stages and other REXX- or Assembler-language programs is that stages also interact with CMS Pipelines.

Your stage interacts with CMS Pipelines in three ways:

- Upon gaining control from CMS Pipelines, an Assembler user-written stage's registers contain information that CMS Pipelines provided about the pipeline.
- By executing *pipeline subcommands* or pipeline Assembler macros.
- By passing a return code to CMS Pipelines on exit.

Pipeline Subcommands

There are many pipeline subcommands. Several are covered in this chapter:

- READTO—reads a record from an input stream.
- OUTPUT—writes a record to an output stream.
- PEEKTO—looks at a record in an input stream without removing it from the stream, as READTO does.
- SHORT—copies all remaining records in the input stream directly to the output stream.
- STAGENUM—returns in variable RC a number indicating the position of your stage in the pipeline.
- CALLPIPE—runs a *subroutine pipeline*.

Pipeline subcommands are analogous to XEDIT subcommands. Like XEDIT, the PIPE command sets up its own subcommand environment, and pipeline subcommands in the stage interact with CMS Pipelines.

The pipeline subcommands give a return code in variable RC. The return codes are the only communications you receive from CMS Pipelines, so it is important for your stages to examine them.

Pipeline subcommands can be run within REXX- or Assembler-user-written stages. The PIPCMD stage is used to issue CMS Pipelines subcommands within REXX user-written stages. The Assembler macro PIPCMD can also issue CMS Pipelines subcommands, but within Assembler user-written stages.

Pipeline Assembler Macros

There are many pipeline Assembler macros. A list of macro names supported by z/VM and a brief description of their function follows: Macros are found in FPLGPI MACLIB.

- PIPCMD—issues CMS Pipelines subcommands to control pipelines.
- PIPCOMMT—increases a stage's commit level or obtains the current aggregate return code.
- PIPDESC—describes the Assembler stage to run.
- PIPEPVR—declares what contains the address that points to a table of addresses required by pipeline Assembler macros.
- PIPINPUT—reads a record and removes it from the currently selected input stream.
- PIPLOCAT—obtains the address and length of the next record in the currently selected stream without removing the record.
- PIPOUTP—writes a record to the currently selected output stream from a buffer.

- PIPSEL—designates the stream specified to be the currently selected stream for subsequent use by Assembler macros that reference streams.
- PIPSEVER—detaches the connected, currently selected stream from the stage in the pipeline that issued the PIPSEVER assembler macro.
- PIPSHORT—copies all remaining records in the currently selected input stream directly to the currently selected output stream.
- PIPSTRNO—returns the stream number of a stream specified by number or name.
- PIPSTRST—returns status information about the specified stream to determine whether or not there is input or output data.

Entry Conditions to an Assembler Stage

Before CMS Pipelines calls the Assembler user-written stage, it loads the following general registers with these initial values:

Register	Contents
R0	Address of four fullwords in storage that point to the user-written stage name, the parameter string, and the name of the next stage in the pipeline. The fourth fullword consists of zeros.
R1	Address of Work Area
R2	Address of the beginning of the user-written stage's parameter string
R3	Length of the user-written stage's parameter string
R4	Highest numbered stream allowed
R5	Address that points to a table of addresses required by pipeline Assembler macros to locate entry points into CMS Pipelines
R6	Message Level
R7	0
R8	Number indicating the position of the stage within the pipeline
R9	Address that points to a table of addresses required by pipeline Assembler macros to locate entry points into CMS Pipelines
R10	Address of PIPDESC macro expansion
R11	0
R12	Address of the entry point defined on the PIPDESC macro
R13	Address of the beginning of the 18 fullword (or greater) save area
R14	Address of instruction to return to when the stage completes processing
R15	Address of the entry point defined on the PIPDESC macro

These registers provide information about the user-written stage such as its position in the pipeline and the highest number of streams allowed. The information provided by register 4 is equivalent to the information obtained from the pipeline subcommand MAXSTREAM. Similarly, the information provided by register 8 is equivalent to the information obtained from the pipeline subcommand STAGENUM.

Register 13 must contain a save area.

The pipeline Assembler macros give a return code in general register 15. The return codes are the only communications you receive from CMS Pipelines, so it is important for your stages to examine them.

Return Code on Exit

When your stage completes processing, it gives a return code to CMS Pipelines. By default, the return code is zero. To set a return code other than zero in a REXX user-written stage, specify the desired code on a REXX EXIT instruction (just as you do in a regular exec). If your stage detects an error, it can finish and report the error by using a nonzero return code on an EXIT instruction. When an Assembler macro completes processing, CMS Pipelines loads the return code into Register 15, then sets the condition code after testing the return code in Register 15.

When an Assembler user-written stage completes processing, CMS Pipelines expects the return code for the stage to be in Register 15.

CMS Pipelines gathers return codes from all stages, returning only one to CMS. It selects the return code to pass to CMS as follows:

- If there are negative return codes, CMS Pipelines returns the negative return code having the highest absolute value. Given return codes -2, -1, and 100, CMS Pipelines returns -2.
- If there are no negative return codes, CMS Pipelines returns the code having the highest value. Given return codes 100, 0, and 98, CMS Pipelines returns 100.

The CMS Pipelines Environment

CMS Pipelines checks the syntax of the pipeline you enter, and controls how each stage in the pipeline is run. The part of CMS Pipelines that performs syntax checking is called the *scanner*. The part of CMS Pipelines that controls stage processing is called the *dispatcher*. After the scanner parses the PIPE command, the dispatcher decides which stage to run, and when. The stages are not necessarily executed in the order that they are written in the PIPE command, and a stage does not necessarily run from start to finish once it does get control. Instead, the dispatcher may let part of one stage execute, then part of another, and so on.

How a Pipeline Runs

Let's see how a pipeline runs by examining the interactions of the dispatcher and the stages. Suppose you write a stage that reads records from its input stream, reverses the order of the characters in the records, and writes the changed records to its output stream. Since we haven't yet discussed pipeline subcommands in detail, we'll use pseudo code to show how it might be written:

```
do until we get a nonzero return code
  Read a record from the input stream (use READTO)
  Reverse the characters (use regular REXX functions)
  Write the record to the output stream (use OUTPUT)
end
```

Figure 126 shows REVIT REXX, an implementation of the above pseudo code.

```
/* REVIT REXX -- Copy input stream to output stream       */
                  /* We get control from CMS Pipelines    */
signal on error     /* Set up error handling              */
do forever
   'readto in'      /* Pipeline subcommand to read a record */
   'output' reverse(in)  /* Write reversed record         */
end
error:
if rc=12 then rc=0  /* RC=12 is a normal condition         */
exit rc             /* Return to CMS Pipelines             */
```

Figure 126. REVIT REXX: A Simple User-Written Stage

The following example shows how to use REVIT:

```
pipe < test file | console
Test 1
Test 2
Test 3
Ready;
pipe < test file | revit | console
1 tseT
2 tseT
3 tseT
Ready;
```

As shown above, REVIT processes one record at a time from the V-format file TEST FILE. Another way to implement REVIT is to use three loops. The first loop reads all the records in the input stream. The second loop reverses all the records. The third loop writes all the records to the pipeline. But this is not very efficient because you must hold all the records in virtual storage.

Remember that your stage may not run from start to finish. Processing a record at a time takes advantage of the dispatcher. Whenever possible, your stages should process a record at a time.

The rest of this section describes an execution of a PIPE command that contains REVIT. Although the description provides insight on how your stages run and how they interact with the dispatcher, you don't need to read the description to successfully write stages. If you do read the rest of this section, remember that it describes what *might* happen, not what *will* happen. The order in which the dispatcher processes stages is not predictable. Also, for brevity, we've simplified some aspects of how the < and > stages work.

With the above disclaimers in mind, let's use REVIT in this sample pipeline:

```
pipe < test data | revit | > reversed data a
```

To summarize what happens, we'll be using charts like the one in Figure 127 on page 80. To read these charts, look at each numbered step in order. The text on that line summarizes what is happening. The text is underneath the stage that is performing the action. For example, the first action that occurs is a READTO by REVIT. The second is a READTO by the > stage, and so on.

```
    < test data      | revit               | > reversed data a

1.                       READTO (must wait)
2.                           .                        READTO (must wait)
3.   Read a file record      .                            .
4.   OUTPUT (must wait)       .                            .
5.       .               Stage resumes                     .
6.       .               READTO finishes                   .
7.       .               Reverse characters                .
8.       .               OUTPUT (must wait)                .

   The dispatcher can run the < stage or the > stage next.
```
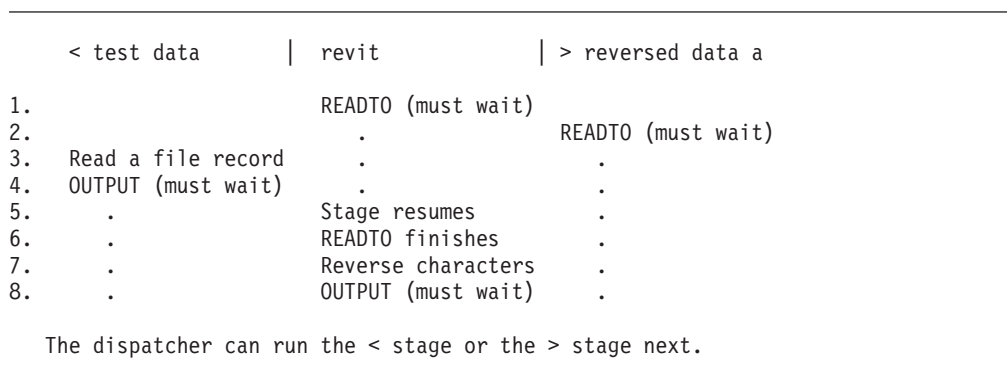
Figure 127. Pipeline Execution Example

When the PIPE command is executed, the dispatcher gets control. The dispatcher can let any of the three stages run. The choice it makes is unpredictable. For our example, assume that the dispatcher decides to let REVIT run first.

Referring to the pseudo code and to Figure 126 on page 79, we see that REVIT executes a READTO pipeline subcommand to read a record from its input stream. At that point, the dispatcher gets control again. (The dispatcher gets control whenever a pipeline subcommand is executed.)

Because REVIT is the first stage to run, there isn't a record immediately available. (The < stage hasn't read any file records yet.) So, the dispatcher can't satisfy the READTO request. Instead, it must run another stage. It can run either the < stage or the > stage.

Let's say that the dispatcher decides to run the > stage. The > stage reads records from its input stream and writes them to a file. The > stage executes a READTO pipeline subcommand to read a record from its input stream. (The built-in stages use the equivalent of pipeline subcommands. In this respect, they are not so different from user-written stages.)

When the > stage executes the READTO, the dispatcher gets control. Once again, no record is available. REVIT has not yet written any records to its output stream, so there is no record for the > stage to read. Now the dispatcher has two stages waiting for READTOs to finish: REVIT and the > stage. So the only remaining stage that can run is the < stage. The dispatcher gives it control.

The < stage reads a record from the file TEST DATA and writes the record to its output stream. To write the record, the < stage uses the OUTPUT pipeline subcommand. The dispatcher gets control once again.

Now the dispatcher has a record that it can give to REVIT. (The output stream of the < stage is connected to the input stream of REVIT.) So, the dispatcher decides to give REVIT control again. Remember that REVIT is still waiting for its first READTO to end. While it was waiting, two other stages have executed: the > stage (which is also waiting) and the < stage (which just ran). Now REVIT's wait is over. READTO processing puts the record in a REXX variable and REVIT resumes. REVIT executes an OUTPUT pipeline subcommand to write the record to its output stream. The characters are reversed by a REXX REVERSE function that is part of the OUTPUT expression. Again the dispatcher gets control.

Can you guess what happens next? The dispatcher has several alternatives:

- It can let the < stage run again. The < stage is waiting for its OUTPUT pipeline subcommand to end.
- It can let the > stage run. The > stage is waiting for its READTO to end. REVIT has made a record available to satisfy the READTO. So the > stage can be dispatched.

One alternative the dispatcher does not have is letting REVIT run again. REVIT cannot run until the > stage runs and reads the record REVIT has just written. Only then can the dispatcher let REVIT's OUTPUT pipeline subcommand finish. The dispatcher does not let an OUTPUT finish until some stage executes a READTO to read the record.

Because the dispatcher usually has several alternatives when it regains control, the order of execution is not predictable. And, as we said in the beginning of this section, you can't assume that a stage runs from start to finish when it does get control. The key point is that anytime a stage executes a pipeline subcommand, it relinquishes control and other stages might execute.

However, the dispatcher does not preempt stages. Once a stage gets control, the dispatcher regains control only when the stage executes a pipeline subcommand (or ends). So if your stage contains an endless loop, for example, CMS Pipelines does not stop it.

Finally, remember that the preceding section was a sample.

How a Pipeline Ends

We've seen, conceptually, how a pipeline runs, but how does it end? Let's revisit REVIT. Again, we'll be referring to this PIPE command:

```
pipe < test data | revit | > reversed data a
```

All records are read, reversed, and written without error. How might the pipeline end? A summary chart is in Figure 128.

```
       < test data        |    revit          |  > reversed data a
1.          .                    .                    .
2.     Read file record          .                    .
3.     End-of-File so            .                    .
4.     Exit 0                    .                    .
5.                          Stage resumes             .
6.                          READTO returns RC=12  .
7.                          So Exit 0                 .
8.                                              Stage resumes
9.                                              READTO returns RC=12
10.                                             So EXIT 0

       The PIPE command ends with a zero return code.
```

Figure 128. A Pipeline Ending with a Zero Return Code

Eventually the < stage is dispatched to read another record but no records remain in the file to be read. So, the < stage ends with a 0 return code. When a stage ends, the dispatcher regains control. Assume REVIT is dispatched next. REVIT executes a READTO.

REVIT's input stream is no longer connected—the stage to which it was connected no longer exists. Therefore, a record cannot be read from the input stream. CMS

Pipelines communicates this to the stage by giving it a return code of 12. Return code 12 on any pipeline subcommand indicates that a stream is no longer connected. In this case, it is the input stream that is not connected (because we received the return code from READTO).

Upon receiving a return code of 12, REVIT ends processing. Its input stream is no longer connected, so there is no point in continuing. To end its processing, REVIT executes a regular REXX EXIT instruction with a zero return code.

A zero return code is warranted because REVIT didn't detect any errors. A disconnected stream (RC=12 on any pipeline subcommand) is not an error condition—it is a normal pipeline occurrence. In general, a user-written stage should exit with a zero return code if it receives a return code of 12 from a pipeline subcommand.

Finally, the > stage is dispatched and executes a READTO. It, too, receives a return code of 12 because its input stream is no longer connected (REVIT no longer exists). The > stage ends with a 0 return code, and the PIPE command itself ends with a zero return code.

Let's take another example, using the same pipeline. Suppose that we are in the midst of processing. Several records have already been read, reversed, and written to disk. Everything is running well. Then the > stage is dispatched to write another record. It detects an error while writing to the file (perhaps the disk is full). The > stage realizes that it cannot proceed so it ends with a nonzero return code. A summary chart is in Figure 129.

```
      < test data      |  revit            |  > reversed data a
1.         .                   .                      .
2.         .                   .                 Stage resumes
3.         .                   .                 READTO finishes
4.         .                   .                 Write record to file
5.         .                   .                 RC<>0, so EXIT RC
6.         .                 Stage resumes
10         .                 OUTPUT runs and gives
11.        .                     RC=12, so Exit 0
12.    Stage resumes
13.    OUTPUT runs and gives
14.       RC=12, so Exit 0

       The PIPE command ends with a nonzero return code.
```

Figure 129. A Pipeline Ending with a Nonzero Return Code

When the > stage ends, the dispatcher regains control. It stores the return code for later use in reporting to CMS. Then it goes about the usual business of dispatching the remaining stages. It does *not* end the pipeline just because a stage gave a nonzero return code.

Suppose the dispatcher gives control to REVIT next. When REVIT executes an OUTPUT pipeline subcommand to write another record, the dispatcher regains control. REVIT's output stream is no longer connected—the stage to which it was connected no longer exists. Consequently, the dispatcher returns control to REVIT, but passes it a return code of 12.

Upon receiving a return code of 12, REVIT ends processing. It can no longer write to its output stream, so there is no point in continuing. To end its processing, REVIT executes a regular REXX EXIT instruction with a zero return code.

Now two stages have ended. The > stage gave a nonzero return code to the dispatcher, while the REVIT stage returned a zero. The dispatcher gives control to the only remaining stage.

The < stage reads a file record and then executes an OUTPUT pipeline subcommand. The dispatcher receives control. It knows that REVIT has ended, so the output stream of the < stage is not connected. It returns control to the < stage, passing a return code of 12 on the OUTPUT pipeline subcommand.

Upon receiving the return code of 12, the < stage also ends. Its output stream is disconnected, so it exits with a return code of 0.

Now the dispatcher knows all stages have ended. One of the stages reported a nonzero return code. The dispatcher returns to CMS, passing back the nonzero return code.

In summary, a PIPE command ends when all of its stages end. The stages themselves decide when to end. Because they are dispatched in any order, stages can end in any order. A stage might decide to end when it:
- Completes its function, or
- Detects an error, or
- Detects that one or more of its streams are disconnected, or
- Detects that there is no more data to read from a device (for device drivers only).

Once a stage ends, the streams that were connected to it become disconnected. This starts a chain reaction that emanates from the ended stage. Soon return codes of 12 spread throughout the pipeline as stages end, and eventually the PIPE command ends.

An Example Stage—HOLD REXX

Figure 130 on page 84 shows a simple stage. It reads records from its input stream and writes them to its output stream without modifying them. The name of the stage is HOLD. It is stored in a regular CMS file named HOLD REXX. The name of the file is also the name of the stage. REXX is the recommended file type, *not* EXEC.

Note that stage names starting with DMS or FPL are reserved for IBM* use. You should not give your stages names that start with DMS or FPL. Refer to the chapter discussing restrictions in the *z/VM: CMS Pipelines Reference* for a complete list of restricted stage names.

```
/* HOLD REXX -- Copy input stream to output stream        */
                   /* We get control from CMS Pipelines    */
signal on error      /* Set up error handling              */
do forever
   'readto record'  /* Pipeline subcommand to read a record */
   'output' record  /* Pipeline subcommand to write a record */
end
error:
if rc=12 then rc=0  /* RC=12 is a normal condition          */
exit rc             /* Return to CMS Pipelines              */
```

Figure 130. HOLD REXX: A Simple REXX User-Written Stage

Except for the READTO and OUTPUT commands, HOLD REXX is like any other exec. READTO and OUTPUT are pipeline subcommands. READTO reads one record from its input stream into a variable. The argument of READTO is the *name* of a variable. In our example, the record is assigned to the REXX variable *record*. Notice that the variable name is *within* the quotation marks (').

OUTPUT writes a record to the pipeline. Unlike READTO, the argument on OUTPUT is a string, not the name of the variable. In this example, the contents of *record* are written.

The REXX SIGNAL instruction sets up the error handling. It causes REXX to watch for commands that give nonzero return codes. If a command gives a nonzero return code, REXX branches to the ERROR label and continues processing.

HOLD REXX loops, reading and writing pipeline data, until a nonzero return code occurs. Then REXX branches to the ERROR label. We know that we'll eventually get a return code of 12 on READTO or OUTPUT. (See "How a Pipeline Ends" on page 81.)

The IF instruction following the error label implements the return code handling described in "How a Pipeline Ends" on page 81. The IF instruction checks for a return code of 12. Return code 12 is a normal condition, so the return code of 12 is changed to a 0. The EXIT instruction ends the filter and passes the return code back to CMS Pipelines.

Writing Stages in Assembler

This section describes specific rules and pipeline Assembler macros to use when writing your own stage in Assembler language.

Every stage that is written in Assembler must have storage defined every time the program is entered to allow the program to be re-entrant. This is set up in the *dummy control section* (DSECT). The stage must contain the PIPDESC macro and the PIPEPVR macro.

Setting up the DSECT

To allow the Assembler program that is your user-written stage to be re-entrant, you must define storage each time you enter the program. You may define a DSECT to map storage obtained for variables used in the program. Variables have to be defined within the storage obtained by the WORKAREA parameter on PIPDESC; they cannot be defined within the *control section* (CSECT) of the program. The first storage definition in the DSECT **must** be at least an 18 fullword save area. Any

variables to be used in the user-written stage should be defined after the save area. To save storage, define the data constants to be used by the user-written stage in the program's CSECT. If you choose to define data constants (DCs) in the work area, initialize them in the program's CSECT.

Using the PIPDESC Macro

The PIPDESC macro describes the Assembler program that is to be run as a user-written stage. It is required in all Assembler user-written stages. PIPDESC defines such characteristics as the name of the stage, the entry point of the program, and the size of the work area and a program identifier to be used in the CMS Pipelines error messages.

For example, the following is a section of code from an Assembler user-written stage named COPYTWO:

```
COPYTWO  PIPDESC  EP=MYPROG,WORKAREA=WRKARLEN,MODULEID=MYPROG
WORKAREA DSECT    0H
SAVEAREA DS       18F
WRKARLEN EQU      *-WORKAREA
```

This macro indicates that the entry point for this program is defined as MYPROG, to be matched with a MYPROG label in a control section (CSECT) of your program. The length of the work area is determined by WRKARLEN, and MYPROG is the program identifier that will be used in CMS Pipelines error messages issued. Note that the entry point (EP) name and the label on the PIPDESC macro must be different.

Using the PIPEPVR Macro

The PIPEPVR macro is also required in all Assembler user-written stages. PIPEPVR declares the address of the table of entry points to CMS Pipelines routines that are used by the Assembler macros. Before your user-written stage gets control from CMS Pipelines, the address of this table is loaded into Register 9. You can subsequently load that address into another available register.

An Example Assembler Stage—COPYCAT

Figure 131 on page 86 shows a simple Assembler stage. It reads records from its input stream and writes them to its output stream without modifying them. The name of the stage is COPYCAT. This name is defined as the label for the PIPDESC statement. The name of the file that contains the COPYCAT source code is PIPSKEL ASSEMBLE.

Note that stage names starting with DMS or FPL are reserved for IBM* use. You should not give your stages names that start with DMS or FPL. Refer to the chapter discussing restrictions in the *z/VM: CMS Pipelines Reference* for a complete list of restricted stage names.

```
PIPSKEL  START
PIPSKEL  AMODE 31
PIPSKEL  RMODE ANY
* This is a BASIC outline to begin your Pipelines Assembler stage
*   STANDARD ENTRY CODE
         REGEQU
         STM      R14,R12,12(R13)    Save caller's registers
         BALR     R12,0              Load base register
         USING    *,R12              Tell assembler what to use
         ST       R13,4(R1)          Save caller's save area pointer
         ST       R1,8(R13)          Save own save area pointer
         LR       R13,R1             Point at my savearea
         USING    WORKAREA,R13       Map the savearea and workarea
* Processing starts here
         PIPEPVR  (R9)               R9 points to table of addresses
         SR       R0,R0              Clear R0
         SR       R1,R1              Clear R1
LABEL1   PIPLOCAT ,                  Peek at a record
         BNZ      CHECK
LOOP     DS       0H
         PIPOUTP  ,                  Write out the record
         BNZ      CHECK
         PIPINPUT (,0)               Consume the record
         BNZ      CHECK
         B        LABEL1             If so, continue processing
CHECK    DS       0H
         C        R15,=F'12'            Is it end of file?
         BNZ      FINISH                If not leave the RC alone
         LA       R15,0                 Set RC to zero
FINISH   DS       0H
*   STANDARD EXIT CODE
         L        R13,SAVEAREA+4     Restore caller's savearea address
         L        R14,12(R13)        Restore original register contents
         LM       R0,R12,20(R13)     Preserving return code in R15
         BR       R14                Return to caller
**********************************************************************
*                    DATA DEFINITIONS
COPYCAT  PIPDESC  EP=PIPSKEL,WORKAREA=WRKARLEN,MODULEID=PIPSKEL,STREAMS=2
WORKAREA DSECT    0H
SAVEAREA DS       18F
WRKARLEN EQU      *-WORKAREA
         END
```

Figure 131. COPYCAT: A Simple Assembler User-Written Stage

Let's examine this program more closely. Except for the CMS Pipelines interfaces, this program is like any other re-entrant and relocatable Assembler program.

PIPEPVR assigns register 9 as the pointer to the table of entry points to CMS Pipelines routines that the Assembler macros will use. PIPEPVR must be coded before any other CMS Pipelines Assembler macros in the CSECT. The first invocation of PIPLOCAT inspects the first record available on COPYCAT's primary input stream. It does not remove that record from the stream. PIPOUTP makes the record available, without modification, to the stage connected to the primary output stream of the COPYCAT stage. PIPINPUT removes the record from the primary input stream. As long as end of file is not reached, the Branch (B) instruction causes control to be passed to the label LABEL1, where this process begins again with PIPLOCAT inspecting the next record available on COPYCAT's primary input stream. When PIPINPUT receives an end-of-file condition, CMS Pipelines loads a return code of 12 into register 15. Because register 15 contains a 12, a Branch on Not Zero (BNZ) instruction causes control to be passed to the label CHECK. This

program does not consider return code 12 to be an error, so 0 is loaded into register 15 as the return code from the COPYCAT stage.

The data definitions consist of the PIPDESC macro and the work area DSECT. The PIPDESC macro defines the entry point for this program as PIPSKEL, the length of the work area as WRKARLEN, and PIPSKEL as the program identifier that will be used in any CMS Pipelines error messages issued. Note that the entry point name PIPSKEL is different than the stage name COPYCAT. The first storage definition in the work area DSECT is the required 18 fullword save area.

Using Your REXX Stage

Use your stages the same way you use built-in stages. For example, to use HOLD REXX in a pipeline:

```
pipe < test file | console
Test 1
Test 2
Test 3
Ready;
pipe < test file | hold | console
Test 1
Test 2
Test 3
Ready;
```

If the name of your stage is the same as the name of a built-in stage, use the REXX stage to run yours. For example, suppose you write a stage named LOCATE that is not case sensitive. To use your LOCATE instead of the built-in LOCATE:

```
pipe < test data | rexx locate /Mixed!/ | console
```

Using Your Assembler Stage

Before you can use your Assembler stage in a pipeline, you must assemble your code using the FPLGPI MACLIB, FPLOM MACLIB, and any other MACLIB your installation requires. Then load the program using the CMS LOAD command. For example, to use your COPYCAT stage enter commands similar to the following:

```
VMFHLASM PIPSKEL FPLVM
LOAD PIPSKEL
```

The level of assembler that you are running must use High Level Assembler Release 1 (program number 5696-234) or higher.

Once the program is assembled and loaded, you can run the Assembler stage using the label on the PIPDESC assembler macro as an operand on the LDRTBLS stage. For example, to run the COPYCAT stage issue a PIPE command similar to the following:

```
pipe literal 1 2 3 | split | ldrtbls copycat | console
```

The resulting terminal output is:

```
1
2
3
Ready;
```

Use the LDRTBLS stage to run a compiled REXX- or Assembler-language user-written stage as a TEXT file in a test environment without disrupting a MODULE file of the same name running in a filter package in the production

environment. This saves you from having to rebuild a filter package of stages several times. (You can run MODULE files with the NUCEXT stage.) Refer to the discussion of filter packages in Chapter 12, "Filter Packages," on page 225, to learn how to build your user-written stage into a filter package. This provides others with the capability of running your Assembler user-written stage as part of a filter package.

Pipeline Subcommands

This section describes common pipeline subcommands. You've already seen READTO and OUTPUT.

READTO Subcommand

READTO reads one record from an input stream. The contents of the record are placed in a variable whose name you specify as the sole READTO operand. A record is read and discarded if you execute READTO without an operand.

The variable RC is set with the return code from READTO. Return code zero means that the function is performed as requested: the variable has a value. Return code 12 means the input stream is disconnected. A likely cause is that all records have already been read. When return code 12 is given, the variable is dropped so you do not inadvertently use obsolete data. Use the REXX SIGNAL ON NOVALUE instruction to detect dropped variables.

Important: The *name* of the variable is the argument to READTO. Be sure to put it inside the quoted string.

OUTPUT Subcommand

OUTPUT writes one record to an output stream. OUTPUT accepts a string as an argument. It writes the string to the output stream. The first blank following the word OUTPUT is not considered to be part of the data. Any other leading blanks are written to the output stream.

The variable RC is set with the return code from OUTPUT. The return code is zero if the line is read by the following stage. Return code 12 is set if the output stream is not connected. See "How a Pipeline Ends" on page 81 for more about the meaning of return code of 12.

PEEKTO Subcommand

Each time you read a record with READTO, a new record is stored in a REXX variable. Sometimes it is convenient to be able to peek at a record but be able to read it again later with READTO, or leave it in the input stream for a subroutine pipeline to read. (Subroutine pipelines are described later in this chapter.) This is exactly what PEEKTO does. Except for leaving the record in the input stream so it can be read again, PEEKTO behaves just like READTO.

PEEKTO without a variable name sets the return code to zero when there is a record available to read. Like READTO, return code 12 means the input stream is disconnected.

You can process the record peeked at, and then execute READTO to remove the record from the input stream. PEEKTO is useful in writing stages that do not delay the record. See "DELAY Stage" on page 155 for more information.

SHORT Subcommand

Often your stage processes its input records to end-of-data and then exits. Sometimes you may wish to copy the remaining pipeline records unmodified to the output stream. Although you can write a loop like the one in HOLD REXX, there is an easier way: the SHORT subcommand.

SHORT causes CMS Pipelines to reconnect streams so that they bypass your stage. Consider the following PIPE command, which uses stages named A, B, and C. Your stage is stage B.

```
pipe A | B | C
```

When your stage (B) runs the SHORT subcommand, CMS Pipelines connects the output stream of stage A directly to the input stream of stage C. The remaining records bypass your stage (stage B).

After you execute SHORT, the input and output streams are no longer available to your stage. Because your stage has not yet ended, however, the dispatcher will still give control to your stage (eventually). At that time you can, for instance, do clean-up processing, but you can no longer execute READTO or OUTPUT to process the stream.

If PEEKTO is the last subcommand issued before the SHORT subcommand, the record seen is included in the remainder of the input stream which is passed on.

HOLD REXX could be more efficiently written as shown in Figure 132.

```
/*          Tight version of HOLD REXX                         */
  'short'                         /* Copy input to output       */
  exit RC                         /* Return                     */
```

Figure 132. Modified HOLD REXX: Using SHORT Pipeline Subcommand

This version of HOLD REXX is just as redundant as the one we saw previously, but faster! In CMS Pipelines, a SHORT is the shortest and fastest path between two stages because the remaining records bypass the stage doing SHORT.

SHORT is also useful when processing headers. Suppose, for example, you are processing input that consists of a header and a body. Your filter needs to modify some lines in the header but not in the body and you want your output to contain both.

Using READTO and OUTPUT subcommands, read and process your header. When you detect the body section (or end of the header) you can execute SHORT. That takes care of copying the body to the output of your filter without modifying the data.

Figure 133 on page 90 shows an example stage that uses SHORT after processing a header. In this case, the stage, AUTHOR, changes the header on a SCRIPT file.

The following is an example of a SCRIPT file containing a header:

```
.********* Start of Header **************************************
.*
.* Security classification:  Company Secret
.*
.* Title:  CMS Pipelines User's Guide
```

```
.*
.* Author:  Joe Smith
.*
.* Filename:  MYBOOK
.*
.********* End of Header ****************************************
:h1.Pipeline Basics
:p.
CMS Pipelines lets you solve big problems by combining small programs.
It lets you do work that would otherwise require someone to
write a new program.
Often you get the result you need
with a single CMS command.
That command is PIPE.
:
:
```

The header consists of SCRIPT comments, which are records beginning with the string .*. One of the header records contains the keyword Author:. It is this record we must replace. The end of the header is indicated by a comment containing the string End of Header. For simplicity, AUTHOR REXX assumes that there is an author record.

```
/* AUTHOR REXX  -- Change the author in the header of a SCRIPT file */
signal on error

authrec='.* Author:  Tim A. Shenka'      /* Set new author record   */
do forever
   'readto record'                       /* Read a pipeline record  */
   uprec=translate(record)               /* Fold it to uppercase    */

   if pos('AUTHOR:',uprec)>0 & left(uprec,2)='.*' then do
      'output' authrec         /* Write the new author record      */
      leave                    /* Leave DO FOREVER                 */
   end
   else 'output' record        /* Otherwise, write record to output */
end

'short'                        /* Copy remaining records to output  */

error:
if rc=12 then rc=0
exit rc
```

Figure 133. AUTHOR REXX: Using SHORT Pipeline Subcommand to Process a Header

AUTHOR REXX reads records and copies them to its output stream until it finds the record containing the author. Then it replaces the author record with the new record and leaves the DO FOREVER loop. Here is an example of how to use AUTHOR in a PIPE command:

```
pipe < edition1 script | author | > edition2 script a
Ready;
```

STAGENUM Subcommand

Occasionally it might be useful for a stage to know its position in the pipeline, also called the *stage number*. The first stage has stage number 1, the second stage has stage number 2, and so on.

The stage number is made available to a stage as the return code of the STAGENUM subcommand. STAGENUM has no parameters; after it is executed, the variable RC contains the stage number.

Most stages are filters that do the same thing in all positions of the pipeline. This is true for all of the filters provided with CMS Pipelines. Some device drivers do different things when they are first in a pipeline. With STAGENUM, your stage can do different things depending on its position.

Figure 134 shows a stage that adds numbers. It looks at the first blank-delimited word (or token) on each record. If the word is a valid number, it adds the number to the REXX variable SUM. Otherwise, it ignores the record. When all input records have been read, ADD writes a record to the pipeline containing the sum.

ADD uses STAGENUM to detect whether it is being used as the first stage. If it is, ADD exits with a return code of 24. (ADD doesn't behave like the built-in filters.)

You can easily modify ADD so it doesn't care what stage it is. Simply remove the STAGENUM subcommand and the next line that checks the return code. In this case, using ADD as the first stage causes it to write a record containing a zero to the pipeline. (The first time READTO is executed it gives a return code of 12 because there is nothing connected to the input stream of ADD.)

```
/* ADD REXX -- Add all numbers appearing as first token on input   */
/*             records.  Ignore any records that do not have a      */
/*             valid number as the first token.                     */

'stagenum'                         /* What stage are we?            */
if rc=1 then exit 24               /* First stage? Exit with RC=24  */

sum=0                              /* Initialize SUM                */
do forever
  'readto i'                       /* Read a record from the pipeline */
    if rc¬=0 then do               /* Check the return code         */
      if rc=12 then 'output 'sum   /* No more input?  Write sum.    */
                                   /* We're already leaving; ignore RC */
      leave                        /* In any case, leave the loop.  */
  end
  val=word(i,1)                    /* Pull off the first token      */
  if datatype(val)='NUM' then      /* Is it a valid number?         */
    sum=sum+val                    /* Yes, so add it to SUM         */
end

exit 0
```

Figure 134. ADD REXX: Using STAGENUM Pipeline Subcommand

The following example shows two sample PIPE commands that show how to use ADD:

```
pipe literal 3 | literal invalid | literal 4 | add | console
7
Ready;
pipe add | console
Ready(00024);
```

Processing Arguments

You can code stages to process arguments that are specified when the stage is invoked. Use the REXX PARSE ARG instruction to get these arguments. Figure 135 shows an improved AUTHOR REXX. The original AUTHOR REXX is in Figure 133 on page 90. This version of AUTHOR REXX accepts the name of the author as an argument. The name is used to create the new author record. The new and changed statements are **highlighted**.

```
/* AUTHOR REXX  -- Change the author in the header of a SCRIPT file  */
signal on error
parse arg name                          /* Get the name            */
if name='' then name='Anonymous'        /* Handle missing argument */

authrec='.* Author:  '||name            /* Build new author record */
do forever
    'readto record'                     /* Read a pipeline record  */
   uprec=translate(record)              /* Fold it to uppercase    */

   if pos('AUTHOR:',uprec)>0 & left(uprec,2)='.*' then do
      'output' authrec        /* Write the new author record    */
       leave                  /* Leave DO FOREVER               */
   end
   else 'output' record       /* Otherwise, write record to output */
end

'short'                       /* Copy remaining records to output  */

error:                        /* Error routine and exit            */
if rc=12 then rc=0
exit rc
```

Figure 135. AUTHOR REXX: Processing Arguments

To change the author record in file EDITION1 SCRIPT and have the result written to EDITION2 SCRIPT, you would enter:
```
pipe < edition1 script | author Ward E. Guy | > edition2 script a
Ready;
```

Executing CP and CMS Commands

CMS Pipelines processes any commands sent to the default environment (the environment in effect when no REXX ADDRESS instructions are issued). It does not forward unresolved pipeline subcommands to CMS or CP. Instead it gives a return code of -7.

Look at the MINUS7 REXX stage in Figure 136. It sends a CMS TELL command to the default environment.

```
/* MINUS7 REXX -- Send a TELL command to the default environment */
'tell * Hello'
'short'
exit rc
```

Figure 136. MINUS7 REXX: Sending a Command to the Wrong Environment

Here is an example of the error produced by MINUS7 REXX:

```
pipe minus7 | console
    2 *-* 'tell * hello'
      +++ RC(-7) +++
Ready;
```

When you want to send a command to CP or CMS from a stage, use the REXX ADDRESS instruction. For example, suppose you want to access a minidisk:

```
address command 'ACCESS 5C5 B/A'      /* Access the minidisk */
```

Do not use the ADDRESS instruction without a command unless you know how to get the pipeline environment back. You may not be able to issue pipeline commands once you have changed the default *CMS* environment to the *COMMAND* environment.

One way to get the command environment back is by changing the environment from within a REXX subroutine (or function). When a subroutine returns, REXX restores the default command environment to the one that was in effect when the subroutine was called. So, it is safe to change the default command environment in a subroutine that does not issue pipeline subcommands and does not call subroutines that issue pipeline subcommands.

Another Example Stage—TITLE REXX

This section presents an example stage named TITLE REXX that adds a title before every 20 lines of input. The TITLE REXX filter can be used just like any other filter in a pipeline:

```
pipe < legumes script  | title Legumes: | console
Legumes:

Peas
Bush beans
Pole beans
Lima beans
Ready;
```

TITLE REXX, in Figure 137, reads the input given to it from the pipeline. It writes a title (given as an argument to it) followed by 20 lines, and then another title, and so on.

```
/* TITLE REXX -- Output a title every 20 lines                    */
parse arg title_line                /* Get input arguments        */

'readto input_line'                 /* Read first record into variable */
do line_count=0 while rc=0          /* Loop while there are records   */
   if line_count//20 = 0 then do    /* 20-line boundary?          */
      'output 'title_line           /* Yes- write the title line... */
      'output  '                    /*      write a blank line     */
   end
   'output 'input_line              /* Write the input record      */
   if rc¬=0 then leave              /* Check RC from OUTPUT        */
   'readto input_line'              /* Read the next pipeline record */
end
if rc=12 then rc=0                  /* Ignore 12, which means EOF  */
exit rc
```

Figure 137. TITLE REXX: A User-Written Stage Example

The WHILE clause on the DO instruction checks the return code after READTO. When all the records are read, READTO gives a return code of 12 and ends the loop.

Notice that the return code is checked after the last OUTPUT pipeline subcommand, but not after the first two. The reason is that once one OUTPUT fails, following OUTPUTs will also fail. So, we save some code by checking only the last OUTPUT.

TITLE REXX also shows how to write a blank record to the output stream. Look at the second OUTPUT pipeline subcommand. Two blanks follow the OUTPUT keyword before the single quotation mark. The first blank is not part of the string that OUTPUT writes, but the second blank is. Therefore, OUTPUT writes a record containing a single blank to the output stream.

Using CALLPIPE to Write Subroutine Pipelines

The CALLPIPE pipeline subcommand lets your stage run another pipeline to process the records in the input stream. For example, suppose you are writing a stage that is to make all records flowing through it 80 bytes in length. Here is one way to do it:

```
/* FIXED REXX -- Make all records 80 bytes in length        */
signal on error     /* Set up error handling                */
do forever
   'readto in'       /* Pipeline subcommand to read a record */
   'output' left(in,80) /* Pad or chop records as necessary  */
end
error:
if rc=12 then rc=0  /* RC=12 is a normal condition          */
exit rc             /* Return to CMS Pipelines              */
```

Figure 138. FIXED REXX: Using READTO and OUTPUT Pipeline Subcommands

Because CMS Pipelines has CHOP and PAD stages, however, there is a simpler way to do it. The CALLPIPE pipeline subcommand makes it possible, as shown in Figure 139.

```
/* FIXED REXX -- Make all records 80 bytes in length        */
'callpipe *: | chop 80 | pad 80 | *:'
exit rc
```

Figure 139. FIXED REXX: Using CALLPIPE Pipeline Subcommand

Rather than use READTO and OUTPUT to process each record, we use a CALLPIPE pipeline subcommand to process all of the records. The operand on CALLPIPE is called a *subroutine pipeline*.

The pairs of asterisks and colons (*:) are *connectors*. They must be in stages by themselves as shown. The connector at the beginning of the subroutine pipeline is known as the *input connector*. The connector at the end is known as the *output connector*. A connector cannot be used in the middle of the subroutine pipeline.

The input and output connectors connect the subroutine pipeline to the pipeline that called your stage. When both connectors are used, CALLPIPE, in effect, inserts a

new section of pipeline in the pipeline that called your stage. By making these connections, all records remaining in the stage's input stream flow through the subroutine and are written to the stage's output stream.

In the following PIPE command, for example, the output of the CMS stage is connected to the input of the FIXED stage. The output stream of FIXED is connected to the input stream of the > stage:

```
pipe cms listfile * * a | fixed | > adisk list a fixed
```

Figure 140 shows a map of the pipeline before FIXED begins to run.
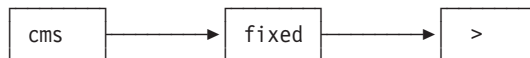
Figure 140. Map of Original Pipeline

When the CALLPIPE subcommand is executed, however, the output stream of the CMS stage is connected to the input stream of CHOP. And, the output stream of PAD is connected to the input stream of the > stage, as shown in Figure 141.
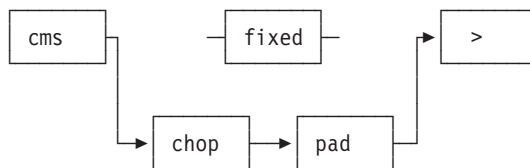
Figure 141. Map of Pipeline When CALLPIPE Is Running

CALLPIPE returns when all stages of the new pipeline have ended. The return code from CALLPIPE is the worst of the return codes from the stages in the subroutine pipeline (just like the PIPE command). When CALLPIPE finishes, CMS Pipelines restores the original connections so that the map of the pipeline is as shown in Figure 140. Then FIXED executes a REXX EXIT instruction and ends.

This leads us to an important point about CALLPIPE. When CALLPIPE is executed, the dispatcher gets control (just as it does for all pipeline subcommands). It adds the stages in the CALLPIPE command to the set of stages it is already dispatching and makes the requested connections. Records flow through the stages in the usual manner. But, and this is the important point, your stage itself does not resume execution until CALLPIPE ends.

When all the stages in the subroutine pipeline have ended CALLPIPE ends, and the original streams are reconnected, making your stage eligible for dispatching again. Your stage in the main pipeline can use both its input and output streams as though CALLPIPE's connections never existed.

Storing Sequences of Stages

CALLPIPE makes it easy to store stages or sequences of stages that you often use in PIPE commands. Suppose that you often use this SPECS stage, which adds sequence numbers to the beginning of a file:

```
specs recno 1 1-* next
```

You could save it in as a subroutine pipeline within a user-written stage:

```
/* ADDSEQ REXX -- Adds sequence numbers to the beginning of records */
'callpipe *: | specs recno 1 1-* next | *:'
exit 0
```

Then use ADDSEQ REXX whenever you want to add sequence numbers:

```
pipe < mybook script | addseq | > newbook script a
Ready;
```

More often, you'll be saving sequences of stage. Figure 142 shows a user-written stage named COUNTWDS REXX. It contains a subroutine pipeline that counts the number of words in its primary input stream, adds some text, and writes the result to its primary output stream. It packages two stages together for frequent use.

```
/* COUNTWDS REXX */
'callpipe',
  '*:',                /* connect to output of stage preceding caller */
  '| count words',   /* count number of words in input stream */
  '| specs /Number of words is/ 1 1-* nextword',  /* prefix record */
                      /* with "Number of words is" */,
  '| *:'             /* connect to input of stage following caller */
exit
```

Figure 142. COUNTWDS REXX: Subroutine Pipeline Example

The connectors used at the beginning and end of the pipeline subcommand allow the subroutine pipeline to take over the input and output streams for the COUNTWDS stage. Records flow into the subroutine pipeline through its input connector and later flow out through its output connector.

`count words` writes to the output stream the number of words delimited by a blank in the input stream. `specs /Number of words is/ 1 1-* nextword` inserts a prefix of `Number of words is` at the beginning of the record containing the count information. It then writes the resulting record to the output stream.

Here's an example of how to use the user-written stage to display the number of words in the file BLACK BOOK:

```
pipe < black book | cons
Anna Karinina
Scarlet OHara
Miss Piggy
pipe < black book | countwds | cons
Number of words is 6.
Ready;
```

Other Formats of Connectors

The connectors we have been using (*:) are actually abbreviations. You'll also see the input and output connectors written like this:

```
*.input:
*.in:
*.output:
*.out:
```

So, we can rewrite FIXED REXX as follows:

```
/* FIXED REXX -- Make all records 80 bytes in length        */
'callpipe *.input: | chop 80 | pad 80 | *.output:'
exit rc
```

You may see other connector formats in CALLPIPE subcommands and ADDPIPE subcommands that others have written. We'll also be using these formats later in Chapter 6, "Multistream Pipelines," on page 107, which also discusses the ADDPIPE subcommand.

Using Connectors with CALLPIPE

The CALLPIPE pipeline subcommands shown in the previous section used connectors to connect the specified pipeline to the input and output streams. In most cases, you'll want to use both connectors, but their use is not required. You can omit one connector from either end of the pipeline, or even from both ends.

Let's take the case in which you omit the connector from the end. Figure 143 shows an example:

```
/* LOGIT REXX                                                        */
'callpipe',
   '*:',                              /* Connect to input stream      */
   '| specs /'date() time()'/ 1',    /* Tack on date and time        */
         '1-* nextword',             /* Put input record             */
   '| >> logit file a'               /* Write records to file        */
exit rc
```

Figure 143. LOGIT REXX: Subroutine without an Output Connector

In LOGIT REXX, a time stamp is added to the records from the input stream. Then the records are appended to the file LOGIT FILE A. But, because there isn't a connector at the end of CALLPIPE, they do not flow out of the user-written stage. Here is an example of how you might use LOGIT:

```
pipe literal Returned Bill Smith's call. | logit
Ready;
pipe literal Worked on Project X today. | logit
Ready;
```

Figure 144 shows a map of the previous PIPE commands when CALLPIPE is running.
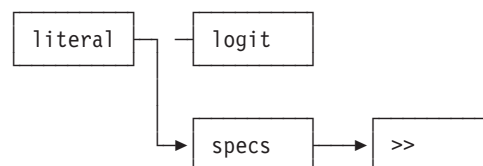
Figure 144. Map of Pipeline Using Only an Input Connector

What happens if you put a stage after LOGIT? Doing so is not an error. LOGIT just doesn't happen to write to its output stream, so no records flow into any following stage.

Figure 145 shows a CALLPIPE pipeline subcommand from which the input connection is omitted. The SEELOG stage is meant to be used as the first stage of a pipeline.

```
/* SEELOG REXX                                              */
'callpipe',
   '< logit file a',             /* Read the LOGIT FILE      */
   '| take last 2',              /* Take the last 2 records  */
   '| *:'                        /* Write to output stream   */
exit rc
```

Figure 145. SEELOG REXX: Subroutine without an Input Connector

Here is an example run:

`pipe seelog | console`
```
10 Dec 1991 12:38:26 Returned Bill Smith's call.
10 Dec 1991 12:38:30 Worked on Project X today.
Ready;
```

Figure 146 shows a map of the above PIPE command when CALLPIPE is running.
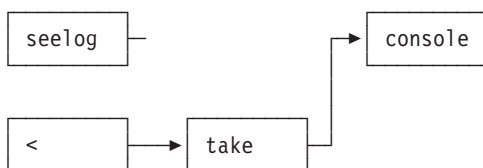
Figure 146. Map of Pipeline Using Only an Output Connector

What happens if you don't use SEELOG as the first stage? We've already seen a similar situation with LOGIT. Putting a stage before SEELOG is not an error. SEELOG doesn't read its input stream, so the records don't make it past SEELOG. For example:

`pipe literal Lost forever | seelog | console`
```
10 Dec 1991 12:38:26 Returned Bill Smith's call.
10 Dec 1991 12:38:30 Worked on Project X today.
Ready;
```

Figure 147 on page 99 shows a map of the above PIPE command when CALLPIPE is running. The < stage is not connected to the output stream from LITERAL. Consequently, the record from LITERAL is not processed.

```
    ┌─────────┐      ┌─────────┐                    ┌─────────┐
    │ literal │ ───► │ seelog  │─                   │ console │
    └─────────┘      └─────────┘                ┌──►└─────────┘
                                                │
                     ┌─────────┐    ┌─────────┐ │
                     │    <    │───►│  take   │─┘
                     └─────────┘    └─────────┘
```
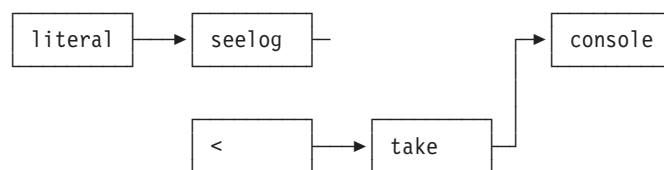
Figure 147. Map of Pipeline with Unconnected Streams

Finally, it is possible to omit all connectors from CALLPIPE. In this case, the subroutine pipeline is completely independent from the stage's input and output streams. See "IMMCMD Stage" on page 160 for an example use of a CALLPIPE pipeline subcommand without connectors.

Using CALLPIPE with Other Pipeline Subcommands

A user-written stage that contains a CALLPIPE pipeline subcommand can also contain other pipeline subcommands. There isn't anything special about CALLPIPE that prevents you from also using pipeline subcommands like READTO and OUTPUT. You just need to be aware of how CALLPIPE is connected to the input or output streams.

Let's look at SEELOG REXX again. Suppose we wanted to process records in the input stream. It is possible because the subroutine pipeline did not read the input stream—the records remain in the input stream. Figure 148 shows a new SEELOG REXX that reads any records in its input stream after the CALLPIPE is processed.

```
/* SEELOG REXX                                                  */
signal on error
'callpipe',
   '< logit file a',            /* Read the LOGIT FILE        */
   '| take last 2',             /* Take the last 2 records    */
   '| *:'                       /* Write to output stream     */
do forever
   'readto in'                  /* Read a record from the input */
   'output' date() time() in    /* Add time stamp and write it */
end

error:
if rc=12 then rc=0
exit rc
```

Figure 148. SEELOG REXX: Using CALLPIPE, READTO, and OUTPUT Pipeline Subcommands

We've added the usual error-handling instructions and a DO loop. The DO loop reads all records in the input stream, prefixes those records with a date and time, and writes them to its output stream. If there aren't any records in the input stream, CMS Pipelines will give a return code of 12 when READTO is executed.

After CALLPIPE ends, SEELOG REXX reads records from its input stream, processes them, and writes them to its output stream. Thus, you would expect to see these records after those from the file LOGIT FILE. That is, in fact, the result you get:

```
pipe literal Lost forever | seelog | console
10 Dec 1991 12:38:26 Returned Bill Smith's call.
10 Dec 1991 12:38:30 Worked on Project X today.
10 Dec 1991 13:19:47 Lost forever
Ready;
```

How would you put the records in input stream to SEELOG before those in the log file? You would have to move the DO loop before the CALLPIPE pipeline subcommand. You would also have to change the error handling, as shown in Figure 149.

```
/* SEELOG REXX                                                  */
do forever                        /* Process the input stream   */
   'readto in'                    /* Read a record              */
   if rc=12 then leave            /* Stream not connected? Leave loop */
   'output' date() time() in      /* Write output record        */
   if rc=12 then exit 0           /* Output not connected? Give up! */
   else if rc<>0 then exit rc     /* Pass other nonzero codes to PIPE */
end
'callpipe',
   '< logit file a',              /* Read the LOGIT FILE        */
   '| take last 2',               /* Take the last 2 records    */
   '| *:'                         /* Write to output stream     */
exit rc
```

Figure 149. SEELOG REXX: Another Variation

As you would expect, the input records are processed first:

```
pipe literal Lost forever | seelog | console
10 Dec 1991 13:32:36 Lost forever
10 Dec 1991 12:38:26 Returned Bill Smith's call.
10 Dec 1991 12:38:30 Worked on Project X today.
Ready;
```

Notice in Figure 149 that we do not end the stage when a return code of 12 is received on READTO. Instead, we just leave the DO loop. In this case, we have some other processing to do. The point is that a return code of 12 does not always mean you must end a stage. In many situations, a return code of 12 serves as a signal to your stage to alter its processing flow. The action to be taken is your choice.

In the above SEELOG variants, the CALLPIPE command did not have a connection to the stage's input stream. Suppose CALLPIPE does have a connection to the input stream. In that case, you could still execute READTO pipeline subcommands before the CALLPIPE. You might, for example, read and process header records by using READTOs and OUTPUTs, and then process the remaining records with a CALLPIPE.

In the following example, OUTPUT is used to write a header record, then CALLPIPE processes the remaining records:

```
/* UENG REXX -- Convert file to uppercase and change header */
signal on error

'output .* This file contains uppercase records'
'callpipe *: | xlate upper | *:'

error:
if rc=12 then rc=0
exit rc
```

UENG REXX writes one header record with an OUTPUT subcommand. Then it executes a CALLPIPE subcommand to read the remaining records from the input stream, translate them to uppercase, and write them to the output stream. Let's look at an example use of UENG:

```
pipe < lower script | console
Apples
Bananas
Cherries
Pears
Ready;
pipe < lower script | ueng | console
.* This file contains uppercase records
APPLES
BANANAS
CHERRIES
PEARS
Ready;
```

Another pipeline subcommand that is handy to use with CALLPIPE is PEEKTO. Suppose you're writing a stage that processes three different kinds of forms. You need to look at the first record to determine which form it is. But, you don't want to read the record using READTO because that would remove the record from the input stream. Once you have determined which form is being processed, you want to process all pipeline records, including the first, with an appropriate subroutine pipeline.

The PEEKTO pipeline subcommand is perfect for this kind of problem. Here is a fragment of a stage that does it:

```
/* Stage fragment to process multiple forms */
signal on error

'peekto in'
select
when pos("Invoice",in) then
   'callpipe *: | ... | *:'   /* CALLPIPE that processes invoices  */
when pos("Order",in) then
   'callpipe *: | ... | *:'   /* CALLPIPE that processes orders    */
when pos("Request for Bid",in) then
   'callpipe *: | ... | *:'   /* CALLPIPE that processes bids       */
otherwise
   exit 99    /* Tell CMS Pipelines you found an unidentified form */
end /* select */

error:
if rc=12 then rc=0
exit rc
```

To summarize what we've said about connections and streams, we end this section with COMBO REXX. (See Figure 150 on page 102.) COMBO processes three records using READTO/OUTPUT, and then three with CALLPIPE, and then three more with READTO/OUTPUT and so on until it gets a nonzero return code. The TAKE 3 stage in CALLPIPE forces it to end after processing three records. CMS Pipelines restores the connections and loops back to READTO/OUTPUT again.

```
/* COMBO REXX -- Take turns handling records              */
signal on error
do forever                       /* Process a group of six records */
   do i=1 to 3                   /* Process only three records */
      'readto record'            /* Read a record              */
      'output From OUTPUT:' record  /* Write it with a tag     */
   end
   'callpipe',
      '*:',
      '| take 3',                /* Process only three records */
      '| specs /From CALLPIPE:/ 1', /* Put tag on record       */
         '1-* nextword',         /* Put input record on it     */
      '|*:'
end

error:
if rc=12 then rc=0
exit rc
```

*Figure 150. COMBO REXX: Using READTO, OUTPUT, and CALLPIPE Pipeline
Subcommands*

Here is an example use of COMBO REXX:

```
pipe < number list | console
one
two
three
four
five
six
seven
eight
nine
ten
Ready;
pipe < number list | combo | console
From OUTPUT: one
From OUTPUT: two
From OUTPUT: three
From CALLPIPE: four
From CALLPIPE: five
From CALLPIPE: six
From OUTPUT: seven
From OUTPUT: eight
From OUTPUT: nine
From CALLPIPE: ten
Ready;
```

Additional CALLPIPE Examples

Figure 151 on page 103 shows FILEDATE REXX. FILEDATE generates a line
identifying the first occurrence of a file in the search order and writes the
information to its output stream.

```
/* FILEDATE REXX -- Generate the information on a file */
parse arg file
if words(file)=2 then file=file '*'

'callpipe',                                    /* Subroutine pipeline */
   'state' file,                               /* Look for it         */
   '| specs 1-22 1',                           /* Rearrange it        */
      '28.7 next',
      '37-44 next',
      '56-* next',
   '| specs 1-* 1.80 right',                   /* Right-adjust        */
   '| *:'                                      /* Write to output     */
exit rc
```

Figure 151. FILEDATE REXX: Using CALLPIPE Pipeline Subcommand

STATE gives return code 28 without issuing a message when the file does not exist. FILEDATE REXX receives this return code (or any worse one) and can act appropriately.

Here is an example run. FILEDATE is used to find the first occurrence of ALL XEDIT:

pipe filedate all xedit | console
```
                        ALL      XEDIT   S2 V    63        94  9/21/91 10:37:37
Ready;
```

Another example you may find useful is a TRACING filter. The argument to TRACING is a string of characters. TRACING prefixes the string to the contents of all records read, displays the records on the terminal, and passes the unmodified record on to the output. The function of the pipeline remains unchanged, but you have added a display of data as it passes through a specific stage. The data is prefixed with a message. Figure 152 shows the TRACING subroutine.

```
/*  TRACING REXX -- show data in middle of the pipeline   */
/*  Add the tag passed to TRACING, display each           */
/*  record, and delete the tag before passing the output. */
 parse arg id                /* get argument in mixed case  */
 if id = '' then do
    'stagenum'               /* get the stage number        */
    id = 'Stage' rc          /* use it as tag               */
 end
 id = id':'                  /* append a colon              */

'callpipe',
   '*:',                     /* Read from input             */
   '| change //'id'/',       /* Insert id first             */
   '| console',              /* Type                        */
   '| specs' length(id)+1'-* 1',          /* Remove id */
   '| *:'                    /* Pass on                     */
exit rc
```

Figure 152. TRACING REXX: Using CALLPIPE Pipeline Subcommand

In the CHANGE stage, a null string is specified as the string to be changed. When a null is specified, the string to be substituted (the contents of id, in this case) is inserted at the beginning of each record passing through the stage. The modified records are displayed, and then the SPECS stage removes the identifier.

The following example shows how to run TRACING. (The first PIPE command shows the contents of the TEMP DATA file.)

```
pipe < temp data | console
This is the first line.
This is the second line.
Ready;
pipe < temp data | tracing first | xlate upper | tracing second | console
first :This is the first line.
second :THIS IS THE FIRST LINE.
THIS IS THE FIRST LINE.
first :This is the second line.
second :THIS IS THE SECOND LINE.
THIS IS THE SECOND LINE.
Ready;
```

Testing Stages

When you develop a new stage it is easiest to test it by itself. LITERAL and CONSOLE stages are especially useful for testing, as are disk files. You can use a device driver for whatever device is convenient.

Two simple ways to test a filter are:

```
pipe literal aa004zz  q  | myfilter | console
```

```
pipe console | myfilter | console
```

Enter the input you want, then look at the output to see if it is correct.

When your input becomes too much to type over again, create a file with test cases. One way to keep test cases organized is by using stylized file names. For example, you might use the name of the stage for the names of files containing the test cases. This lets you use the file type to indicate which test case it is (for example, MYFILTER TEST1, MYFILTER TEST2).

To test your filter against the data in MYFILTER TEST1, displaying the result on the terminal:

```
pipe < myfilter test1 | myfilter | console
```

If the output is lengthy, consider using result files with stylized file names:

```
pipe < myfilter test1 | myfilter | > myfilter result1 a
```

From a FILELIST display, the previous two commands can be simplified a little to save on typing. Enter, for example, one of these commands on the FILELIST display line for MYFILTER TEST1:

```
pipe < / | /n | console
```

```
pipe < / | /n | > /n result1 a
```

When you are developing a set of filters, test them from left to right and store the output of the first filter on disk so that you can use it over and over again to test the second filter, and so on.

When entering PIPE commands on a FILELIST display, do not use a slash (/) to delimit strings in filters (such as LOCATE). The slash (/) is resolved to the file name, file type, and file mode of the file listed, so you cannot use that for a delimiter. We often use a comma (,) to delimit the argument string to LOCATE. For example:

```
pipe < / | locate ,pattern, | console
```

Tracing Stages

You can use the REXX TRACE instruction when debugging stages. For example, the following statement starts an interactive trace:

```
Trace ?R      /* Starts interactive REXX debug */
```

Each line of your stage is displayed as it is executed. After each line is displayed, execution pauses until you press the ENTER key or until you end the trace by entering TRACE OFF. Use the SAY instruction to display the contents of variables.

In lengthy stages, frame the code that needs to be traced:

```
Trace Results       /* Starts REXX trace                 */
    ....            /* code that is traced on the terminal */
Trace Negative      /* Stops REXX trace                  */
```

All other trace options can also be used. See the *z/VM: REXX/VM Reference* for more about TRACE.

If you use several user-written stages in a pipeline, you would usually trace only one at a time. When you trace more than one, the displayed traces will be interleaved. Remember that CMS Pipelines uses a dispatcher and that execution of the stages is interleaved. Consequently, so are the trace displays.

You might want to trace two stages just to see the dispatcher in action. Remember that the dispatcher gets control whenever a pipeline subcommand is executed. At that time the dispatcher can choose to run any other stage. Note that you do not see the trace record for a pipeline subcommand until it ends—another traced stage might be dispatched before the pipeline subcommand ends.

Improving Performance

You can improve performance of stages in several ways:
- Invoke the CMS EXECLOAD command for the stages when they are first called. See the *z/VM: CMS Commands and Utilities Reference* for more about the EXECLOAD command.
- Put the programs into a shared segment.
- Compile the programs with the REXX compiler.

An application using many stages can be packaged into a *filter package* and installed as a nucleus extension. This may improve performance the same way that using EXECLOAD does. You also have the convenience of storing a single file on a common disk for users of the application. See Chapter 12, "Filter Packages," on page 225 for more information on creating and using filter packages.

Chapter 6. Multistream Pipelines

Many CMS Pipelines stages can use multiple input streams and multiple output streams. These stages are like houses that have several front doors and several back doors. So far we have been admitting records through only one front door, the primary input stream, and have been whisking them out the corresponding back door, the primary output stream (Figure 153).

```
                           ┌──────────┐
                           │  Stage   │
                           │          │
───Primary Input Stream───▶│          │────Primary Output Stream───▶
                           │          │
                           │          │
                           └──────────┘
```
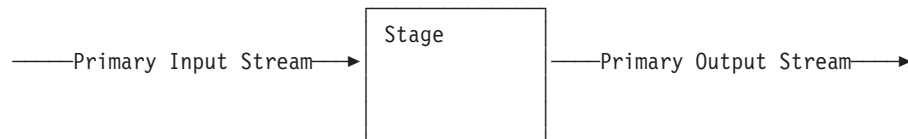
Figure 153. Stage with One Input and One Output Stream

Now we'll let records pass through the other front and back doors, which we refer to as the secondary input and output streams (Figure 154).

```
                          ┌─────────┐
───Primary Input Stream──▶│  Stage  │───Primary Output Stream──▶
                          │         │
───Secondary Input Stream▶│         │───Secondary Output Stream▶
                          └─────────┘
```
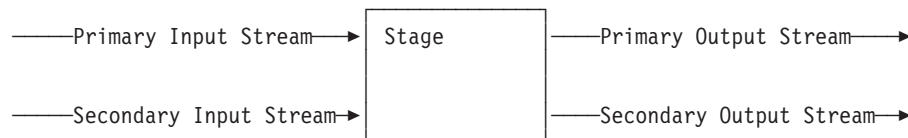
Figure 154. Stage with Two Input and Output Streams

Some stages use even more than two input and output streams. As we'll see, the use of multiple streams greatly extends the range of problems you can solve with CMS Pipelines.

How Stages Use Multiple Streams

Different stages use multiple input and output streams in different ways. Some, like LOOKUP, use a secondary input and a secondary and tertiary output stream. Some, like LOCATE, use a secondary output, but not a secondary input. Others, like FANOUT, can use *more* than two output streams.

The LOCATE stage, for example, writes records that are selected to its primary output stream. It writes records that are *not* selected to its secondary output stream (see Figure 155 on page 108).

107

```
Primary Input Stream                    Primary Output Stream

┌─────────────┐       ┌─────────────┐       ┌─────────────┐
│ BOB SMITH   │──────▶│ LOCATE /BOB/│──────▶│ BOB SMITH   │
│ SUE JONES   │       │ Stage       │       └─────────────┘
└─────────────┘       │             │
                      │             │       ┌─────────────┐
                      │             │──────▶│ SUE JONES   │
                      └─────────────┘       └─────────────┘

                                            Secondary Output Stream
```
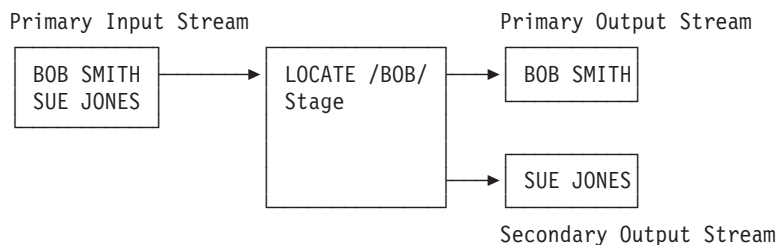
Figure 155. LOCATE with a Secondary Stream

The FANOUT stage, which is described in this chapter, writes each record it reads from its primary input stream to all of its connected output streams (Figure 156).

```
Primary Input Stream                    Primary Output Stream

┌─────────────┐       ┌─────────────┐       ┌─────────────┐
│ BOB SMITH   │──────▶│ FANOUT      │──────▶│ BOB SMITH   │
│ SUE JONES   │       │ Stage       │       │ SUE JONES   │
└─────────────┘       │             │       └─────────────┘
                      │             │
                      │             │       ┌─────────────┐
                      │             │──────▶│ BOB SMITH   │
                      └─────────────┘       │ SUE JONES   │
                                            └─────────────┘

                                            Secondary Output Stream
```
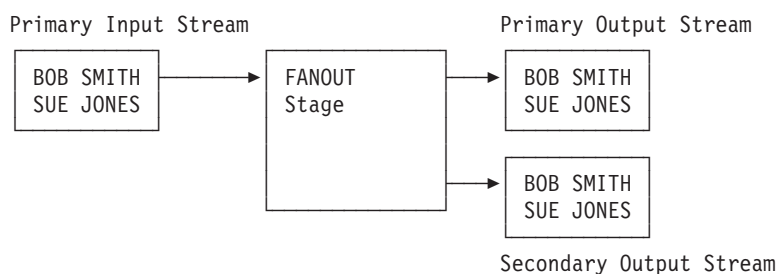
Figure 156. FANOUT with Multiple Output Streams

The records flowing from these secondary outputs can be processed by other stages (Figure 157).

```
              ┌─────────────┐     ┌─────────────┐     ┌─────────────┐
  ───────────▶│ LOCATE /BOB/│────▶│ COUNT WORDS │────▶│ > BOB DATA A│
              │ Stage       │     │ Stage       │     │ Stage       │
              └─────────────┘     └─────────────┘     └─────────────┘
                     │
                     │            ┌─────────────┐
                     └───────────▶│ > NOTBOB    │
                                  │   DATA A    │
                                  │ Stage       │
                                  └─────────────┘
```
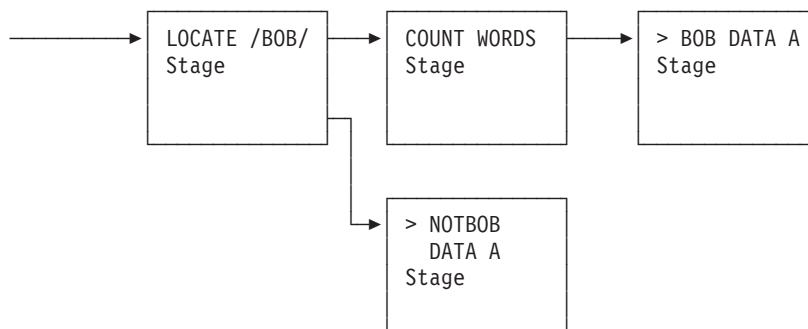
Figure 157. Processing Secondary Outputs

Like LOCATE, most filters write rejected data to their secondary output streams. DROP, for example, writes the discarded records to its secondary output stream. CHOP writes the discarded portion of each record to its secondary output stream.

Some stages, for example SPECS, do not have a secondary output stream. When in doubt, refer to the stage descriptions in *z/VM: CMS Pipelines Reference*.

Writing Multiple Pipelines

When more than one input or output stream is used, we no longer have a map in which all stages are arranged in a straight line. Instead, we have multiple pipelines. To use more than one input stream or more than one output stream, or a combination, we need to write multiple pipelines in a single PIPE command. To show how to write multiple pipelines, we'll use a simple example.

```
┌─────────────────┐      ┌──────────────────┐
│ < profile exec  │ ───► │ > profile save a │
└─────────────────┘      └──────────────────┘

┌─────────────────┐      ┌──────────────────┐
│ < all notebook  │ ───► │ > all save a     │
└─────────────────┘      └──────────────────┘
```
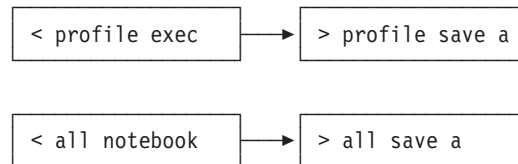
Figure 158. Maps of Two Independent Pipelines

Figure 158 shows two pipelines. Although the pipelines aren't connected in any way, you can still put both of them in a single PIPE command. To do so, use the ENDCHAR option on the PIPE command to define an *end character*. (See "Specifying PIPE Options" on page 8 for information about specifying PIPE command options.) Then use that end character in the PIPE command to indicate where each pipeline ends.

> **End Characters**
>
> You can use any character for the end character if it is not defined as a stage separator, escape character, or one of the characters `*.:()` which have special meaning.

Figure 159 shows how you would write the above pipelines in a single PIPE command. In the example, a question mark is defined and used as the end character. Notice that an end character is not used after the last pipeline. Specifying one would cause an error.

```
                  ┌──────── End character ────────┐
                  │                               ▼

pipe (endchar ?) < profile exec | > profile save a ? < all notebook | > all save a
                  └──────────────┬──────────────┘   └─────────────┬─────────────┘
                            First pipeline                   Second pipeline
```
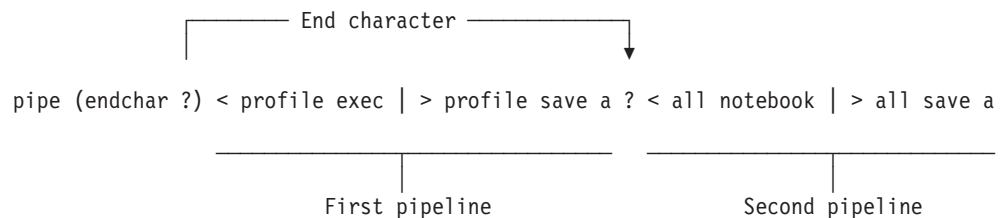
Figure 159. Two Pipelines Separated by an End Character

The records flowing through the two pipelines are completely independent of each other. In fact, you could get the same results by using two PIPE commands.

The above example shows the mechanics of writing multiple pipelines in a single PIPE command. But, we haven't said anything about connecting streams together. In each of the above pipelines, the primary output stream of the < stage command

is connected to the primary input stream of the > stage. We know this because the stages are adjacent, connected with the stage separator (|). To use the secondary input and output streams of stages, we need some way to connect stages that are not adjacent.

Connecting Streams

This section describes the different ways that you can connect streams. When stages are adjacent to each other in a pipeline, the primary output stream of a stage is connected to the primary input stream of the stage to its right. CMS Pipelines makes these connections automatically for us.

To connect the streams of stages that do not happen to be adjacent, however, we need to use *labels* on the stages. A label can have from one to eight characters. It must be followed by a colon (for example, a:). Keep in mind that if you have two of the same stages (for instance two LOCATE stages) in one pipeline, each LOCATE stage must have a unique label.

To connect to a stage's multiple streams, first put a label in front of the stage whose secondary input or output stream you wish to use. This *defines* (or declares) the label. Then put a matching label elsewhere in the PIPE command in a stage by itself (see Figure 160). This is called a *label reference*. Labels must be defined in the PIPE command before any references to them. (Otherwise, an error occurs.)

The label definition on a stage allows the possibility of multiple streams to be connected through intersecting pipelines, and each label reference in a subsequent pipeline *defines* a new input and output stream for the stage. Streams are defined in input/output pairs, even if one of the pair is not used. (Unlike secondary streams, the primary input and output streams for a stage are defined by specifying the stage in a pipeline.)

```
/* */                          Label Definition
                       ┌─────────
'pipe (endchar ?)', ▼
    '< test data │ a: locate /BOB/ │ > bob data a', /* First pipeline  */
    '?',                                    /* End character   */
    'a: │ > notbob data a'                  /* Second pipeline */

        ┌─────
        └───── Label Reference
```

Figure 160. Defining and Referencing Labels

The location of the matching label determines what connections are made between the stages. The matching label can be in one of three locations:

1. At the beginning of a pipeline

   In this case, CMS Pipelines makes connections to the secondary outputs of the stage defining the label.

2. At the end of a pipeline

   In this case, CMS Pipelines makes connections to the secondary inputs of the stage defining the label.

3. In the middle of a pipeline.

   In this case, CMS Pipelines makes connections to the secondary inputs *and* secondary outputs of the stage defining the label.

In Figure 160 on page 110, for example, the label reference is at the beginning of the second pipeline. So, the secondary output stream of LOCATE is connected to the primary input stream of the `> notbob data a` stage. All the records that LOCATE rejects are written to the file NOTBOB DATA A.

The next three sections describe the above cases.

Connecting to a Secondary Output Stream

Often, when you want to use the secondary output of a stage, you are, in effect, trying to write a PIPE command with a map like the one in Figure 161.

Figure 161. Generic Map for Connecting to Secondary Outputs

The map shows that the secondary output of STAGE-B is connected to the primary input of STAGE-D. How do you write a PIPE command to implement the map? First, write the two pipelines in a single PIPE command:

```
/* Two pipelines, but no connections yet */
'pipe (endchar ?)',
   'stage-a | stage-b | stage-c',   /* First pipeline  */
   '?',                             /* End character   */
   'stage-d'                        /* Second pipeline */
```

Then define a label (a) by putting it in front of `stage-b`:

```
/* Two pipelines with a label defined, but no connections yet */
'pipe (endchar ?)',
   'stage-a | a: stage-b | stage-c', /* First pipeline  */
   '?',                              /* End character   */
   'stage-d'                         /* Second pipeline */
```

Finally, connect the secondary output of `stage-b` to the primary input of `stage-d` by putting the matching label at the beginning of the second pipeline. Notice that the matching label is in a stage by itself:

```
/* Two pipelines with a label defined and with connections */
'pipe (endchar ?)',
   'stage-a | a: stage-b | stage-c', /* First pipeline  */
   '?',                              /* End character   */
   'a: | stage-d'                    /* Second pipeline */
```

The label reference defines both a secondary input and output stream for `stage-b`. Because of the position of the label reference, however, the secondary input stream is not connected. So, in this example, the secondary input stream is defined but not connected, while the secondary output stream is both defined and connected.

Whatever `stage-b` writes to its secondary output stream will now flow to the primary input stream of `stage-d`.

We used fictitious stages to show how to make connections. To create a real PIPE command, substitute your own stages. For example, look at the following PIPE command (a map is shown in Figure 162):

```
/* Matching label at beginning of pipeline */
'pipe (endchar ?)',
  '< vote data | a: locate /YES/ | > yes data a',  /* First pipeline  */
  '?',                                              /* End character   */
  'a: | > no data a'                                /* Second pipeline */
```

The < stage reads the file VOTE DATA. Then the LOCATE stage finds all records that contain the string YES. It writes these records to its primary output stream. The > stage writes the records to the file YES DATA A.

LOCATE writes the records that do not contain YES to its secondary output stream. The label a connects LOCATE's secondary output stream to the primary input of the > no data a stage. That stage writes the records to the file NO DATA A.
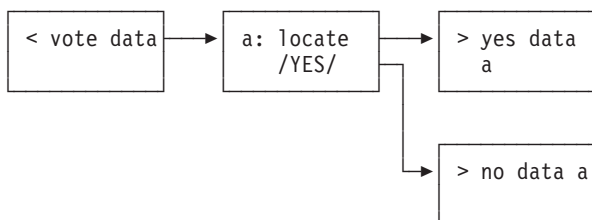
Figure 162. Map Showing Secondary Output

So far we've been showing multistream pipelines with one pipeline per exec line. We did this because that format conveniently matches the layouts of the maps. In an actual exec, the above PIPE command could be written on several lines, with comments:

```
/* Putting it together                            */
'pipe (endchar ?)',
  '< vote data',        /* Read the VOTE DATA file      */
  '| a: locate /YES/',  /* Select records containing YES */
  '| > yes data a',     /* Write them to YES DATA A      */
  '?',
  'a:',                 /* Process LOCATE rejects        */
  '| > no data a'       /* Write them to NO DATA A       */
```

Connecting to a Secondary Input Stream

When you want to supply records to the secondary input of a stage, you are, in effect, trying to write a PIPE command with a map like the one in Figure 163 on page 113.
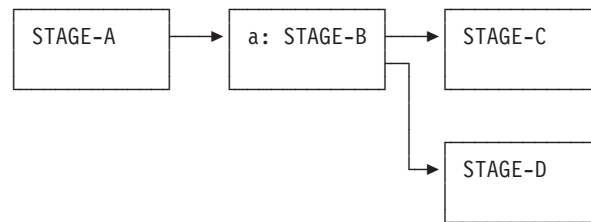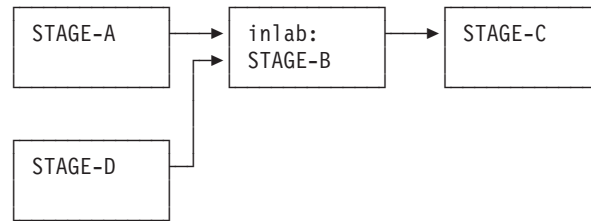
Figure 163. Generic Map for Connecting to Secondary Inputs

The map shows that the primary output stream of STAGE-D is connected to the secondary input of STAGE-B. The following example shows how to make the connection:

```
/* Connecting to a secondary input */
'pipe (endchar ?)',
   'stage-a | inlab: stage-b | stage-c',
   '?',
   'stage-d | inlab:'
```

The label `inlab` is defined on STAGE-B. Because the matching label is used at the end of the second pipeline, the primary output stream of STAGE-D is connected to the secondary input stream of STAGE-B. Therefore, any record that STAGE-D writes will flow into the secondary input of STAGE-B.

The label reference defines a secondary input and a secondary output stream for STAGE-B. In this example, the secondary input stream is both defined and connected, while the secondary output stream is defined but not connected.

Later in this chapter we discuss filters that combine streams, such as FANIN and FANINANY. When using filters like these, you'll find yourself writing pipelines with matching labels at the end, just as we did here.

Connecting to Both the Secondary Input and the Secondary Output

There may be times when you want to use both the secondary input and the secondary output of a stage. Figure 164 shows a map of a stage that has connections to both its secondary input stream and its secondary output stream.
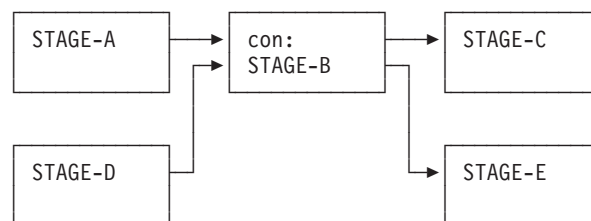
Figure 164. Generic Map for Connecting to Secondary Inputs and Secondary Outputs

The following example shows how to make the connections:

```
/* */
'pipe (endchar ?)',
   'stage-a | con: stage-b | stage-c',
   '?',
   'stage-d | con: | stage-e'
```

The records from STAGE-D flow into the secondary input of STAGE-B, while the records from the secondary output of STAGE-B flow into STAGE-E. Records do not flow from STAGE-D to STAGE-E through the stage containing the label `con`.

The label reference defines a secondary input and a secondary output stream for STAGE-B. In this case, both of the new streams are connected.

Later in this chapter we describe the LOOKUP stage. We'll see that in some uses of LOOKUP, you'll be using a map similar to the one in Figure 164 on page 113. For those uses, you'll need to write sections of pipelines like those above.

Using Several Secondary Streams

The previous examples used the secondary streams of a single stage. To solve complex problems, however, you will often need to use the secondary streams of several stages.

To use secondary streams of several stages, just define different labels. For example, suppose you want to write all records containing the string `BOB` to one file. Of the remaining records, you want to write those containing `SUE` to another file, and all other records to a third file. Figure 165 shows a map of what you need to do.

Figure 165. Using Several Secondary Streams

Figure 166 on page 115 shows the PIPE command to do it.

```
/* LOCATE3X */
'pipe (endchar ?)',
   '< test data',
   '| a: locate /BOB/',
   '| > bob data a',
   '?',
   'a:',
   '| b: locate /SUE/',
   '| > sue data a',
   '?',
   'b:',
   '| > other data a'
```

Figure 166. Three Pipelines in One PIPE Command

The first pipeline writes all records having the string BOB to the file BOB DATA A. The records rejected by locate /BOB/ flow into locate /SUE/ in the second pipeline. Records having the string SUE are written to the file SUE DATA A.
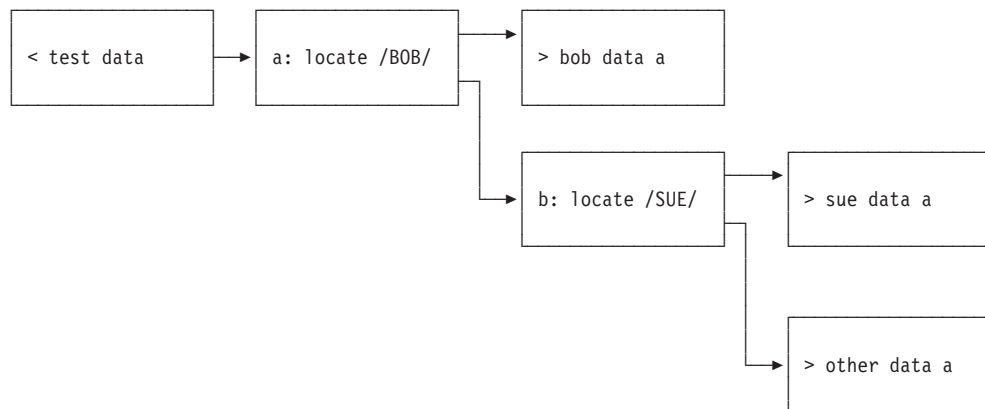
The LOCATE stage in the second pipeline (locate /SUE/) defines a new label (b). The secondary output stream of this LOCATE consists of all records that have neither BOB nor SUE, which is precisely what we want in the last output file. The third pipeline writes the records to OTHER DATA A. The label reference in the third pipeline connects the secondary output stream of locate /SUE/ to the primary input stream of > other data a.

Stages for Multistream Pipelines

This section introduces several stages that are frequently used in multistream pipelines. Those stages are:
- FANOUT
- FANINANY
- FANIN
- OVERLAY
- MERGE
- LOOKUP.

In addition, this section revisits the SPECS and COUNT stages.

FANOUT Stage

By using labels on filters we can process data that the filter would otherwise discard. But, what if we just want to *copy* the records in the pipeline to more than one output stream? Use the FANOUT stage.

FANOUT copies each record that it reads from its input stream to all of its output streams. To use FANOUT, put a label in front of it and refer to that label elsewhere in the pipeline, as you do with other filters.

For example, suppose you are compiling demographic data. From a master file you want to extract two sets of records and place them in separate files. In one file, you want the names of all people who were born in 1956. In another file, you want to list all males. All information concerning a person is on one master file record.

Here are some example records:

```
Name              Sex       Date of Birth  State
Smith, Robert M.  MALE      12 06 1956     NY
Jones, Morgan E.  FEMALE    05 05 1959     PA
Public, Waldo Q.  MALE      11 13 1960     CA
```

It is not possible to use the secondary stream of LOCATE to solve the problem. Suppose you use LOCATE to select all records containing 1956 and write them to a file name 1956 DATA A. Then you try to use the secondary stream of LOCATE /1956/ to find all males. This will not yield correct results because the secondary stream would not contain any records from 1956. Your output file would be missing all males born in 1956.

Because you want all records to be processed by both LOCATEs, use FANOUT to copy the stream.
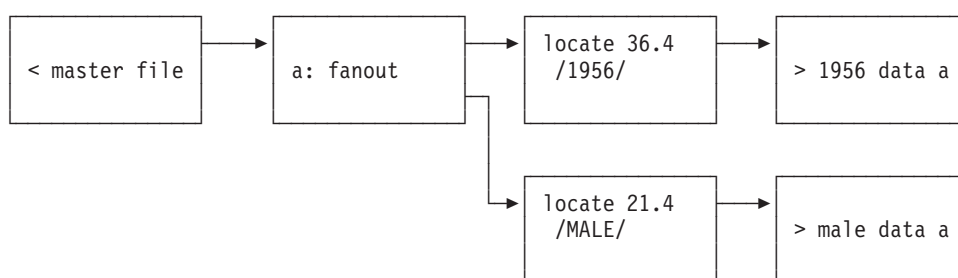
Figure 167. Map of FANOUT Example

Figure 168 shows the PIPE command.

```
/* FANOUT example                                    */
'pipe (endchar ?)',
   '< master file',        /* Read master file               */
   '| a: fanout',          /* Copy records to all output streams */
   '| locate 36.4 /1956/', /* Locate everyone born in 1956       */
   '| > 1956 data a',      /* Write records to file          */
   '?',
   'a:',
   '| locate 21.4 /MALE/', /* Locate all males               */
   '| > male data a'       /* Write records to file          */
```

Figure 168. FANOUT Example

What if you wanted to create a third file in which all of the residents of New York state were listed? You need to define yet another output stream (referred to as a *tertiary stream*). FANOUT supports more than two streams. To define a third output stream, refer to the same label again. Figure 169 on page 117 shows a map of the pipeline.
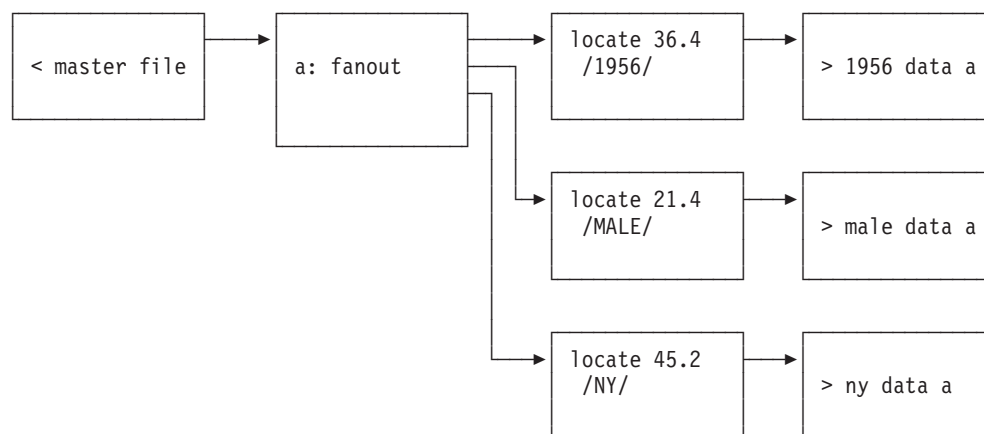
Figure 169. Map of FANOUT Example Using a Tertiary Stream

Figure 170 shows the PIPE command. A third pipeline is added. The first stage in the pipeline is a label that refers to the label defined on FANOUT, so an output stream of FANOUT is connected to the input stream of the stage `locate 45.2 /NY/`.

```
/* FANOUT example */
'pipe (endchar ?)',
   '< master file',        /* Read master file                */
   '| a: fanout',          /* Copy records to all output streams */
   '| locate 36.4 /1956/', /* Locate everyone born in 1956    */
   '| > 1956 data a',      /* Write records to file           */
   '?',
   'a:',
   '| locate 21.4 /MALE/', /* Locate all males                */
   '| > male data a',      /* Write records to file           */
   '?',
   'a:',
   '| locate 45.2 /NY/',   /* Locate all NY residents         */
   '| > ny data a'         /* Write records to file           */
```

Figure 170. FANOUT Example Using a Tertiary Stream

FANOUT is not the only stage that supports more than two streams. To determine whether a stage supports more than two streams, refer to the stage description in the *z/VM: CMS Pipelines Reference*. To use more than two streams on stages that support them, use the label repeatedly, as we did in the example.

FANINANY Stage

FANINANY reads a record from any input stream that has one and writes the record to its output stream. It is useful when you want to combine the records of several pipelines.

For example, Figure 171 on page 118 shows how to use FANINANY with LOCATE to do an OR function with CMS Pipelines. The pipeline creates a file listing all EXEC and SCRIPT files on file mode A. Any record containing the string `EXEC` or `SCRIPT` in the appropriate columns is selected. FANINANY reads the selected records from both input streams and writes them to its output stream.

Figure 171. Map of FANINANY Example

Figure 172 shows the PIPE command to do it.

```
/* LFD EXEC -- a FANINANY Example                              */
'PIPE (endchar ?)',
   'cms listfile * * a (noheader label',   /* Put list in pipeline    */
   '| a:locate 10.5 /EXEC /',            /* Find the execs          */
   '| f:faninany',                      /* Combine the streams     */
   '| > WANTED FILES A',                /* Write the list in a file */
   '?',                                 /* non-EXEC file types go here */
   'a:',
   '| locate 10-16 /SCRIPT /',          /* Find any SCRIPT files   */
   '| f:'                               /* Route them to FANINANY  */
```
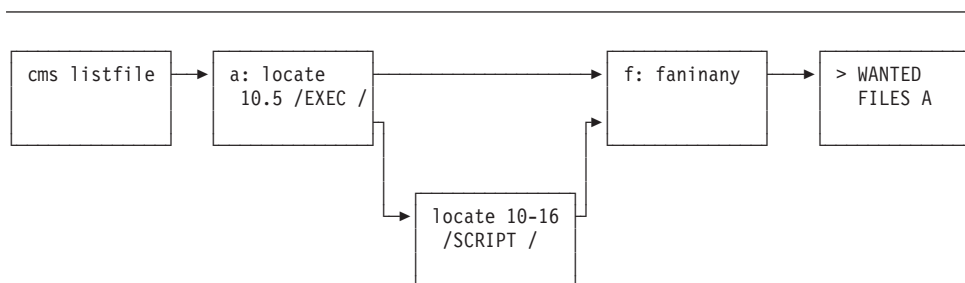
Figure 172. LFD EXEC: A FANINANY Example

The CMS stage issues a LISTFILE command that puts a list of all files on file mode A in the pipeline. The next stage, LOCATE, finds the files having a file type of EXEC. LOCATE writes matching records to its primary output stream. These records flow into FANINANY. LOCATE writes records that do not match to its secondary output stream. A second pipeline, which begins with a stage containing only the label a, processes the rejects.

The second pipeline looks for any SCRIPT files, again with a LOCATE stage. Any records that match are sent to the next stage. The next stage contains the label f. This connects the pipeline to the secondary input of FANINANY.

FANINANY reads records from its primary and secondary inputs in whatever order they arrive. FANINANY writes all records to its primary output. In effect, FANINANY combines its two input streams. The next stage, a > stage, writes them to a file.

The records in WANTED FILES are in the same relative order as they were in the output from the CMS LISTFILE command. This is because the multistream portion of this pipeline contains only a LOCATE stage and LOCATE does not *delay the records*. We will talk about delaying the records later in this chapter. Use SORT after FANINANY if you wish the lines in a specific order.

Identifying Streams

In the operands of some stages that we'll be discussing, you'll need to refer to specific streams. So far we've been referring to streams as *primary input streams*, *secondary input streams*, and so on. However, these terms cannot be used as stage operands. Instead, we use *stream numbers* or *stream names* as stage operands to identify a particular stream.

Stream Numbers

Stream numbers apply to a single stage. Each stage has a primary stream, which is stream 0. If the stage also uses a secondary stream, that stream number is 1. A tertiary stream is number 2, and so on. When several streams are fed into a stage or flow out of a stage, the pipelines are associated with streams in the order in which the pipelines are written. For example, the FANOUT stage in the following example uses four output streams: the primary and three others. The primary stream is stream 0. The other pipelines are connected to streams in the order that the pipelines are written:

```
/* FANOUT using four output streams                           */
'pipe (endchar ?)',
   '< master file',
   '| a: fanout',          /* Primary stream (stream 0)        */
   '| > all data a',
   '?',
   'a:',                   /* This LOCATE stage is connected to */
   '| locate 36.4 /1990/', /* FANOUT's output stream 1         */
   '| > 1990 data a',
   '?',
   'a:',                   /* This LOCATE stage is connected to */
   '| locate 21.4 /MALE/', /* FANOUT's output stream 2         */
   '| > male data a',
   '?',
   'a:',                   /* This LOCATE stage is connected to */
   '| locate 45.2 /NY/',   /* FANOUT's output stream 3         */
   '| > ny data a'
```

A map of the pipeline, with FANOUT's stream numbers, is shown in Figure 173.
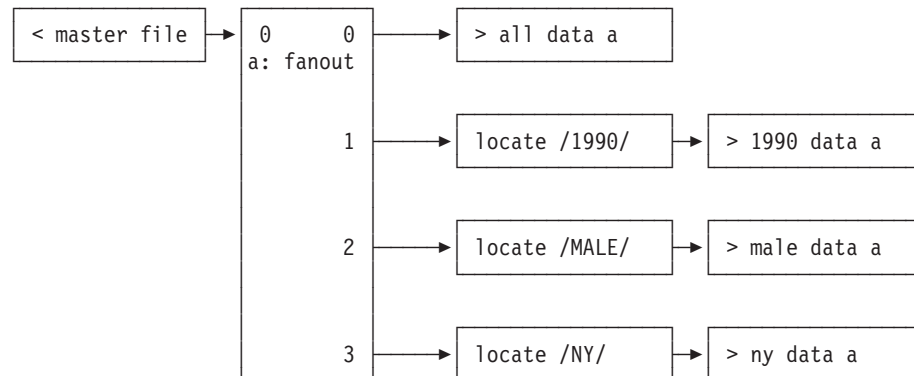
Figure 173. Map of Pipeline Showing Stream Numbers

Notice that all the other stages have connections to only their primary input and output streams (stream 0).

Stream Names

A *stream name* is a stream identifier that assigns a symbolic name to a stream. You name a stream to avoid keeping track of its stream number. Instead of using a stream number as an operand on a stage that combines streams, you can refer to that stream by the stream name you have assigned.

You may name a stream on the PIPE command, and on the ADDPIPE and CALLPIPE subcommands like this: add an identifier to the label by writing the label (in this case a), immediately followed by a period (.) and up to 4 alphabetic

characters or a combination of alphabetic characters and digits that includes at least one alphabetic character. A stream identifier must be immediately followed by a colon with no intervening blanks.

Simpler rules apply to the ADDSTREAM subcommand, where you name a stream as described above, but without the surrounding label or colon. For instance:

```
a.mstr:
```

assigns the symbolic name `mstr` to a stream that can be referenced by a stage. See Figure 177 on page 121 for an example that uses a stream number to identify the primary input stream and a stream name to identify the secondary input stream.

FANIN Stage

Like FANINANY, FANIN reads records from its input streams and writes those records to its output stream. But, unlike FANINANY, FANIN reads all the records from one stream before reading records from another. By default, FANIN reads all of its primary stream, then all of its secondary, and so on, until it has processed all input streams. Stream numbers or stream names can be entered as operands to specify a different order, or a subset of all connected streams, or both.

For example, one way to read TEST DATA and TEST1 DATA into a pipeline is shown in Figure 174. Another way is to use APPEND (see Figure 124 on page 72).
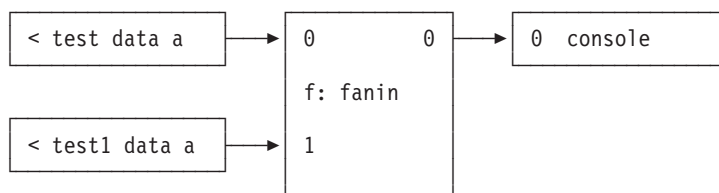
Figure 174. Map of FANIN Example

Figure 175 shows the PIPE command.

```
/* FANIN example showing default stream order                   */
'pipe (endchar ?)',
   '< test data a',        /* Read first file                   */
   '| f: fanin',           /* Gather streams in default order   */
   '| console',            /* Display files                     */
   '?',
   '< test1 data a',       /* Read second file                  */
   '| f:'                  /* Feed it to FANIN (is FANIN stream 1) */
exit rc
```

Figure 175. FANIN Example Showing Default Stream Order

Figure 175 defines two streams into FANIN. The two input streams are connected to stages that read the desired files.

To specify a different order, put stream numbers after the FANIN keyword, as shown in Figure 176. In that example, FANIN reads all of secondary input stream 1, then all of primary input stream 0. The order in which the files are displayed is reversed. TEST1 DATA A is displayed followed by TEST DATA A.

```
/* FANIN example showing stream numbers                        */
'pipe (endchar ?)',
  '< test data a',        /* Read first file                  */
  '| f: fanin 1 0',       /* Request a specific order on FANIN   */
  '| console',            /* Display files                    */
  '?',
  '< test1 data a',       /* Read second file                 */
  '| f:'                  /* Feed it to FANIN (is FANIN stream 1) */
exit rc
```

Figure 176. FANIN Example Showing Stream Numbers

Figure 177 shows the same example with a stream name identifying the secondary input stream. When you use stream names, you don't need to keep track of the stream numbers.

```
/* FANIN example showing stream identifiers                    */
'pipe (endchar ?)',
  '< test data a',        /* Read first file                  */
  '| f: fanin td1 0',     /* Request a specific order on FANIN */
  '| console',            /* Display files                    */
  '?',
  '< test1 data a',       /* Read second file                 */
  '| f.td1:'              /* Feed it to FANIN as stream TD1     */
exit rc
```

Figure 177. FANIN Example Showing Stream Identifiers

OVERLAY Stage

OVERLAY reads records from all its input streams and creates a record that is the overlay of them all, in the sense of the XEDIT OVERLAY subcommand. Each character of the output is from the record from the highest numbered stream with a nonblank character in the corresponding position. Unlike the XEDIT OVERLAY command, underscores are treated like any other character.

In the following example, we create lines of text for a direct mail marketing campaign. The file MASTER FILE contains a mailing list, one name per record. The file DIRMAIL SCRIPT contains a single line of text to be printed on the envelopes. We want to overlay the standard line of text with personal information extracted from the records in MASTER FILE. The line of text contains space for the personal information.

Figure 178 on page 122 shows a map of what we need to do. The TAKE stage is used to limit the number of MASTER FILE records being processed. SPECS is used to extract information from the MASTER FILE records. DUPLICATE duplicates the record from DIRMAIL SCRIPT. A standard line of text and the desired personal information flow into OVERLAY. OVERLAY combines the two records and then CONSOLE displays the results.
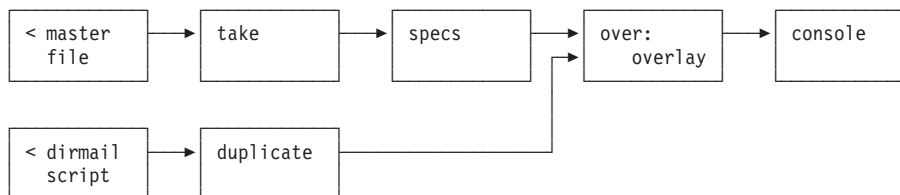
Figure 178. Map of Overlay Example

The complete exec, named OVERLAY, is in Figure 179.

```
/* OVERLAY EXEC -- Overlay date for direct mail envelopes        */
numrecs=3                     /* Number of master file records to do  */
'pipe (endchar ?)',
   '< master file',          /* Read the MASTER FILE                  */
   '| take' numrecs,         /* Take desired number of records        */
   '| specs 36.4 18.4',      /* Create output record containing year  */
   '| over: overlay',        /* Overlay year with direct mail script  */
   '| console',              /* Display records                       */
   '?',
   '< dirmail script',       /* Read the direct mail script           */
   '| duplicate' numrecs-1,  /* Make correct number of copies         */
   '| over:'                 /* Feed to secondary input of OVERLAY    */
exit rc
```

Figure 179. Example of OVERLAY Stage: OVERLAY EXEC

The variable numrecs is set to the number of MASTER FILE records to be
processed. This number is used in the TAKE stage and in the DUPLICATE stage to
create the correct number of output records. The SPECS stage extracts a field
containing the year of birth from the MASTER FILE input record. It positions the
year of birth in the output record such that it will overlay the space reserved in the
standard text. Below is an example run. Note the location of the birth dates in the
MASTER FILE records:

```
pipe < master file | console
SMITH, ROBERT M.    MALE      12 06 1956      NY
JONES, MORGAN E.    FEMALE    05 05 1959      PA
PUBLIC, WALDO Q.    MALE      11 13 1960      CA
Ready;
pipe < dirmail script | console
Were you born in     ?  Read the important message inside!
Ready;
overlay
Were you born in 1956?  Read the important message inside!
Were you born in 1959?  Read the important message inside!
Were you born in 1960?  Read the important message inside!
Ready;
```

SPECS, Revisited

SPECS supports multiple input streams with the keyword SELECT. It reads one
record from all its input streams for each output record built.

Use the keyword SELECT followed by a stream number or a stream name
whenever you wish to refer to input data from a record on a stream other than the

primary one (or the one selected previously in the list of operands). Subsequent input fields (up to the next SELECT keyword) refer to the record on the stream specified.

Figure 181 shows an example that reads a file, changes some of its records, and then displays the original and changed records side-by-side. (The first 35 characters of each record are displayed.) The map for the pipeline is in Figure 180.
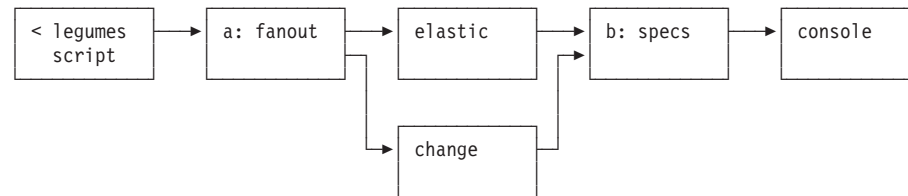

Figure 180. Map of SPECS SELECT Example

```
/* SELECT EXEC -- Demonstrate SPECS Select operand              */
'pipe (endchar ?)',
   '< legumes script',              /* Read file                     */
   '| a: fanout',                   /* Copy records to secondary output */
   '| elastic',                     /* Prevent a pipeline stall      */
   '| b: specs 1-35 1',             /* Get data from primary stream (0) */
      'select 1',                   /* Select the secondary stream (1) */
      '1-35 41',                    /* Put data in second half of record */
   '| console',                     /* Display the results           */
   '?',
   'a:',                            /* Records from FANOUT           */
   '| change /Pole beans/**Crop Failure**/',  /* Change the data      */
   '| b:'                           /* Send back to SPECS            */
exit rc
```

Figure 181. Example of SPECS SELECT Operand: SELECT EXEC

In the example, the < stage reads the file LEGUMES SCRIPT and writes the records to its output stream. FANOUT copies the records to its secondary stream, where CHANGE is used to change some of the data. The SPECS stage puts the original record and the changed record side-by-side on a single output record, which CONSOLE then displays.

ELASTIC stage is used between FANOUT and SPECS to avoid a *pipeline stall*. A pipeline stall is a new kind of error you can get with multistream pipelines. A stall occurs when the dispatcher cannot run any of the stages because every stage is waiting for some other stage to perform some action. CMS Pipelines detects a stall and ends the PIPE command. (Try running the exec after removing the ELASTIC stage.)

A stall occurs when ELASTIC is omitted because SPECS needs two records at a time, but FANOUT writes one record to each output stream in turn. When FANOUT writes a record to stream 0, it waits for SPECS to read it. SPECS tries to read the record as well as one on its secondary input stream. But, there isn't a record available in the secondary stream—FANOUT cannot write it because it is waiting for its first output to finish. In effect, SPECS and FANOUT are waiting for each other.

This causes all the other stages to wait. The < stage waits for FANOUT to read the next record. The CONSOLE stage waits for SPECS to write a record. Thus, we have a stall.

By inserting ELASTIC, the stall is avoided. In this case, ELASTIC reads the record that FANOUT writes to its primary output stream. FANOUT is then free to write a record to its secondary output stream. When SPECS demands its two records, CMS Pipelines is able to make the records available.

Here is an example run:

```
pipe < legumes script | console
Peas
Bush beans
Pole beans
Lima beans
Ready;
select
Peas                                    Peas
Bush beans                              Bush beans
Pole beans                              **Crop Failure**
Lima beans                              Lima beans
Ready;
```

SPECS stops when all input streams are empty. If one stream empties before the others, SPECS acts as though that stream contains null records.

COUNT, Revisited

COUNT is an unusual stage. It writes different outputs depending on whether its secondary stream is connected. Previous COUNT examples showed how COUNT works when its secondary stream is not connected. It writes a single record containing the requested tallies to its primary output. When the secondary stream is connected, however, this is not the case.

When the secondary stream is connected, COUNT copies its *input stream*, not the tally record, to its primary output. The record containing the tallies is written to the *secondary* output stream instead of the primary.

This makes COUNT far more useful. Look at the WORDUSE EXEC in Figure 182 on page 125. It lists all the words that occur in a file in alphabetic order. Preceding each word is a number indicating the number of times the word was used. After the list of words, a summary is displayed that tells the number of lines in the file, the total number of words, and the number of unique words.

```
/* WORDUSE EXEC -- Word Use Analyzer                                */
parse arg fid
'pipe (endchar ?)',
  '<' fid,                      /* Read the file              */
  "| xlate lower 41-7f 40 ' ' ", /* Translate to lowercase, and get */
         ,                      /*    rid of punctuation      */
  '| a: count words lines',     /* Count words and lines      */
  '| split',                    /* Put one word per record    */
  '| sort count',               /* Sort unique with a count of dups */
  '| b: count lines',           /* Count the number of unique words */
  '| specs 1-10 1 11-* 15',     /* Format the sorted records  */
  '| f: fanin',                 /* Make sure summary lines at end */
  '| literal Times Used   Word',  /* Write a header           */
  '| console',                  /* Display it                 */
  '?',
  'a:',                  /* Process the word and line count      */
  '| specs /Number of lines: / 1 words2 next write',
       '/Total number of words: / 1 words1 next',
  '| f:',
  '?',
  'b:',                  /* Process the unique word count        */
  '| specs /Total number of unique words: / 1 words1 next',
  '| f:'
```

Figure 182. WORDUSE EXEC: Example Exec to Analyze Word Use

After the file is read, the records are translated to lowercase (XLATE) because we don't want the results to be case-sensitive. XLATE also eliminates punctuation by changing characters in the range X'41' to X'7F' to blanks (X'40'). The pair of apostrophes causes apostrophes to be retained in the records. Otherwise, all contractions (for example, "don't") and some possessives (for example, "Mary's") would be split into two words.

Next the lines and words are counted, but the record containing the count is written to COUNT's secondary stream (label a). In this example, the COUNT stage copies its input to its output stream, for processing by SPLIT.

SPLIT puts each lowercase word on a separate record, so the words can be sorted by the SORT stage. (SORT sorts records, not items *within* records.) The COUNT operand on SORT causes SORT to eliminate duplicate records. The number of duplicates for each record is placed in the first ten bytes of the output record. The record itself follows, beginning with byte 11. So, SORT COUNT gives us the number of times each word was used. (More about SORT is in "Sorting Records (SORT)" on page 49.)

Each record flowing out of SORT contains a unique word. To count the total number of unique words, a COUNT LINES stage is added, and the record containing the count is written to the secondary output (label b:). COUNT passes the records themselves to the next stage.

The remainder of the first pipeline formats the records, collects the summary records from the other pipelines, adds a header, and displays everything. The two other pipelines process records from the two COUNT stages. They add explanatory text to the counts and send the formatted records back to the first pipeline.

Notice that FANIN is used to collect the records from all three streams. FANINANY is not used because we want the list of words to precede the summary records. FANIN guarantees this. Try substituting FANINANY in WORDUSE—the records are not displayed in the desired order.

Figure 183 shows a sample run of WORDUSE. The input file TEST DATA contains these lines:

```
=======================================
Don't worry about me--I can take
care of myself.
```

```
worduse test data
Times Used    Word
         1    about
         1    can
         1    care
         1    don't
         1    i
         1    me
         1    myself
         1    of
         1    take
         1    worry
Number of lines: 3
Total number of words: 10
Total number of unique words: 10
Ready;
```

Figure 183. Counting Several Items

MERGE Stage

MERGE copies records from all its input streams to its primary output stream. MERGE is intended to combine sorted lists such that the output from MERGE is in order. When the records in the input streams are not sorted, MERGE still combines the records. In this case, however, the order in which MERGE writes the records is not predictable. MERGE does not verify that the input streams are sorted.

You can specify column ranges on MERGE in the same way as for SORT. When column ranges are specified, MERGE orders the records according to the data in the specified ranges. If you omit column ranges, MERGE uses the whole record to determine the order.

Figure 184 on page 127 shows the map of a PIPE command that combines the responses from two CMS LISTFILE commands. Because the responses from CMS LISTFILE are sorted, the output from MERGE will be in order.
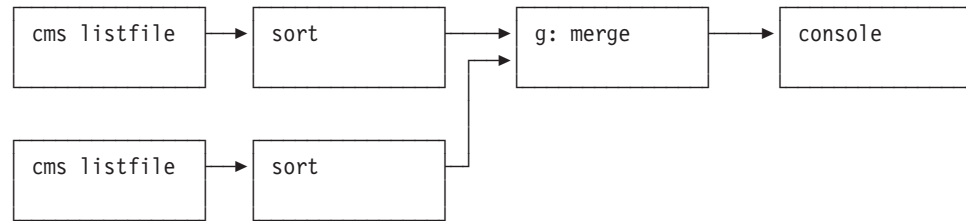
Figure 184. Map of MERGE Example

Figure 185 shows the PIPE command for the above map. The CMS stages list the files on file mode A and file mode C. These lists are then sorted. The output streams of both SORT stages are connected to the input streams of MERGE. Because column ranges are omitted from MERGE, it uses the entire record to determine the output order. CONSOLE displays the records from MERGE.

```
/* LISTMRG EXEC -- Display merged list of all files on two file modes */
'pipe (endchar ?)',
   'cms listfile * * a',   /* List all files on first file mode     */
   '| sort',               /* Sort the list                         */
   '| g: merge',           /* Merge the sorted lists                */
   '| console',            /* Display the merged and sorted list    */
   '?',
   'cms listfile * * c',   /* List all files on second file mode    */
   '| sort',               /* Sort the list                         */
   '| g:'                  /* Feed to MERGE                         */
exit rc
```

Figure 185. MERGE Stage Example: LISTMRG EXEC

The following is an example run of LISTMRG EXEC:

```
listmrg
$SHRLIS$ XEDIT    C2
ADD      REXX     A1
ADDGROUP HELPRACF C1
ADDUSER  HELPRACF C1
ALL      NOTEBOOK A0
ALTDSD   HELPRACF C1
ALTGROUP HELPRACF C1
ALTSEQ   HELPDFSO C1
ALTUSER  HELPRACF C1
ASCLM    EXEC     C1
ASPF     EXEC     C1
AUTH     NOTE     A1
AUTHOR   REXX     A1
AUTHOR   T01      A1
AUTHOR   T02      A1
AUTHOR   T03      A1
AUTHOR   T04      A1
AUTOLOG  HELPBXF  C5
   ⋮
```

LOOKUP Stage

LOOKUP matches records in its primary input stream with records in its secondary input stream and writes matched and unmatched records to different output

streams. Whole contents of records are matched by default, or the records are matched on the basis of a *key field* (the contents of a specified range of columns in the records).

Before finding records, LOOKUP builds the *reference*. It does so by reading all records on the secondary input stream into a buffer (called the *reference*). These records are called *master records*. LOOKUP discards master records with duplicate keys while loading the buffer.

After building the reference, LOOKUP reads records from its primary input stream and looks for a matching record in the set of reference records. The records read from the primary input stream are referred to as *detail records*. By default, entire records are compared, but you can specify column ranges to look for a key.

Upon finding a match, LOOKUP, by default, writes the detail record and the matching master record to its primary output stream. Use the operand DETAILS to get just the detail records. If a detail record does not have a matching master record, LOOKUP writes the detail record to its secondary output stream.

After processing all the detail records, LOOKUP writes all unreferenced master records to its tertiary output stream. By unreferenced we mean those not matched by at least one detail record. LOOKUP writes the unreferenced records in ascending order by their keys.

Figure 186 summarizes the streams used by LOOKUP.

```
Detail in ──────  0        0  ────── Matching detail and master

                     LOOKUP
Master in ──────  1        1  ────── Unmatched detail

                          2  ────── Unreferenced master
```
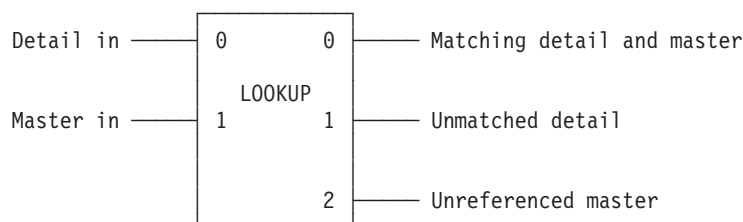
Figure 186. Map of LOOKUP Stage

You can see LOOKUP work by executing an exec like the one in Figure 187 on page 129.

```
/* LOOKSTR EXEC -- demonstrate LOOKUP streams                   */
'pipe (endchar ?)',
   'literal car'||,             /* Record to search for...       */
   '| literal snowmobile'||,    /* Another record to search for... */
   '| l: lookup',               /* Look for the records          */
   '| specs /Primary: / 1',     /* LOOKUP writes matching detail and */
      '1-* next',               /*   master records to primary output */
   '| console',                 /* Display records               */
   '?',
   'literal truck'||,           /* Master record for reference   */
   '| literal boat'||,          /* Master record for reference   */
   '| literal car'||,           /* Master record for reference   */
   '| l:',                      /* Connect to secondary input & output */
   '| specs /Secondary: / 1',   /* LOOKUP writes unmatched detail to */
      '1-* next',               /*   its secondary output stream */
   '| console',                 /* Display unmatched detail records */
   '?',
   'l:',                        /* LOOKUP writes unreferenced master */
   '| specs /Tertiary: / 1',    /*   records to its tertiary output */
      '1-* next',               /*   in ascending order          */
   '| console'                  /* Display unreferenced master records */
exit 0
```

Figure 187. LOOKUP Stage Example: LOOKSTR EXEC

In the first pipeline, LITERAL stages are used to create detail records. LOOKUP writes the matching detail and master records to its primary output stream. SPECS prefixes the records with identifying text and CONSOLE displays the records.

In the second pipeline, the reference is created. LITERAL stages are used to create master records. These records flow into the secondary input of LOOKUP. The label l makes the connection. That same label also makes a connection to the secondary output stream of LOOKUP. (See "Connecting to Both the Secondary Input and the Secondary Output" on page 113 for more about this type of connection.) LOOKUP writes the unmatched detail records to its secondary output. SPECS prefixes the records with identifying text, and CONSOLE displays them.

In the third pipeline, the label l connects the tertiary output of LOOKUP. LOOKUP writes unreferenced master records to its tertiary output. Again SPECS and CONSOLE are used to display the records.

The following example shows the response from LOOKSTR:

```
lookstr
Secondary: snowmobile
Primary: car
Primary: car
Tertiary: boat
Tertiary: truck
Ready;
```

LOOKUP writes `snowmobile` to its secondary output stream because there isn't a matching reference record. The record `car` is written twice: the first record is the detail record, while the second is the matching master record. The records `boat` and `truck` are unreferenced master records. LOOKUP writes unreferenced master records to its tertiary output stream.

Another LOOKUP example is shown in Figure 189 on page 130. (A map is shown in Figure 188 on page 130.) The example (VALIDATE REXX) shows the use of column ranges on LOOKUP to identify a search key. It also shows the use of the

DETAILS operand. When DETAILS is specified, LOOKUP writes matching detail records to its primary output stream, but not the corresponding master record.
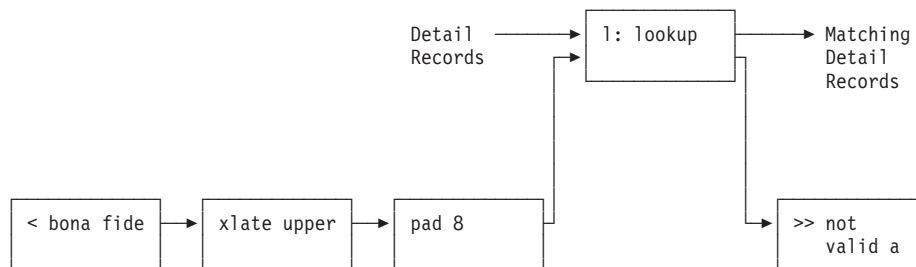

Figure 188. Map of VALIDATE REXX

VALIDATE REXX checks the authorization of the user ID in the first eight columns of the input record against the list of authorized users stored in the file BONA FIDE. Unauthorized requests are appended to a log file.

```
/* VALIDATE REXX -- Ensure commands are from authorized users        */
'callpipe (endchar ?)',
   '*:',                          /* Read input from calling pipeline   */
   '| l: lookup 1.8 details',     /* Search reference                   */
   '| *:',                        /* Pass good ones on                  */
   '?',
   '< bona fide',                 /* Read list of authorized users      */
   '| xlate upper',               /* Make them uppercase                */
   '| pad 8',                     /* Pad them to 8 characters           */
   '| l:',                        /* Feed to LOOKUP and get unmatched   */
   '| >> not valid a'             /* Write unmatched data               */
exit rc
```

Figure 189. VALIDATE REXX: Example of LOOKUP

The first pipeline consists of LOOKUP, an input connector, and an output connector. VALIDATE REXX expects its input stream to contain records to be validated. Those that are valid (that is, found in the reference) are written to the output stream. The column range 1.8 is used to define the key to be used in the search. DETAILS is specified to avoid having the master record written to the output stream.

The second pipeline reads the file BONA FIDE, prepares the records, and writes those records to LOOKUP's secondary input stream. LOOKUP builds the reference from these records. LOOKUP writes unmatched detail records to its secondary output. These records flow into the >> stage and are written to the file NOT VALID A.

In the following example, uppercase user IDs are used because LOOKUP is case-sensitive.

```
pipe < bona fide | console
Bill
Ted
Denise
Mike
Ready;
pipe literal TED     This should get through. | validate | console
TED     This should get through.
```

```
Ready;
pipe literal BOGUS   This should fail. | validate | console
Ready;
pipe < not valid | console
BOGUS   This should fail.
Ready;
```

Pipeline Stalls

With multistream pipelines you may see a new kind of error: a stall. A stall occurs when the dispatcher cannot run any of the stages because every stage is waiting for some other stage to perform some function. Usually stalls are caused by stages that read multiple input streams in a particular order or that need records to be available on more than one stream at the same time. A stall occurs when the preceding stages do not deliver records in the order needed or do not provide multiple records concurrently.

When a stall occurs, you receive a return code of -4095, messages saying the pipeline is stalled, and messages listing the state of all the stages in the pipeline. A dump of the control block structure is automatically written to a file named PIPDUMP LIST*nnnn* (where *nnnn* is a number). This file is intended for use by service personnel.

What can you do when a pipeline stalls? First, look at any stages that have multiple input streams. Of these stages, identify any stages that need records in a particular order (such as FANIN) or that need more than one record at a time (such as SPECS and OVERLAY).

Next, analyze the pipeline to see what order the records are being delivered to these stages. You need to look at the stages that are supplying records. It helps to draw a map of the pipeline. Look for earlier stages that have secondary outputs connected. These stages often deliver records in a particular order and that order is not what the stage combining the streams needs.

After finding the problem, you can either change the order of delivery of the records, or you can change the stage combining the streams to match the supplied order.

A common stall is shown in Figure 190. There may be many other stages between FANOUT and FANIN, but in many cases these other stages don't matter. It is the FANOUT/FANIN combination that causes the problem. FANIN needs to read all the records of one stream before reading the next. FANOUT, however, writes one record to each of its output streams.
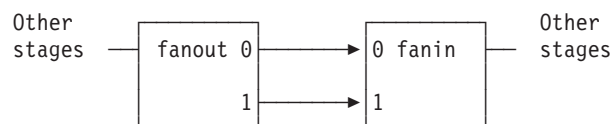

Figure 190. Example Stall Involving a Stage that Needs Records in Order

FANOUT writes a record on its output stream 0. FANIN reads this record. Then FANOUT tries to write a copy of the record on its output stream 1. It waits for

FANIN to read the record, but FANIN is waiting for FANOUT to write another record on stream 0. FANIN will not read from its input stream 1 until input stream 0 is empty. The pipeline is stalled.

Figure 191 shows one way to fix the stall. A BUFFER stage is added. BUFFER doesn't write any records to its output stream until it has read all the records in its input stream. So, when FANOUT writes a record to its output stream 0, FANIN reads that record. When it writes to stream 1, the BUFFER stage reads the record. After FANOUT writes its last record to stream 1, BUFFER writes the records to its output stream. Because FANIN has, by this time, processed all the records in its input stream 0, it can now process the records supplied by BUFFER on stream 1.
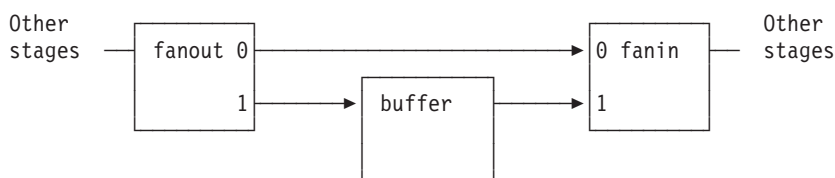
```
Other                                                      Other
stages  ──┌─────────┐                    ┌─────────┐       stages
          │ fanout 0├──────────────────→ │0 fanin  ├──
          │         │       ┌────────┐   │         │
          │        1├─────→ │ buffer ├─→ │1        │
          └─────────┘       │        │   └─────────┘
                            └────────┘
```

Figure 191. Fixing a Stall with a BUFFER Stage

You can also fix the stall by replacing the BUFFER stage in Figure 191 with another built-in stage called ELASTIC. The ELASTIC stage works like BUFFER. However, ELASTIC reads only as many records into a buffer as necessary to prevent a stall.

Another solution is to substitute FANINANY for FANIN. FANINANY does not care about the order in which records are delivered to it. Of course, the order of records flowing out of FANINANY is not the same order that FANIN would have provided. So, FANINANY might not be an acceptable solution.

If FANINANY fixes the stall, but delivers the records in the wrong order, you might be able to work around it. You could, in some cases, use SPECS with a RECNO operand to put numbers on the records. After the records are processed and combined by FANINANY, add a SORT stage to put the records in order. Then use another SPECS stage to remove the record numbers.

Figure 192 shows another common stall. In this case a SPECS stage contributes to the problem. It expects two records at a time.
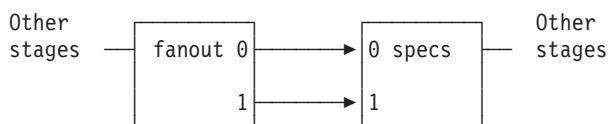
```
Other                                    Other
stages  ──┌─────────┐      ┌─────────┐   stages
          │ fanout 0├────→ │0 specs  ├──
          │         │      │         │
          │        1├────→ │1        │
          └─────────┘      └─────────┘
```

Figure 192. Example Stall Involving a Stage that Needs Multiple Records

In this case, FANOUT writes a record on its output stream 0. It is waiting for its OUTPUT operation to finish. SPECS tries to read records from both of its input streams. Because FANOUT hasn't yet written a record on its output stream 1, SPECS waits. While SPECS waits, FANOUT's OUTPUT operation cannot complete. So, it cannot write a record on its output stream 1. The pipeline is stalled.

Figure 193 shows a solution. HOLD REXX (see Figure 130 on page 84) breaks the stall. HOLD simply reads a record from its input stream and writes it to its output stream. By doing this, HOLD lets FANOUT complete its OUTPUT operation on stream 0. FANOUT can then write a record on its output stream 1. When SPECS is dispatched, the two records it needs are available.
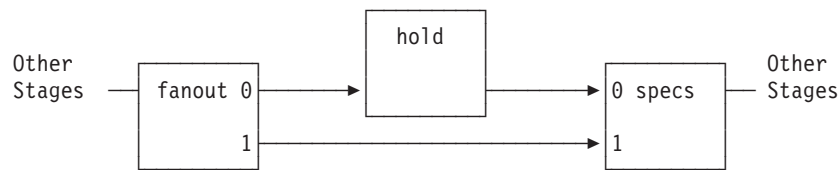
```
                              ┌──────────┐
                              │   hold   │
Other        ┌───────────┐    └──────────┘    ┌──────────┐   Other
Stages  ─────│ fanout  0 │──────────────▶     │ 0  specs │── Stages
             │         1 │─────────────────▶  │ 1        │
             └───────────┘                    └──────────┘
```

Figure 193. Fixing a Stall with HOLD REXX

In some situations, you may be able to fix a stall by using either HOLD REXX or a BUFFER stage. When deciding which one to use, remember that BUFFER holds all the records it reads in virtual storage while HOLD REXX does not. If you want to conserve virtual storage, HOLD REXX is a better choice.

BUFFER is not the only stage that reads all records from its input stream before writing them. SORT, by the nature of its processing, also does this. You may also have some user-written stages at your disposal that buffer records. Assuming you need the functions provided by these stages, they can be used instead of BUFFER to break stalls or to avoid stalls when designing the pipeline.

Although the stalls we have shown involved FANOUT stages, a FANOUT stage is not a prerequisite for a stall. Consider the example in Figure 194. In this case, a LOCATE stage is involved.
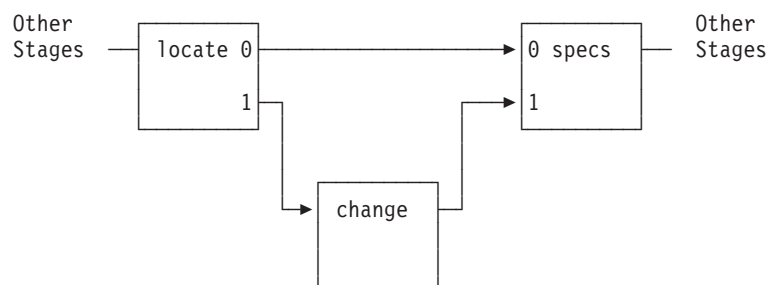
```
Other        ┌───────────┐                    ┌──────────┐   Other
Stages  ─────│ locate  0 │─────────────────▶  │ 0  specs │── Stages
             │         1 │───────┐        ┌─▶  │ 1        │
             └───────────┘       │        │    └──────────┘
                                 │        │
                                 ▼        │
                           ┌──────────┐   │
                           │  change  │───┘
                           └──────────┘
```

Figure 194. Example Stall Involving a LOCATE Stage

Because of the way the LOCATE is connected, only one record at a time is delivered to SPECS. To fix the stall you need to add two BUFFER stages: one between LOCATE and SPECS and the other between CHANGE and SPECS. All records that flow on either stream are held by the BUFFER stages until all the input is read. Then SPECS can read the records from the BUFFER stages two at a time. If one stream has more records than the other, the stream with fewer records will be disconnected from SPECS when all the records are read. (SPECS can handle this condition.)

Maintaining the Relative Order of Records

Although the order in which the dispatcher runs stages is unpredictable, in certain situations the relative output order of records involving a multistream pipeline is predictable. This section explains how to write multistream pipelines so the order of the output records is predictable. The following concepts are discussed:

- How each stage of a pipeline runs
- How stages delay the records
- How to predict relative record order.

How Each Stage of a Pipeline Runs

As we discussed in "The CMS Pipelines Environment" on page 78, a stage does not run from start to finish once the dispatcher gives it control. Once a stage has written a record to its output stream, that stage is *blocked*, which means it cannot run again until the stage connected to its output stream consumes the record. A stage *consumes a record* when it reads a record from its input stream and removes the record from that input stream. Once a record has been consumed by a stage, it cannot be read again by that stage. A stage is also blocked when it is waiting to read a record, but no records are currently available.

To apply this concept to a user-written stage, a stage that issues an OUTPUT pipeline subcommand is blocked until the stage connected to its output stream consumes the record with a READTO pipeline subcommand. Although the PEEKTO pipeline subcommand can also be used to read records, PEEKTO does not consume records from its input stream.

How Stages Delay the Records

Let's consider the following command:

```
PIPE A | B | C
```

If stage A writes a record to its output stream, stage A stops running until the record is consumed by stage B. Therefore, stage B determines when stage A can continue to run. Stage B can process its records in one of the following ways:

1. It can read a record and remove it from its input stream. This allows stage A to continue running. This is how READTO processes records.

2. It can read a record without removing the record from its input stream. This prevents stage A from running. This is how PEEKTO processes records.

This implies that there are two methods for writing stages. The first method *delays the records*. The second method does not delay the records. When a stage delays the records, a record's progress through the stage is buffered, or held up. This occurs when the output of stage A is consumed by stage B allowing stage A to resume running before stage B writes its output. When a stage does not delay the records, the records can progress through the stage without being held up or buffered.

To determine whether a built-in stage delays the records, refer to the usage notes section of the stage's description in the *z/VM: CMS Pipelines Reference*. Don't confuse the DELAY stage with delaying the records.

To write a user-written stage that does not delay the records, use the following logic:

1. Use PEEKTO to read a record without consuming it
2. Process the contents of the record
3. Use OUTPUT to write one or more records to the output stream

4. Use READTO to consume the record previously read with PEEKTO
5. Repeat steps 1-4 as needed.

How to Predict Relative Record Order

In order to maintain the relative order of records in a set of multistream pipelines, the pipelines must:

*   Start at one common stage
*   Be split into multiple pipelines using only stages that do not delay the records
*   Contain only stages that do not delay the records
*   Be combined into a single stream using a stage that combines multiple streams as records arrive (for example, FANINANY).

By using the process shown in the following examples, you can determine how records flow through a set of pipelines.

Note: The examples assume a specific order of execution of the stages involved. The actual execution order as determined by the dispatcher may be different. You can trace the PIPE command to see the actual order in which the dispatcher runs the stages. Note that a subsequent trace of the same command may show the stages running in a different order.

Example 1 - Not Delaying the Records

Look at the following PIPE command (a map is shown in Figure 195):

```
/* NODELAY EXEC */
'pipe (endchar ?)',
  '< DELAY INPUT',              /* Read DELAY INPUT file */
  '| l: locate /a/',            /* Find records containing a */
  '| xlate upper',              /* translate records to uppercase */
  '| f: faninany',              /* combine streams back together */
  '| > DELAY OUTPUT A',         /* write result to DELAY OUTPUT A */
  '?',                          /* beginning of second pipeline */
  'l:',                         /* define 2ndary output for locate */
  '| f:'                        /* define 2ndary input for faninany */
```
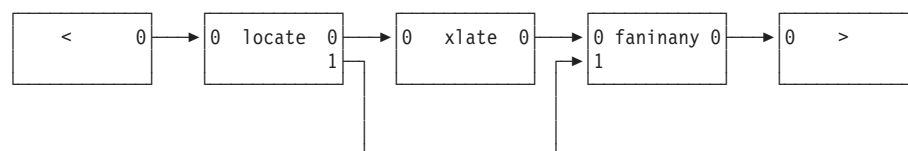

Figure 195. Map of NODELAY EXEC

Assume that the file DELAY INPUT contains the following records:

```
a1
b2
a3
b4
```

When you run NODELAY EXEC, conceptually the following takes place:

1. The < stage reads the first record `a1` from the disk or directory and writes the record to its output stream. The < stage is now blocked until its output record is consumed.
2. The LOCATE stage starts running. LOCATE finds the record `a1` available and processes it. It searches the record for an `a` and then selects an output stream.

The record contains an `a`, so LOCATE writes the record `a1` to its primary output stream. Because LOCATE has not consumed the `<` stage's output, `<` still cannot run. The LOCATE stage is now blocked until its output record is consumed.

3. The XLATE stage starts running. XLATE looks at its input stream and obtains the record `a1`. XLATE reads the record, translates it to uppercase, and writes the result to its output stream. Because XLATE has not consumed the LOCATE stage's output, LOCATE still cannot run. The XLATE stage is now blocked until its output record is consumed.

4. The FANINANY stage starts running. It finds a record available on its primary input stream. FANINANY looks at the record and copies it to its output stream. Because FANINANY has not consumed the XLATE stage's output, XLATE still cannot run. The FANINANY stage is now blocked until its output record is consumed.

5. The `>` stage starts running. It looks at its input stream and finds the `A1` record. `>` writes the record to the file. Because there is nothing attached to its output stream, `>` writes no output record. `>` consumes the `A1` record from its input stream making FANINANY eligible to run. `>` then looks for another input record. `>` cannot run until another record is ready for it.

6. FANINANY resumes, consumes its input, and then looks for another input record. FANINANY cannot run until another record is ready for it.

7. XLATE resumes, consumes its input, and looks for another input record. XLATE cannot run until another record is ready for it.

8. LOCATE resumes, consumes its input, and looks for another input record. LOCATE cannot run until another record is ready for it.

9. `<` resumes, reads the record `b2` from the disk or directory, and writes the record to its output stream. The `<` stage is now blocked until its output record is consumed.

10. The LOCATE stage resumes and it looks at its input stream. LOCATE finds the record `b2` available and processes it. It searches the record for an a and then selects an output stream. The record does not contain a so LOCATE writes the record to its secondary output stream. LOCATE has not consumed the `<` stage's output, so `<` still cannot run. The LOCATE stage is now blocked until its output record is consumed.

11. FANINANY resumes. It finds a record available on its secondary input stream. FANINANY looks at the record and copies it to its output stream. FANINANY has not consumed the LOCATE stage's output, so LOCATE still cannot run. The FANINANY stage is now blocked until its output record is consumed.

12. `>` resumes and finds the `b2` record. `>` writes the record to the file and then to its output stream. `>` consumes the `b2` record from its input stream making FANINANY eligible to run. `>` then looks for another input record. `>` cannot run until another record is ready for it.

13. FANINANY resumes, consumes its input, and looks for another input record. FANINANY cannot run until another record is ready for it.

14. LOCATE resumes, consumes its input, and looks for another input record. LOCATE cannot run until another record is ready for it.

At this point, the following records have been written to the output file:

```
A1
b2
```

The process continues for the `a3` and `b4` records until the output file contains:

```
A1
b2
A3
b4
```

Then the following happens:

1. < attempts to read a record and finds that there are no more records in the file. < ends causing its output stream to be disconnected. Because this stream is disconnected, the LOCATE stage knows the end of the file is reached and the stage becomes eligible to run.

2. LOCATE determines that its input stream is disconnected. LOCATE ends causing its output streams to be disconnected. Because these streams are disconnected, the XLATE stage knows the end of the file is reached and the stage becomes eligible to run.

3. XLATE determines that its input stream is disconnected. XLATE ends causing its output stream to be disconnected. Because this stream is disconnected, the FANINANY stage knows the end of the file is reached and the stage becomes eligible to run.

4. FANINANY determines that its input streams are disconnected. FANINANY ends causing its output stream to be disconnected. Because this stream is disconnected, the > stage knows the end of the file is reached and the stage becomes eligible to run.

5. > determines that its input stream is disconnected and ends.

Now the entire set of pipelines has completed processing and the PIPE command ends. Note that every stage in this example looked at its input record, processed it, wrote it out, and then consumed the input record. None of the stages in this example delayed the records.

Once again, remember that the actual order in which the stages run is unpredictable, but the order of the resulting output records is the same.

Example 2 - Can Delay the Records
If you have a multistream pipeline that contains a stage that can delay the records, the order may be unpredictable.

For example, look at the following PIPE command (a map is shown in Figure 196 on page 138):

```
/* CANDELAY EXEC */
'pipe (endchar ?)',
   '< DELAY INPUT',        /* read DELAY INPUT file */
   '| l: locate /a/',      /* find records containing a */
   '| copy',               /* delay by one record */
   '| f: faninany',        /* combine records back together */
   '| > DELAY OUTPUT A',   /* write result to DELAY OUTPUT A */
   '?',                    /* start of second pipeline */
   'l:',                   /* define secondary output for locate */
   '| f:'                  /* define secondary input for faninany */
```
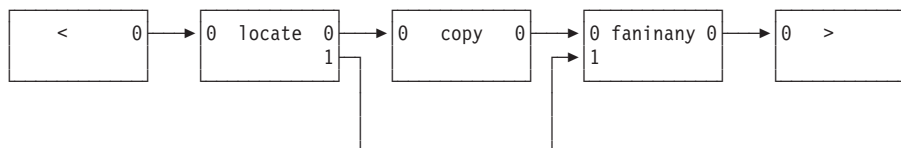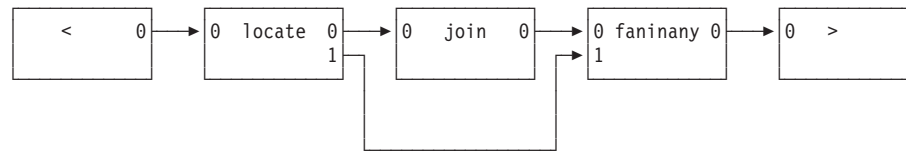
Figure 196. Map of CANDELAY EXEC

Assume that the file DELAY INPUT contains the following records:

```
a1
b2
a3
b4
```

When you run CANDELAY EXEC, conceptually the following takes place:

1. The < stage reads the first record `a1` from the disk or directory and writes the record to its output stream. The < stage is now blocked until its output record is consumed.

2. The LOCATE stage starts running. LOCATE finds the record `a1` available and processes it. It searches the record for an `a` and then selects an output stream. The record contains an `a`, so LOCATE writes the record `a1` to its primary output stream. Because LOCATE has not consumed the < stage's output, < still cannot run. The LOCATE stage is now blocked until its output record is consumed.

3. The COPY stage starts running. COPY looks at its input stream and obtains the record `a1`. COPY saves the record `a1` and consumes it. At this point LOCATE can run or COPY can continue running.

   If LOCATE runs next, it will consume its input enabling < to run. This enables LOCATE to process the `b2` record and write the record to its secondary output stream which would cause `b2` to arrive at FANINANY before the `a1` record.

   If COPY continues to run, it will write the `a1` record to its primary output stream which causes `a1` to be the next record processed by FANINANY.

   Because it is unpredictable whether LOCATE or COPY will run, the order of records in the file DELAY OUTPUT is also unpredictable.

Processing continues until the set of pipelines completes.

Example 3 - Delaying the Records

There are cases in which a multistream pipeline contains stages that delay the records, but the order of the output records can still be predicted. For example, look at the following PIPE command (a map is shown in Figure 197 on page 139):

```
/* DELAY EXEC */
'pipe (endchar ?)',
  '< DELAY INPUT',          /* read DELAY INPUT file */
  '| l: locate /a/',        /* find records that contain an a */
  '| join 1',               /* join pairs of records */
  '| f: faninany',          /* combine records back together */
  '| > DELAY OUTPUT A',     /* write result to DELAY OUTPUT A */
  '?',                      /* start of second pipeline */
  'l:',                     /* define secondary output for LOCATE */
  '| f:'                    /* define secondary input for FANINANY */
```

```
┌──────────────┐   ┌──────────────┐   ┌──────────────┐   ┌──────────────┐   ┌──────────────┐
│ <        0   ├──▶│ 0  locate  0 ├──▶│ 0    join  0 ├──▶│ 0 faninany 0 ├──▶│ 0    >       │
│              │   │            1 │   │              │   │            1 │   │              │
└──────────────┘   └────────────┬─┘   └──────────────┘   └──────────────┘   └──────────────┘
                                │                          ▲
                                └──────────────────────────┘
```

Figure 197. Map of DELAY EXEC

Assume that the V-format file DELAY INPUT contains the following records:

```
a1
b2
a3
b4
```

When you run DELAY EXEC, conceptually the following takes place:

1. < reads the first record a1 from the disk or directory and writes the record to its output stream. The < stage is now blocked until its output record is consumed.

2. The LOCATE stage starts running. LOCATE finds the record a1 available and processes it. It searches the record for an a and then selects an output stream. The record contains an a, so LOCATE writes the record a1 to its primary output stream. Because LOCATE has not consumed the < stage's output, < still cannot run. The LOCATE stage is now blocked until its output record is consumed.

3. The JOIN stage starts running. JOIN looks at its input stream and obtains the record a1. Because JOIN needs two input records to build an output record, JOIN saves the contents of the record, consumes the record, and looks for another input record. JOIN can not run until another record is ready for it.

4. LOCATE resumes, consumes its input, and looks for another input record. LOCATE cannot run until another record is ready for it.

5. < resumes, reads the record b2 from the disk or directory, and writes the record to its output stream. The < stage is now blocked until its output record is consumed.

6. The LOCATE stage resumes and it looks at its input stream. LOCATE finds the record b2 available and processes it. It searches the record for an a and then selects an output stream. The record does not contain a so LOCATE writes the record to its secondary output stream. LOCATE has not consumed the < stage's output, so < still cannot run. The LOCATE stage is now blocked until its output record is consumed.

7. The FANINANY stage starts running. It finds a record available on its secondary input stream. FANINANY looks at the record and copies it to its output stream. Because FANINANY has not consumed the LOCATE stage's output, LOCATE still cannot run. The FANINANY stage is now blocked until its output record is consumed.

8. The > stage starts running. It looks at its input stream and finds the b2 record. > writes the record to the file. Because there is nothing attached to its output stream, > writes no output record. > consumes the A1 record from its input stream making FANINANY eligible to run. > then looks for another input record. > cannot run until another record is ready for it.

9. FANINANY resumes, consumes its input, and then looks for another input record. FANINANY cannot run until another record is ready for it.

10. LOCATE resumes, consumes its input, and looks for another input record. LOCATE cannot run until another record is ready for it.

11. < reads the next record `a3` from the disk or directory and writes the record to its output stream. The < stage is now blocked until its output record is consumed.

12. The LOCATE stage resumes and it looks at its input stream. LOCATE finds the record `a3` available and processes it. It searches the record for an `a` and then selects an output stream. The record contains an a, so LOCATE writes the record to its primary output stream. LOCATE has not consumed the < stage's output, so < still cannot run. The LOCATE stage is now blocked until its output record is consumed.

13. JOIN resumes running. It finds a record available on its input stream. JOIN combines the record with the `a1` record, which it saved, and writes the combined record `a1a3` to its output stream. JOIN has not consumed the LOCATE stage's output, so LOCATE still cannot run. The JOIN stage is now blocked until its output record is consumed.

14. FANINANY resumes. It finds a record available on its primary input stream. FANINANY looks at the record and copies it to its output stream. FANINANY has not consumed the JOIN stage's output, so JOIN still cannot run. The FANINANY stage is now blocked until its output record is consumed.

15. > resumes and finds the `a1a3` record. > writes the record to the file and then to its output stream. > consumes the `a1a3` record from its input stream making FANINANY eligible to run. > then looks for another input record. > cannot run until another record is ready for it.

16. FANINANY resumes, consumes its input, and looks for another input record. FANINANY cannot run until another record is ready for it.

17. JOIN resumes, consumes its input, and looks for another input record. JOIN cannot run until another record is ready for it.

18. LOCATE resumes, consumes its input, and looks for another input record. LOCATE cannot run until another record is ready for it.

At this point, the following records have been written to the output file:

```
b2
a1a3
```

The process continues for the `b4` record until the output file contains:

```
b2
a1a3
b4
```

Then the following happens:

1. < attempts to read a record and finds that there are no more records in the file. < ends causing its output stream to be disconnected. Because this stream is disconnected, the LOCATE stage knows the end of the file is reached and the stage becomes eligible to run.

2. LOCATE determines that its input stream is disconnected. LOCATE ends causing its output streams to be disconnected. Because these streams are disconnected, the JOIN stage knows the end of the file is reached and the stage becomes eligible to run.

3. JOIN determines that its input stream is disconnected. JOIN ends causing its output stream to be disconnected. Because this stream is disconnected, the FANINANY stage knows the end of the file is reached and the stage becomes eligible to run.

4. FANINANY determines that its input streams are disconnected. FANINANY ends causing its output stream to be disconnected. Because this stream is disconnected, the > stage knows the end of the file is reached and the stage becomes eligible to run.

5. > determines that its input stream is disconnected and ends.

Now the entire set of pipelines has completed processing and the PIPE command ends.

Pipeline Subcommands for Multistream Pipelines

PI

This section describes several pipeline subcommands you can use when writing stages that process multiple streams:

- SELECT—selects the input and output stream to be processed by subsequent pipeline subcommands.
- MAXSTREAM—returns the number of the highest stream defined in the PIPE command.
- STREAMNUM—validates a stream number.
- ADDPIPE—adds a pipeline to the set of running pipelines.
- SEVER—disconnects the current stream.

CALLPIPE is also revisited.

On the subcommands, you can identify input and output streams by keywords and in some instances by stream numbers or names.

SELECT Pipeline Subcommand

READTO, OUTPUT, and PEEKTO subcommands act on the currently selected stream. Until now, we haven't been selecting streams. By default, CMS Pipelines has been using stream 0 (the primary input and output stream) for READTOs, OUTPUTs, and PEEKTOs. To select a different stream to be used on following pipeline subcommands, use the SELECT pipeline subcommand.

SELECT selects the stream identified by a keyword (INPUT, OUTPUT, or BOTH) followed by the stream number or name. Figure 198 on page 142 shows how to use SELECT. It shows a stage that reads records from its primary input stream and writes those records to a primary and a secondary output stream (in a manner similar to FANOUT).

```
/* MYFANOUT REXX -- A simplified FANOUT to show SELECT         */
signal on error

'select input 0'  /* Select primary input stream, which is default */

do forever
   'peekto record'      /* Copy a record from input stream 0       */
   'select output 0'    /* Select output stream 0...               */
   'output' record      /* ...write the record to output stream 0  */
   'select output 1'    /* Select output stream 1...               */
   'output' record      /* ...write the record to output stream 1  */
   'readto record'      /* Read a record from input stream 0       */
end

error:
if rc=12 then rc=0
exit rc
```

Figure 198. Example of the SELECT Pipeline Subcommand: MYFANOUT REXX

The `select input` subcommand in MYFANOUT REXX is not really needed because, by default, the primary input stream (stream 0) is selected. We put it in the program so you could see how to write one. To select a secondary input, specify the number 1.

To test MYFANOUT REXX, write an exec that uses both output streams:

```
/* MYTEST EXEC -- Exec to test MYFANOUT REXX                        */
'pipe (endchar ?)',
  'literal Test data',             /* Create record                */
  '| a: myfanout',                 /* Feed it to our stage          */
  '| specs /Output Stream 0/ 1',   /* Put identifier in output record */
     '1-* nextword',               /* Put data in output record     */
  '| f: faninany',                 /* Combine streams               */
  '| console',                     /* Display results               */
  '?',
  'a:',
  '| specs /Output Stream 1/ 1',   /* Put identifier in output record */
     '1-* nextword',               /* Put data in output record     */
  '| f:'
exit rc
```

The following is an example run of MYTEST EXEC:

```
mytest
Output Stream 0 Test data
Output Stream 1 Test data
Ready;
```

What happens if you don't use both output streams? When MYFANOUT REXX tries to select output stream 1, the SELECT pipeline subcommand gives a return code of 4. This SELECT return code indicates that the selected stream is not defined. The SIGNAL instruction causes control to pass to the label `error`, and MYFANOUT ends with a return code of 4.

MAXSTREAM Pipeline Subcommand

MAXSTREAM returns the number of the highest stream defined. The number is given in the return code from MAXSTREAM. The number is the highest number allowed in a SELECT command for the input or output stream (specify INPUT or OUTPUT as an operand).

Figure 199 shows an improved MYFANOUT REXX. MAXSTREAM is used to detect the number of defined output streams. A record is written to each of these streams.

```
/* MYFANOUT REXX -- Write record to all defined output streams   */

'maxstream output'   /* Get number of defined output streams     */
outcount=rc          /* Save that number                         */

signal on error      /* Now intercept nonzero return codes       */

'select input 0'  /* Select primary input stream, which is default */

do forever
   'readto record'       /* Read a record from input stream 0     */
   do i=0 to outcount    /* Write a record to each output stream  */
      'select output' i  /* Select output stream ...              */
      'output' record    /* ...write the record to it             */
   end
end

error:
if rc=12 then rc=0
exit rc
```

Figure 199. Example of the MAXSTREAM Pipeline Subcommand: MYFANOUT REXX

MAXSTREAM gives you the number of the highest stream defined, but it does not indicate whether the streams are connected. In the above example, OUTPUT gives a return code of 12 if the selected stream is disconnected.

To test the improved MYFANOUT REXX, use an exec like this one:

```
/* MYTEST1 EXEC -- Test the improved MYFANOUT */
'pipe (endchar ?)',
   'literal Test',
   '| a: myfanout',
   '| specs /Output stream 0/ 1',
      '1-* nextword',
   '| f: faninany',
   '| console',
   '?',
   'a:',
   '| specs /Output stream 1/ 1',
      '1-* nextword',
   '| f:',
   '?',
   'a:',
   '| specs /Output stream 2/ 1',
      '1-* nextword',
   '| f:',
   '?',
   'a:',
   '| specs /Output stream 3/ 1',
      '1-* nextword',
   '| f:'
exit rc
```

Here is an example run of MYTEST1 EXEC:

```
mytest1
Output stream 0 Test
Output stream 1 Test
Output stream 2 Test
Output stream 3 Test
Ready;
```

STREAMNUM Pipeline Subcommand

The STREAMNUM pipeline subcommand tests whether a given stream is defined. If the stream is defined, the stream number is given in the return code. Otherwise, STREAMNUM gives a negative return code. For operands, write a keyword indicating an input or output stream (INPUT or OUTPUT) followed by the stream number, stream name, or an asterisk (*). An asterisk means the currently selected stream.

This command is handy to validate a stream identifier or to determine which stream is currently selected.

CALLPIPE, Revisited

You can write subroutine pipelines that use multiple input and output streams. To do so, specify the ENDCHAR option on the CALLPIPE command and separate the pipelines with end characters. (CALLPIPE has the same options as the PIPE command.)

When using multiple input streams, we need to tell CALLPIPE which one to use. Use the full format for connectors:

```
*.input.0:     <-- Identifies primary input stream 0
*.output.0:    <-- Identifies primary output stream 0
```

Figure 200 shows LOCDEPT REXX. LOCDEPT expects personnel records as input. It finds all records beginning with the names of department members and writes those records to its primary output stream. Records that do not begin with names of department members are written to the secondary output stream.

```
/* LOCDEPT REXX -- Locate personnel records for department members   */
'callpipe (endchar ?)',
   '*.input.0:',              /* Connect primary input stream         */
   '| a: find Smith_'||,      /* Look for Smith                       */
   '| d: faninany',           /* Combine all department members       */
   '| *.output.0:',           /* Write department records to output 0 */
   '?',
   'a:',
   '| b: find Jones_'||,      /* Look for Jones                       */
   '| d:',                    /* To FANINANY                          */
   '?',
   'b:',
   '| c: find Davis_'||,      /* Look for Davis                       */
   '| d:',                    /* To FANINANY                          */
   '?',
   'c:',
   '| *.output.1:'            /* Rejects to output stream 1           */
exit rc
```

Figure 200. Example of Multistream Subroutine Pipeline: LOCDEPT REXX

The following is an example run. Records of department members are prefixed with an asterisk (*).

```
pipe < salary data | console
Miles          25000
Smith          36500
Jones          22000
Davis          44199
Bush           32072
Rogers         16054
Thomas         18098
Ready;
pipe (end ?) < salary data| a:locdept| specs /*/ 1 1-* 2| console ? a:| console
Miles          25000
*Smith            36500
*Jones            22000
*Davis            44199
Bush           32072
Rogers         16054
Thomas         18098
Ready;
```

ADDPIPE Pipeline Subcommand

The ADDPIPE subcommand adds a pipeline to the set of executing pipelines. At first glance, this seems similar to what CALLPIPE does, but there are two important differences between ADDPIPE and CALLPIPE.

The first important difference is that the stage issuing ADDPIPE continues to run in parallel with the new pipeline. When CALLPIPE is used, on the other hand, the stage issuing the CALLPIPE waits until the CALLPIPE subcommand ends.

The second important difference is that ADDPIPE expands the kinds of connections you can make between the new pipeline and the existing pipeline. CALLPIPE lets you specify a connector at the beginning or the end of the pipeline. One or both connectors can be omitted. ADDPIPE supports these connections and many others.

By using the connectors effectively, you can have your stage remap the surrounding pipeline. This lets you do work in pipelines that you would otherwise have to do with REXX instructions.

ADDPIPE Format

ADDPIPE accepts a pipeline as an operand. There isn't anything special about ADDPIPE that prevents you from using other pipeline subcommands (such as READTO, OUTPUT, or CALLPIPE) in the same stage. The new pipeline can be joined to the pipeline that called your stage by using connectors.

Connectors can be used at either end or both ends, or they can be omitted. Unlike CALLPIPE, an input or an output connector can be specified at either end of the pipeline or both ends. As always, though, connectors cannot be in the middle of the pipeline. To avoid confusion, specify the full format of the connector (*.input: and *.output:). A stream number can also be specified (for example, *.input.0:), but it is often omitted. As always, the stream number defaults to 0. Because input and output connectors can be used at either end of the pipeline, there are nine possible combinations:

1. ```addpipe   b | c```

2. ```addpipe   b | c | *.input:```

3. ```addpipe *.output: | b | c```

4. ```addpipe *.input: | b | c```

5. ```addpipe   b | c | *.output:```

6. ```addpipe *.input: | b | c | *.input:```

```
7.  addpipe *.output: | b | c | *.output:
8.  addpipe *.input:  | b | c | *.output:
9.  addpipe *.output: | b | c | *.input:
```

The spacing in the above examples is for clarity. Fictitious stage names are used to keep the examples simple and to focus your attention on the general abilities that ADDPIPE provides rather than on specific problems.

Each connection possibility lets you redraw the surrounding map in a different way. In the following sections, we discuss each of these connection variations. Some connection variations are more useful than others. We'll point them out along the way.

Another thing to remember is that in some variations the original connection (that is, the original pipeline map) can be restored by executing a SEVER pipeline subcommand. (SEVER is described in "SEVER Pipeline Subcommand" on page 153.) In some cases the original connections cannot be restored. We'll explain why later in the following section.

ADDPIPE Connections

In our discussion of the connection variations, we'll be referring to the PIPE command shown in Figure 201. That command consists of three fictitious stages: A, Z, and D. Stage Z is actually Z REXX. It is the user-written stage from which we'll be executing ADDPIPE pipeline subcommands. The other maps in this section show how the original map (Figure 201) is changed when the ADDPIPE subcommand is executed.

```
pipe A | Z | D
```


Figure 201. The Original Pipeline

Variation 1:   The first variation is an ADDPIPE subcommand that does not have connectors (see Figure 202). The ADDPIPE subcommand is issued from the Z stage.

```
addpipe B | C
```


Figure 202. ADDPIPE Map: B | C

In this case, the stages in the ADDPIPE subcommand operate independently of the original pipeline. Stages B and C are dispatched along with A, Z, and D. The order in which the stages are dispatched is not predictable. The stage that issued the

ADDPIPE command (stage Z) could be dispatched before the stages it added with ADDPIPE (stages B and C). In fact, there is no way for stage Z to tell when stages B and C have finished.

The return code from ADDPIPE indicates whether the stages have been added successfully to the set of running stages. Nonzero return codes indicate syntax errors in the ADDPIPE subcommand itself. They do not indicate whether the stages ran successfully. (Remember, control may return to your stage before the added stages are finished.)

If an added stage ends with a nonzero return code, the return code is reported by the original PIPE command that called your stage.

Figure 203 shows an example of this ADDPIPE variation. The stage expects file identifiers in its input records. It uses the file name in an ADDPIPE command to make an uppercase copy of the file. Because records are likely to flow through the BACKUP stage faster than the files can be copied, several copy operations may be active at the same time.

```
/* BACKUP REXX -- Make copies of the files named in the input records */
signal on error

do forever
   'readto record'              /* Read a record containing a file ID */
   parse var record fn . fm .   /* Parse the file ID                  */
   'addpipe',                   /* Add a pipeline...                  */
     '<' record,                /* ...read the file                   */
     '| xlate upper',           /* ...translate to uppercase          */
     '| >' fn 'backup' fm       /* ...write it to a BACKUP file       */
   'output' record              /* Write file ID to output stream     */
end

error:
if rc=12 then rc=0
exit rc
```

Figure 203. ADDPIPE Example: BACKUP REXX

In the following PIPE command, BACKUP is used to make uppercase copies of all SCRIPT files on file mode A. The names of the files are displayed by CONSOLE.

```
pipe cms listfile * script a | backup | console
DIRMAIL  SCRIPT   A1
DMSC5XMP SCRIPT   A1
EMPLOYEE SCRIPT   A1
FRUITS   SCRIPT   A1
LEGUMES  SCRIPT   A1
LOWER    SCRIPT   A1
MYBOOK   SCRIPT   A1
RECORDS  SCRIPT   A1
SAMPLE   SCRIPT   A1
SNIP     SCRIPT   A1
VMLETTER SCRIPT   A1
Ready;
```

Variation 2: The next connection variation is shown in Figure 204 on page 148. The original pipeline is altered such that the output from stage C is connected to the input to stage Z.

```
addpipe B │ C │ *.INPUT:
```
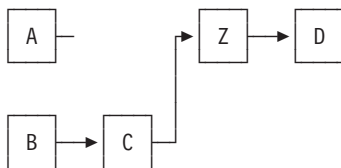

*Figure 204. ADDPIPE Map: B | C | *.INPUT:*

Figure 205 shows an example of this variation. SECPARM REXX writes a record containing a security notice to the output stream. Then it copies all other records in its original input stream to its output stream. SECPARM REXX reads a parameter file that contains one record to determine what security record should be written. While there are several ways to do this, ADDPIPE is used.

```
/* SECPARM REXX -- Add header comment with security notice          */

/* Add a pipeline to read parameter file */
'addpipe < parm data │ xlate upper │ *.input.0:'

/* Process the record in the parameter file                         */
'readto record'        /* Read the record from the ADDPIPE stages   */

select
when pos('CONFIDENTIAL',record)>0 then
   'output /* This exec is company confidential */'
otherwise;
   'output /* This exec is unclassified */'
end

'sever input'  /* Now sever ADDPIPE connection and restore previous */
'short'        /* Copy records from original connection to output   */
exit rc
```

Figure 205. ADDPIPE Example: SECPARM REXX

The stages added by ADDPIPE read the file PARM DATA, translate the record to uppercase, and then feed the record to SECPARM's input stream. An appropriate security classification is selected and OUTPUT writes the security record to SECPARM's output stream.

How is it possible for SECPARM to read the record written by XLATE? Remember that the stages added by ADDPIPE run concurrently with SECPARM itself. ADDPIPE alters the map of the pipeline, so the stages are dispatched just as stages are dispatched in any other pipeline.

For instance, assume that SECPARM is dispatched before the stages added by ADDPIPE. When SECPARM executes a READTO subcommand, the dispatcher gets control. The map of the pipeline has already been changed by ADDPIPE. So, CMS Pipelines knows that the output stream of XLATE UPPER is connected to the input stream of SECPARM. CMS Pipelines also knows it must dispatch the stages added by ADDPIPE to satisfy the READTO. This is no different than the dispatching discussed in "How a Pipeline Runs" on page 78.

It's important to remember that ADDPIPE changes the map of the pipeline when it is executed. When reading execs containing ADDPIPE subcommands, it helps to draw the map of the pipeline. Use the generic maps in this section as a guide.

After the parameter record has been read, we want to copy all of the records in the *original* input stream to SECPARM to the output stream. A SHORT subcommand will do it, but first we have to reconnect the original input stream. (We want to revert back to the original pipeline as depicted in Figure 201 on page 146.)

CMS Pipelines does not automatically restore the original connections when the stages added by ADDPIPE end. To restore the original input stream, we execute a SEVER INPUT subcommand.

The SEVER INPUT subcommand disconnects the current input stream. The current input stream happens to be the output from the XLATE stage that ADDPIPE added. CMS Pipelines remembers any previous connections. It uses a *stack* in case several ADDPIPE commands were executed. When a connection is severed, CMS Pipelines restores the next one on the stack (if any). In the example, the only input stream connection on the stack is the original connection. CMS Pipelines restores this connection, and our original map is restored.

Finally, a SHORT subcommand is executed, and SECPARM ends. An example run of SECPARM follows:

```
pipe < parm data | console
CONFIDENTIAL
Ready;
pipe < legumes script | secparm | console
/* This exec is company confidential */
Peas
Bush beans
Pole beans
Lima beans
Ready;
```

Variation 3: The next variation involves an output connector (Figure 206).

```
addpipe *.output: | B | C
```


---
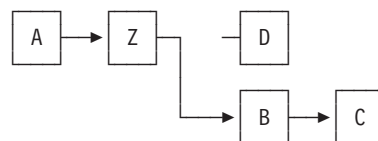
*Figure 206. ADDPIPE Map: *.OUTPUT: | B | C*

As the map shows, the output stream from stage Z is connected to the input stream of stage B. The output is diverted from stage D. Figure 207 on page 150 shows an example named TRACER REXX.

TRACER REXX accepts a string of text as an operand. It appends the string of text, along with the date and time, to a file named TRACER LOG. Then it copies all the records in its input stream to its original output stream (that is, to stage D).

```
/* TRACER REXX -- Write text to TRACER LOG                          */
parse arg text
'addpipe *.output.0: | >> tracer log a' /* Add stages, change connections */
'output' date() time() text            /* Write record to TRACER LOG    */
'sever output'                         /* Restore original connection   */
'short'                                /* Copy records                  */
exit rc
```

Figure 207. ADDPIPE Example: TRACER REXX

The ADDPIPE subcommand connects the primary output stream of TRACER REXX to the primary input stream of the >> stage. When the OUTPUT subcommand is executed, its record becomes the input to the >> stage.

After the tracer record is written, TRACER copies the input records to the original output stream. The SEVER OUTPUT subcommand restores the original output connection (which is the only one in the stack of output connections). The map of the pipeline reverts to the map shown in Figure 201 on page 146. The SHORT subcommand copies the original input stream to the original output stream.

Notice that it was not necessary to execute a SEVER subcommand for the input stream. The original input stream was not changed by ADDPIPE.

The following example shows a PIPE command that uses TRACER and a PIPE command that displays the log.

```
pipe < salary data | tracer Need to verify Smith's salary | console
Miles          25000
Smith          36500
Jones          22000
Davis          44199
Bush           32072
Rogers         16054
Thomas         18098
Ready;
pipe < tracer log | console
20 Jan 1992 16:05:19 Need to verify Smith's salary
Ready;
```

Variation 4: In some connection variations, it is not possible to reconnect to the original input stream. Consider the next variation, which is shown in Figure 208.

```
addpipe *.input: | B | C
```
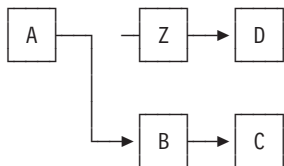

*Figure 208. ADDPIPE Map: *.INPUT: | B | C*

In this variation, A's output stream is diverted to stages B and C. Once the new connection is established, you cannot sever it and return to the old connection. The stages B and C run concurrently with your stage (Z). If you were able to sever the connection after ADDPIPE executes, you would have no way of knowing how many

records stages B and C processed, if any. You wouldn't even know if the stages still existed. So, severing the input stream in this variation is not permitted. Because Z cannot process its original input stream, this connection variation is one of the least useful.

We've already covered most of the ADDPIPE concepts. The remaining variations truly are variations on the same concepts.

Variation 5: The connection variation in Figure 209 has a problem similar to one in the previous variation. The ADDPIPE subcommand connects the output stream of stage C to the input stream of stage D. The output stream of stage Z is disconnected.

```
addpipe B | C | *.output:
```
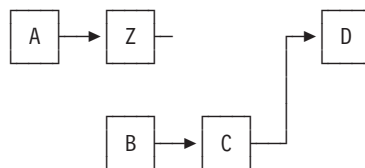

*Figure 209. ADDPIPE Map: B | C | *.OUTPUT:*

In this variation, the output connection established by ADDPIPE cannot be severed. Stage Z does not know how many records stage C has written, and Z cannot predict when it will be dispatched. If Z could sever the output connection, the processing status of C and D would be unpredictable. Consequently, a sever is not allowed.

Variation 6: This variation is far more useful (Figure 210) than the previous two. It lets you insert stages before your stage.

```
addpipe *.input: | B | C | *.input:
```
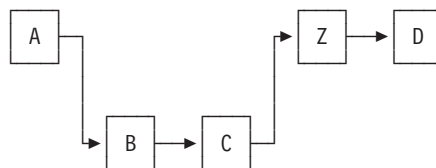

*Figure 210. ADDPIPE Map: *.INPUT: | B | C | *.INPUT:*

By using ADDPIPE to insert stages, you can do more work in CMS Pipelines instead of in REXX. For example, the following code fragment selects records containing the string `NY` and sorts them. Then the preprocessed records are read by READTO:

```
⋮
'addpipe *.input: | locate /NY/ | sort | *.input:'
do forever
   'readto record'
   /* Add REXX instructions here for tasks that cannot be done by */
   /* CMS Pipelines                                               */
   'output' record
```

```
end
```
⋮

Severing the input connection is not allowed. If a sever were allowed, it would not be possible to predict how many records would be preprocessed before the SEVER subcommand is executed.

Variation 7: Another useful variation is shown in Figure 211. It lets you insert stages after your stage.

```
addpipe *.output: | B | C | *.output:
```
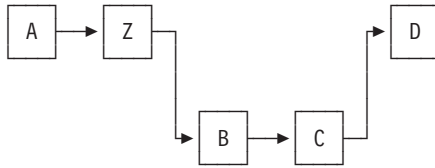

*Figure 211. ADDPIPE Map: *.OUTPUT: | B | C | *.OUTPUT:*

The stages process the records that your stage writes to its output stream. Even though the stages act as a post-processor, execute the ADDPIPE subcommand *before* you write records you want to post-process. Otherwise, the records you write before the ADDPIPE command will be processed by stage D (as defined in the original map):
⋮
```
'output' record   /* Write a record to be processed by stage D */
'addpipe *.output: | xlate upper | *.output:'  /* Change map   */
'output' record   /* Write a record to be processed by XLATE   */
```
⋮

The connection made in this variation cannot be severed. (You would not be able to tell how many records were processed by the ADDPIPE stages.)

Variation 8: The connections in Figure 212 are similar to those used in CALLPIPE when both connectors are specified. In effect, you substitute the ADDPIPE stages for stage Z. With CALLPIPE, however, the original connections are automatically restored in some cases. Because ADDPIPE stages run concurrently, the original connections cannot be restored. You could process some records with READTO and OUTPUT subcommands, and then let stages B and C process the remainder.

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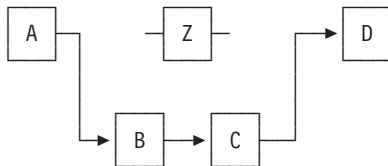

*Figure 212. ADDPIPE Map: *.INPUT: | B | C | *.OUTPUT:*

This variation provides another way for us to do a SHORT:

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

Variation 9: The last connection variation is most unusual. The output stream of stage Z is connected to the input stream of stage B. The output of C is connected to the input of Z. We have created a loop—a stall is possible. You can sever the connections made by this ADDPIPE (before a stall, of course).

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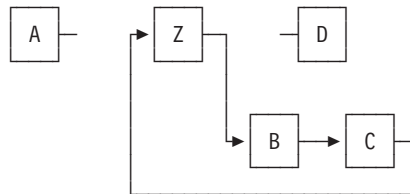

*Figure 213. ADDPIPE Map: *.OUTPUT: | B | C | *.INPUT:*

SEVER Pipeline Subcommand

Use the SEVER subcommand to disconnect the currently selected stream. If the disconnected stream was connected to another stage previously, the connection is restored. SEVER requires one operand (INPUT or OUTPUT) that identifies the stream you want to sever.

SEVER is often used after ADDPIPE. You can use it to restore previously connected streams after you have processed some records on the stream connected with ADDPIPE.

SEVER can also be used to sever a stream after you are done processing it. Some selection stages, for instance TAKE, use SEVER when they switch from the primary output to the secondary output. This reduces the probability of a stall.

For example, suppose your stage is done processing input records. There may be more records in the input stream, but you do not want to process them. Instead of issuing a SEVER, your stage continues with other processing. Because you haven't severed the connection, the stage that was supplying records on your input stream waits indefinitely the next time it writes a record. You have no intention of reading the record, but CMS Pipelines doesn't know that. So, the stage waits, increasing the probability of a stall.

When you issue a SEVER, however, CMS Pipelines knows you no longer intend to read records on that stream. It can give a return code of 12 to any stage that was waiting for you to read its output. By freeing the stage behind you, you may free a stage ahead of you, and in a roundabout way, save your own stage from stalling.

When control returns from your stage, CMS Pipelines severs all streams still connected to that stage.

PI **end**

Chapter 7. Event-Driven Pipelines

In CMS Pipelines, an *event-driven* pipeline is one that contains one or more stages that wait for an event. Regular pipelines, on the other hand, run without waiting for an event to occur. They process to completion as fast as they can.

Event-driven pipelines aren't as complicated as they sound. Your home probably contains several *event-driven* appliances. Perhaps you have a coffee machine that brews coffee automatically in the morning. Or, perhaps you have a telephone answering machine that answers the phone when it rings. Both of these appliances are event-driven. One waits for a certain time of day, while the other waits for an incoming call.

Although you can't get CMS Pipelines to brew a good cup of coffee, you can have it start long-running jobs overnight. Or, you can have it respond to messages when you are away from your desk. You can also use it to set up *service virtual machines*.

Service virtual machines are virtual machines that provide resources to several users. For instance, a service virtual machine might accept requests for data, look up the data, and send it back to the requesting user. Service virtual machines typically run with the console disconnected.

Stages for Event-Driven Pipelines

This section describes three stages that pertain to event-driven pipelines. All are considered device drivers because they write data to the pipeline (although not immediately). The three stages are:

- DELAY

  This stage lets you do things at a certain time or after some time has passed. DELAY is similar to the timer on the coffee machine.

- IMMCMD

  This stage lets you enter commands to a pipeline that is waiting for an event to occur. These commands are known as *immediate commands*. They are like other immediate commands, as described in the *z/VM: CMS Commands and Utilities Reference*. IMMCMD lets you control the pipeline even though it is waiting to do something else. For instance, IMMCMD can let you stop a long-running pipeline or change some aspect of its processing. IMMCMD is similar to the controls provided on your appliances. Even though you have your answering machine set to respond to calls, you can still override it and pick up a call if you wish.

- STARMSG

  This stage lets you do things in response to incoming messages. The pipeline could, for example, record the message in a CMS file and respond to the user that you are away from your desk. A more complex pipeline, as might be found in a service virtual machine, could analyze the message and send data (such as a file) back to the user.

DELAY Stage

DELAY reads a record from its input stream, waits for some event to occur, and then copies that record to its output stream. DELAY repeats this process for every input record, processing them one at a time.

The events for which DELAY waits are time-based. DELAY can be told to wait for a particular time of day or for some time interval to elapse. DELAY determines how long to wait by looking at the first blank-delimited word of each input record. That word should specify the time that DELAY is to wait before copying the record to its output.

Each input record can specify a different delay. The input records can also contain other data after the delay information. Although DELAY looks at only the first blank-delimited word when determining the delay, it copies the entire input record to its output record when the time has elapsed.

Let's look at an example of DELAY. To start an exec named COFFEE EXEC at 6:30 A.M., you could enter:

```
pipe literal 6:30 | delay | specs /exec coffee/ 1 | cms
#cp disc
```

LITERAL writes a record containing `6:30` to its output stream. DELAY reads the record and starts waiting for the next 6:30 A.M. (whether it is today or tomorrow). DELAY expects 24-hour clock notation as input:
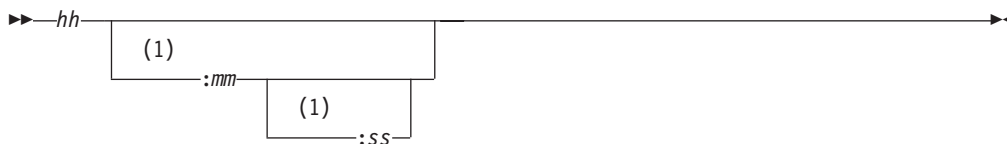
Notes:

1   No blanks are allowed in this position.

Meanwhile, you enter a CP DISCONNECT command and go home. (Enter `#cp disc` or press the PA1 key and then enter `disc`.)

At 6:30 A.M., DELAY copies that one input record to its output stream. SPECS reads this record and writes an output record containing `exec coffee`—SPECS does not use the data in the input record. The CMS stage reads the record, which contains `exec coffee`, and passes it to CMS. CMS runs the COFFEE exec and the pipeline ends.

An exec to do the same thing is shown in Figure 214. It accepts two operands: the time of day at which to run the command, and the command to be run. A stage is added that appends the output of the command to a log file.

```
/* LATER EXEC                                               */
parse arg time command
'pipe',
   'literal' time,          /* Put the time in the pipeline       */
   '| delay',               /* Wait for the specified time        */
   '| specs /'command'/ 1', /* Write output record containing command */
   '| cms',                 /* Pass it to CMS                     */
   '| >> LATER LOG A'       /* Log any console output             */
```

Figure 214. LATER EXEC: A DELAY Example

Another use for DELAY is to run a command at regular intervals. The following example shows how to run the COFFEE exec once every hour:

```
pipe literal +1:00:00 |duplicate *|delay|specs /exec coffee/ 1|cms
#cp disc
```

LITERAL writes a record containing the string `+1:00:00` to its output stream. The plus (+) indicates an interval instead of a time of day. (If you forget the plus, the record is delayed until 1:00 A.M.) To indicate minutes and seconds, use +mm:ss. To indicate seconds, just type a number without a colon (for a 100 second delay: LITERAL +100).

DUPLICATE * generates an infinite number of those records, but only one at a time. (DUPLICATE generates a second record only after the DELAY stage reads the first record.) DELAY reads the record written by DUPLICATE. DELAY understands +1:00:00 to mean wait one hour.

After an hour elapses, DELAY writes the record to its output stream. SPECS writes a record containing `exec coffee` and CMS executes it. DELAY starts timing the next interval when the stage after it processes the output record. COFFEE is issued less frequently than once an hour if it takes an appreciable time to process the response. You might adjust the delay if the processing always takes the same time.

It's possible to generate a command and run it immediately. Figure 215 shows this.

```
/* LATER2 EXEC                                                      */
'pipe',
  'literal +1:00:00',         /* Put one hour interval in the pipeline   */
  '|duplicate *',             /* Copy its input record                   */
  '|delay',                   /* Wait for the specified amount of time    */
  '|literal Go.',             /* Write output record containing Go.       */
  '|specs /exec coffee/ 1',   /* Write output record containing 'exec coffee' */
  '|cms',                     /* Pass it to CMS                          */
  '|console'                  /* Display output on the terminal           */
```

Figure 215. LATER2 EXEC: A DELAY Example

Even though DELAY waits, other stages run as fast as they can until they need a record that DELAY must supply. To run a command immediately, put a LITERAL stage between DELAY and SPECS. Because LITERAL writes its operand to its output stream *before* it reads its input, LITERAL is able to write its operand immediately. Then LITERAL tries to read a record from its input stream and waits because the record is delayed. The stages following LITERAL immediately process the first record written by LITERAL and then they wait for the next record from DELAY. Here's an example run of LATER2:

```
later2
Coffee started...
#cp disc
```

The LITERAL stage writes `Go.` to its output stream. SPECS reads the record and writes a record containing the string `exec coffee` to its output stream. CMS reads the record written by SPECS and executes `exec coffee`. COFFEE is a simple exec that displays the message `Coffee started`:

```
/* z/VM Coffee Support */
say 'Coffee started...'
exit
```

To stop a long-running PIPE command, enter the PIPMOD STOP command:

```
pipe literal +1:00:00 | delay | console
pipmod stop
Ready;
```

Figure 216 shows another delay program. This program, named EVERY REXX, contains a subroutine pipeline. EVERY accepts two arguments: a delay (without a plus sign) and a command. EVERY writes the command line to the pipeline after the specified delay. It uses DUPLICATE * to create an infinite number of these commands, one at a time.

```
/* EVERY REXX -- Write a line after a delay */
signal on novalue
parse arg holdup cmd
if ¬abbrev('IMMEDIATE',translate(holdup),3)/* One right now?   */
   then istring='',                        /* No...            */
   else do
      parse arg . holdup cmd               /* Yes, parse again */
      istring='| literal go.'
   end

'callpipe',
   'literal +'holdup,          /* Make a relative delay      */
   '| duplicate *',            /* As many as needed          */
   '| delay',                  /* Wait                       */
    istring,                   /* Fire one immediately, maybe */
   '| specs x'c2x(cmd) '1',    /* Turn it into a command     */
   '| cms',                    /* Pass it to CMS             */
   '| *:'                      /* Pass to output             */

exit rc

novalue:
   say 'No argument was specified'
   exit
```

Figure 216. EVERY REXX: Example Subroutine Pipeline for Delaying Commands

The SPECS stage shows how to write any string without worrying about the stage separator: convert it to hexadecimal.

The first blank-delimited word on an input line specifies either the time in hours, minutes, and seconds, or IMMediate.

Figure 217 shows another example exec that lets you delay commands.

```
/* DOIT EXEC has the following formats:                           */
/*                                                                */
/* DOIT (IMMediate) < AT hh:mm                        >  command  */
/*                  < IN n    <Hours|Minutes|Seconds> >           */
/*                  < EVERY n <Hours|Minutes|Seconds> >           */
parse upper arg token duration line

/* Run command immediately ? */
if abbrev('IMMEDIATE',token,3) then do
   parse upper arg . token duration line /* Re-parse if IMM is specified  */
   istring='| literal go.'            /* Yes, build a pipeline segment */
end
else istring=''

dup=0
select
when token='AT' then do
   cmd=line
   duration=duration||':00'
   end
when token='EVERY' | token='IN' then do
   parse var line units cmd
   if token='EVERY' then dup='*'
   select
   when abbrev('HOURS',units,1)   then duration='+'||duration||':00:00'
   when abbrev('MINUTES',units,1) then duration='+'||duration||':00'
   when abbrev('SECONDS',units,1) then duration='+'||duration
   otherwise do
      say 'Must specify Hours, Minutes, or Seconds'
      exit 2
      end
   end /* select */
   end
otherwise do
   say 'Must specify AT, IN, or EVERY'
   exit 1
   end
end /* select */
'pipe literal 'duration,
   ' | duplicate 'dup,
   ' | delay ',
     istring,
   ' | specs /'cmd'/ 1 ',
   ' | cms'
exit rc
```

Figure 217. DOIT EXEC: Example Exec for Delaying Commands

To use DOIT to execute a CMS TELL command in 10 seconds, you would enter:

```
doit in 10 seconds tell exec * hi.
```

Or, to start COFFEE immediately and at 6:00 A.M., enter:

```
doit imm at 6:00 coffee
```

To run COFFEE every hour, enter:

```
doit every 1 hour coffee
```

To end a long-running pipeline, like the one started by DOIT when EVERY is specified, enter:

```
pipmod stop
```

PIPMOD STOP is a command that immediately terminates the PIPE command.

IMMCMD Stage

IMMCMD lets you set up *immediate commands* to control long-running pipelines. An immediate command is a command that takes priority over some long-running process. The system interrupts the long-running process temporarily and executes the immediate command.

One use of immediate commands is to let the user execute CMS commands while the pipeline is running. Another useful immediate command would provide an alternative way to end long running pipelines. You might, for example, define an immediate command named STOP that causes the PIPE command to end. Examples of both are in this section.

Note: Because of the way screens are refreshed, it is recommended that you run the examples in this section in line mode instead of full-screen CMS. Enter the CMS command QUERY FULLSCREEN to see if full-screen CMS is on. If it is, enter SET FULLSCREEN OFF to return to line-mode operation.

To define an immediate command, specify the name of the command as an argument on the IMMCMD stage. IMMCMD does two things. It sets up an immediate command with the name you specify as an operand. (The immediate command is just like any other CMS immediate command.) Then IMMCMD waits for you to type the operand as an immediate command. When you enter that immediate command, IMMCMD writes a record to its output stream. The record consists of any arguments specified on the immediate command, but not the name of the immediate command itself. If no operands are supplied on the immediate command, IMMCMD writes a null record to the pipeline.

Figure 218 shows an example of IMMCMD. Try entering a PIPE command like the one shown in the figure. When you enter the command, the IMMCMD stage waits for you to enter `mycom`. In the example, `mycom hello` is entered. IMMCMD writes a record containing `hello` to its output stream, and CONSOLE displays it. To end the pipeline, enter PIPMOD STOP as shown.

```
pipe immcmd mycom | console
mycom hello
hello
pipmod stop
Ready;
```

Figure 218. IMMCMD Stage Example

IMMCMD is often used in PIPE commands that contain other long-running pipelines. Pipelines that define immediate commands should be placed before other long-running pipelines. Otherwise, the long-running pipeline could run first, and the immediate command will not be set up until it is too late.

In Figure 219 on page 161, IMMCMD defines a STOP immediate command in the first pipeline. The second pipeline executes a CP MESSAGE command every 5 seconds. The streams of the two pipelines are not connected.

```
/* RPTMSG EXEC -- Repeat a message every 5 seconds            */
'PIPE  (endchar ?)',
   'immcmd stop',              /* Output here when user types STOP */
   '| specs /PIPMOD STOP/ 1',   /* Write PIPMOD STOP to output     */
   '| command',                 /* Issue it                        */
   '?',
   'literal +5',               /* Write record containing delay   */
   '| duplicate *',             /* Duplicate it                    */
   '| delay',                   /* Delay                           */
   '| specs /message * hello/ 1',  /* Write a CP MESSAGE command     */
   '| cp'                       /* Issue the command               */
exit rc
```

Figure 219. IMMCMD Stage Example: RPTMSG EXEC

When a STOP command is entered, IMMCMD writes a record containing the operands, if any, to its output. If there aren't any operands on the STOP command, IMMCMD writes a null record. Then SPECS writes string PIPMOD STOP to its output stream. Finally, COMMAND executes PIPMOD STOP, ending the PIPE command. Here is an example run of RPTMSG:

```
rptmsg
 15:55:07  * MSG FROM YOURID  : HELLO
 15:55:12  * MSG FROM YOURID  : HELLO
 15:55:17  * MSG FROM YOURID  : HELLO
 15:55:22  * MSG FROM YOURID  : HELLO
stop
Ready;
```

The subroutine in Figure 220 shows how to define more than one immediate command. It defines two immediate commands: STOP and CMS. The CMS immediate command sends commands to CMS for processing. Both of these pipelines are independent of each other and run concurrently. Notice that neither pipeline is connected with the caller's input or output streams—this is perfectly acceptable. Even though the streams are not connected, the PIPE command processes them concurrently.

```
/* ASYNCMS REXX -- Issue asynchronous CMS command or stop pipeline */
'callpipe (endchar ? name ASYNCMS)',
   'immcmd stop',                   /* STOP commands:          */
   '| specs /PIPMOD STOP/ 1',        /* Make the pipeline stop   */
   '| command',                      /* Issue it                 */
   '?',
   'immcmd cms',                    /* CMS commands:           */
   '| subcom cms',                   /* Issue to CMS            */
   '| *:'                           /* Connect to caller        */
exit rc
```

Figure 220. Example Subroutine to Enter Asynchronous Commands: ASYNCMS REXX

With ASYNCMS, you can rewrite RPTMSG EXEC as shown in Figure 221 on page 162.

```
/* RPTMSG1 EXEC -- Improved                                          */
'PIPE  (endchar ?)',
   'asyncms',                    /* Define STOP and CMS commands     */
   '?',
   'literal +5',                 /* Write record containing delay    */
   '| duplicate *',              /* Duplicate it                     */
   '| delay',                    /* Delay                            */
   '| specs /message * hello/ 1',  /* Write a CP MESSAGE command     */
   '| cp'                        /* Issue the command                */
exit rc
```

Figure 221. Example Use of ASYNCMS REXX

To enter a CMS or CP command while the pipeline is running, enter CMS followed by the command you want to execute. For example, to enter a CMS SENDFILE command:

```
cms exec sendfile test data a to denise
```

ASYNCMS passes the command to CMS for processing. (If the command is a CP command, CMS passes it to CP.) In the following example run of RPTMSG1, the user enters a CMS QUERY DISK A command and a STOP command:

```
rptmsg1
 16:09:30  * MSG FROM YOURID  : HELLO
 16:09:35  * MSG FROM YOURID  : HELLO
cms query disk a
LABEL  VDEV M  STAT  CYL TYPE BLKSIZE  FILES  BLKS USED-(%) BLKS LEFT  BLK TOTAL
BAR191 191  A   R/W    3 3380 4096        101        124-28      326        450
 16:09:40  * MSG FROM YOURID  : HELLO
 16:09:45  * MSG FROM YOURID  : HELLO
stop
Ready;
```

STARMSG Stage

The STARMSG stage lets you write lines from the CP Message system service (*MSG) or the Message All system service (*MSGALL). Before using STARMSG, you need to understand the Message system service. It is described in the *z/VM: CP Programming Services* book. To use the Message system service, specify *MSG as the operand to STARMSG. To use the Message All system service, specify *MSGALL as the operand to STARMSG. If you do not specify an operand on STARMSG, it uses the Message system service by default. Before using STARMSG, you must also make sure CMS FULLSCREEN is set to SUSPEND or OFF.

The STARMSG stage sets up a connection to the system service and waits for a message to arrive. When a message arrives, STARMSG writes a record to its output stream. That record contains an 8-byte message class followed by the 8-byte user ID from which the message originated and any message data.

Figure 222 on page 163 shows the record format.

class	user ID	message ...

```
 1      8 9    16  17
```

For example,

```
00000001USERID  Here is the message text
```

Figure 222. Format of STARMSG Output Records

There are simple service virtual machines that process commands sent with the CP SMSG command from users on the same system. More sophisticated servers can service requests forwarded as RSCS messages. Here is an example of the first kind:

```
/* Process SMSG requests                                        */
'CP SET SMSG IUCV'  /* Direct SMSGs to IUCV                      */
'PIPE',
   'starmsg',       /* Trap messages                             */
   '| specs 9-* 1', /* Strip off message class (columns 1 through 8)   */
   '| validate',    /* Verify that user is authorized            */
   '| specs 9-* 1', /* Strip the user ID from the message        */
   '| subcom cms'   /* Pass the message, which should be a command, to CMS */
```

In the example, STARMSG traps only SMSG messages, so the message class prefix is always the same. It is discarded. VALIDATE is a user-written stage (see Figure 189 on page 130). It ensures that only those we trust get service. Use selection filters and multiple streams to process requests from users in particular ways.

STARMSG sets up the immediate command HMSG. Enter HMSG to stop STARMSG. HMSG lets an orderly clean-up occur. PIPMOD STOP will also stop STARMSG, but it stops all stages that are waiting for an external event such as DELAY and IMMCMD.

Let's look at the output records generated by STARMSG. First we issue SET CPCONIO IUCV to direct CP command responses to IUCV. We then issue SET MSG IUCV and SET SMSG IUCV to direct messages to IUCV.

```
set cpconio iucv
Ready;
set msg iucv
Ready;
set smsg iucv
Ready;
pipe starmsg | console
#cp msg * hi
00000001YOURID  HI
#cp query time
00000003YOURID  TIME IS 16:22:48 EST WEDNESDAY 10/30/91
00000003YOURID  CONNECT= 01:31:32 VIRTCPU= 000:08.73 TOTCPU= 000:14.79
#cp smsg * Let's send a special message.
It should appear as class 4
00000004YOURID  LET'S SEND A SPECIAL MESSAGE.  IT SHOULD APPEAR AS CLASS 4
hmsg
Ready;
```

Example File Server

This section shows how to combine IMMCMD and STARMSG to create a file server. A *file server* is a program that manages a collection of files. Users send requests related to those files to the file server, and the file server processes those requests. (z/VM's Shared File System uses the file server concept.) On z/VM, a file server typically runs in a disconnected virtual machine.

Complex file servers let users read files, write files, create locks on files, and so on. Varying communication mechanisms are used between the user machines and the file server machines. The file server we show here is very simple and has limited error handling. It processes only one request, which is to retrieve a file. For communications, the user machine uses the CMS TELL command to send requests to the server, and the server virtual machine uses the CMS SENDFILE command to send files to the user. The file server may also communicate with the user through the CMS TELL command to track errors.

Example Requester

Our design requires users to send a structured message to the server machine when they want a file. Rather than rely on users to enter the proper TELL command, we create an exec, named FGET, to do it. FGET is shown in Figure 223.

```
/* FGET EXEC -- send a file request to the server machine */
parse upper arg fn ft fm .
if fn='' then exit 1
if ft='' then ft='SCRIPT'
if fm='' then fm='A'

'EXEC TELL serverid AT nodeid $FGET$ 'fn ft fm
exit 0
```

Figure 223. FGET EXEC: Example Requester

In the TELL command, substitute the user ID of the server for *serverid*. Substitute the node ID of the server for *nodeid*.

FGET accepts a file name, file type, and a file mode as input. To get the server machine's ALL NOTEBOOK A file, for example, a user would enter:

```
fget all notebook a
```

FGET builds a structured message and executes the TELL command. Notice that the server may be on a different node. Also notice that the message contains the string $FGET$ followed by the desired file identifier. The string $FGET$ is used by the server to distinguish file requests from other messages it may receive.

Example Server

After defining the structure of the message, we need to determine how that message will look when STARMSG intercepts it. We already know that STARMSG writes records to its output stream in the following form:

```
00000001USERID  Message text
```

The first 8-byte field contains the message class. Our structured messages are being sent with the CMS TELL command, so message class 1 is the only class of interest. The second 8-byte field contains the user ID, while the rest of the record contains the message text.

In theory, the messages we get from users should look like this:

```
00000001MELINDA $FGET$ ALL NOTEBOOK A
```

But, we may get messages over the network:

```
00000001RSCS    From VMNODE(JOHN): $FGET$ ALL NOTEBOOK A
```

We may also get messages that have nothing to do with file requests:

```
00000001RSCS    File (7968) spooled to YOURID -- origin VMND5(SMITHA) 08/14/91 1
0:11:52 EDT
00000001RSCS    From VMND2(JONESY): Are you there?
00000001LISA    Please reaccess the tools disk.
```

Furthermore, we may want to limit access to certain user IDs. Messages from unauthorized users will look like any other messages:

```
00000001NOTMYBUD$FGET$ COMPANY SECRETS A
```

We need some way to verify user IDs within the server virtual machine.

We decide that our file server should do the following:

1. Direct messages to IUCV so that STARMSG can get them.

   If we don't direct the messages to IUCV, they will be displayed on the server machine's console. They will not be intercepted by STARMSG.

2. Set up a STOP immediate command and a CMS immediate command (use ASYNCMS, which is shown in Figure 220 on page 161).

   STOP lets us end the PIPE command. CMS lets us direct commands to CMS from the server console while the pipeline is running.

3. Turn on message trapping (STARMSG).

4. Convert messages from local users and from remote users (users having a different node ID) to a standard format.

   Messages we need to process are delivered from STARMSG in two different formats. Rather than have two paths through subsequent code, we'll convert the messages to a standard format.

5. Filter out any message that does not start with the string $FGET$.

6. Process the requests embodied in these standard-format messages.

7. Reroute messages to the console when the pipeline is stopped.

We could do all of these functions in one lengthy pipeline, but a better approach would be to create a stage for each major function. By doing so, we can solve the problem a piece at a time.

Let's begin by writing the main pipeline, which is shown in Figure 224 on page 166.

```
/* MYSERV EXEC -- a simple file server program.             */

'cp set msg iucv'     /* Have CP route messages to IUCV          */
'pipe (endchar ?)',
    'asyncms',        /* Set up STOP command and CMS imm. commands */
    '?',
    'starmsg',        /* Trap incoming messages               */
    '| generic',      /* Convert message to generic form      */
    '| fgetmsg',      /* Filter out unwanted messages         */
    '| request'       /* Handle the request                   */
piperc=rc             /* Save the pipeline return code        */

'cp set msg on'       /* Reroute messages to the console      */
exit piperc           /* Exit with pipeline return code       */
```

Figure 224. Example File Server: MYSERV EXEC

GENERIC, FGETMSG, and REQUEST are all stages yet to be written. During development, it's useful to create do-nothing stages for those remaining to be written. This lets you test the pipeline as you develop it. An example do-nothing stage for FGETMSG might look like this:

```
/* FGETMSG REXX -- to be added later */
'callpipe (name fgetmsg endchar ?)',
   '*:',
   '| *: '
```

Later you can fill in the missing code.

GENERIC REXX—Convert Messages to a Standard Format

There are two formats of messages: those sent by users of your system (local users) and those forwarded to you from the RSCS machine (from remote users). To see these formats on your own system, enter the following command:

```
pipe starmsg | > format data a
```

Then log on to another user ID on your system and send a message to the user ID running the PIPE command. Next, log on to a user ID on another system and send another message. Finally, log on to the user ID running the PIPE command and enter HMSG to stop it. Then examine the file FORMAT DATA A. It should look like this:

```
00000001MIKE1   Hello.
00000001RSCS    From VMNODE(MIKE2): Hello again.
```

MIKE1 is the user ID on the local system, while MIKE2 is the user ID on the remote system. The node ID of the remote system is VMNODE. RSCS is the user ID of the local machine that receives messages from remote systems and passes them on to you.

The form of the message from the remote system may vary depending on which networking products are used at your installation. By sending messages to yourself from remote systems and observing the results in FORMAT DATA, you can deduce the form being used. The example file server assumes remote messages have the format shown above.

GENERIC REXX, shown in Figure 225 on page 167, determines whether the message is local or remote. It then transforms the record to a standard format, which later stages can easily process. The standard format created is:

node_ID user_ID message_text

One or more blanks are between the node ID, user ID, and the message text. The message text starts in column 21. GENERIC gets the node IDs of remote systems from the record itself. However, the node ID of the local system is not on the record. GENERIC uses the CMS IDENTIFY command to determine it.

GENERIC distinguishes local and remote messages by looking at the user ID that sent the message. GENERIC uses the CMS IDENTIFY command to determine the user ID of the RSCS virtual machine.

```
/* GENERIC REXX -- convert the STARMSG record to a generic form      */
address command 'PIPE',
   'cms identify',
   '| var istring'
if rc¬=0 then exit rc
parse var istring . . node . rscs .
'callpipe (name generic endchar ?) ',
   '*:',                            /* Connect input to calling pipeline */
   '| specs 9-* 1',                 /* Drop message class from record    */
   '| a: nlocate 1-8 /'left(rscs,8)'/',   /* Send net msgs to other pipe */
   '| specs /'left(node,8)'/ 1',      /* Put node in generic record... */
      '1-8 nextword',                 /* ...followed by user ID        */
      '9-* 21',                       /* ...and message in column 21   */
   '| f: faninany',                 /* Combine both streams          */
   '| *:',
   '?',
   'a:',                            /* Connect to source of RSCS requests */
   '| specs 9-* 1',                 /* Get rid of the RSCS token      */
   '| specs words2-* 1',            /* Get rid of the keyword FROM    */
   '| specs pad 40',                /* Specify blank as pad character */
      'words1 1.20',                 /* Put first word in first 20 columns */
      'words 2-* next',             /* Put the rest in column 21      */
   '| xlate 1-20 ( 40 ) 40 : 40',      /* Get rid of (, ), and :      */
   '| f:'                           /* Connect back to the first pipeline */
exit rc
```

Figure 225. Example Filter to Create Generic Records: GENERIC REXX

GENERIC uses a SPECS stage to drop the first 8 bytes of the input record. These bytes contain the message class number. We already know the messages are class 1 messages, so we don't need the information. Then NLOCATE is used to determine whether the message is from a local or remote user.

For local messages, SPECS is used to add the node ID of the local system to the record. This node ID was determined by issuing a CMS IDENTIFY command and parsing the response. The node ID is assigned to the variable `node`. The SPECS stage is written so that blanks occur between the node ID and the user ID. SPECS puts the message text starting at column 21, as required for the standard format.

Remote messages are written to the secondary output stream of NLOCATE. The records written by NLOCATE look like this:

```
RSCS    From VMNODE(JOHN): $FGET$ ALL NOTEBOOK A
```

The message class number has already been removed. SPECS is used to remove another 8 bytes from the record, yielding:

```
From VMNODE(JOHN): $FGET$ ALL NOTEBOOK A
```

Those 8 bytes contain the user ID of the networking machine, which we no longer need. Next, we must get rid of the word `From`. We use SPECS with the WORDS operand to do it:

```
   .
   .
   .
   '| specs words2-* 1',          /* Get rid of the keyword FROM        */
   .
   .
   .
```

Words 2 through the end of the record are copied to the output record starting at column 1. The first word (`From`) is not copied. The result of SPECS is:

```
VMNODE(JOHN): $FGET$ ALL NOTEBOOK A
```

The record is close to the format we need. We need to get rid of the parentheses and the colon. The XLATE stage can do it, but XLATE needs a column range on which to operate. So, we use another SPECS to put the first blank-delimited word in the first 20 columns of the record. The PAD operand causes blanks (X'40') to be used to pad the string:

```
   .
   .
   .
   '| specs pad 40',              /* Specify blank as pad character    */
      'words1 1.20',              /* Put first word in first 20 columns */
      'words 2-* next',           /* Put the rest in column 21          */
   '| xlate 1-20 ( 40 ) 40 : 40', /* Get rid of (, ), and :        */
   .
   .
   .
```

The record is now in the correct format:

```
VMNODE JOHN           $FGET$ ALL NOTEBOOK A
```

GENERIC uses FANINANY to pick up the record from whichever route it travelled, and then writes that record back to its output stream.

FGETMSG REXX—Filtering Out Unwanted Messages

FGETMSG REXX is a stage that gets rid of any message that does not start with the string `$FGET$`. Messages starting with the string `$FGET$` were most likely generated by the FGET requester exec. It is possible that `$FGET$` is at the beginning of the message for some other reason, but that risk is accepted to keep the example brief.

As Figure 226 shows, we save all rejected messages in the file MYSERV OTHERMSG.

```
/* FGETMSG REXX -- get rid of messages not starting with $FGET$     */
'callpipe (name fgetmsg endchar ?)',
   '*:',                              /* Message text starts in column 21 */
   '| a: locate 21.6 /$FGET$/',  /* of the generic records.          */
   '| *:',
   '?',
   'a:',                              /* Rejects flow into this pipeline  */
   '| >> myserv othermsg a'      /* Save them in a file...           */
exit rc
```

Figure 226. Example Filter for $FGET$: FGETMSG REXX

REQUEST REXX—Process the Request

REQUEST REXX is called after the records have been transformed and filtered. First it verifies that the user is authorized. Then it verifies the existence of the requested file. If all is well, it builds a CMS SENDFILE command and executes it. If an error is detected, a message is sent to the user via the CMS TELL command.

REQUEST REXX is shown in Figure 227.

```
/* REQUEST REXX -- process the request in the generic record   */

'callpipe (name request endchar ?) ',
   '*:',
   '| chkauth',                   /* Check user's authorization  */
   '| c: find VALID',             /* Was it okay?                */
   '| specs words 2-* 1',         /* Remove indicator from record */
   '| chkfile',                   /* Check existence of file     */
   '| d: find VALID',             /* Did it exist?               */
   '| specs /exec sendfile/ 1',   /* Build a SENDFILE command    */
      'words 5-7 nextword',       /* ...add file identifier      */
      '/to/ nextword',            /* ...add TO keyword           */
      'word3 nextword',           /* ...add user ID              */
      '/at/ nextword',            /* ...add AT keyword           */
      'word2 nextword',           /* ...add node ID              */
   '| cms',                       /* Execute the SENDFILE        */
   '|*:',
   '?',                           /* User not authorized         */
   'c:',                          /* Send error message          */
   '| specs /exec tell/ 1',       /* Build a TELL command        */
      'words3 nextword',          /* ...add user ID              */
      '/at/ nextword',            /* ...add AT keyword           */
      'words2 nextword',          /* ...add node ID              */
      '/* You are not authorized./ nextword',  /* ...add text   */
   '| cms',                       /* Execute the TELL            */
   '?',
   'd:',                          /* File not found              */
   '| specs /exec tell/ 1',       /* Build a TELL  command       */
      'words3 nextword',          /* ...add user ID              */
      '/at/ nextword',            /* ...add AT keyword           */
      'words2 nextword',          /* ...add node ID              */
      '/* File not found./ nextword',   /* ...add text          */
   '| cms'                        /* Execute the TELL            */
exit rc
```

Figure 227. Example Request Processor: REQUEST REXX

REQUEST REXX calls two other stages to check the authorization and to verify that the file exists. These stages, CHKAUTH and CHKFILE, return the record with either the word VALID or INVALID inserted at the beginning of it. VALID/INVALID indicates whether the test was successful.

Most of the work in REQUEST REXX is done with SPECS stages. Data is shuffled about and command strings are built that are passed to the CMS stage for execution. Notice that the WORDS operand on SPECS is used to move tokens. Using column numbers would be too tedious.

The CHKAUTH REXX stage is shown in Figure 228 on page 170. It expects records that adhere to the generic format described earlier. CHKAUTH writes the record back to the pipeline with an extra word added to indicate whether the user is authorized. The word is added to the beginning of the record. VALID is added if the user is authorized. Otherwise INVALID is added.

```
/* CHKAUTH REXX -- Verify that a user is authorized              */
'callpipe (name chkauth endchar ?)',
   '*:',
   '| specs words1 1.8 words2 10.8 words 3-* 19', /* Move data      */
   '| l: lookup 1.17 detail',         /* Check user against list    */
   '| specs /VALID / 1 1-* next',     /* Tack on VALID              */
   '| f: faninany',                   /* Collect records            */
   '| *:',
   '?',
   '< valid users',                   /* Read in list of valid users */
   '| xlate upper',                   /* Translate list to uppercase */
   '| specs words1 1.8 words2 10.8',  /* Move data to specific cols. */
   '| l:',                            /* Connect to LOOKUP secondary */
   '| specs /INVALID / 1 1-* next',   /* Tack on INVALID            */
   '| f:'                             /* Feed back to main pipeline  */
exit rc
```

Figure 228. Verify Authorization: CHKAUTH REXX

When a record enters CHKAUTH, it is immediately changed by SPECS. SPECS moves the user ID and node ID to specific columns, so that they can be compared by LOOKUP. (See page "LOOKUP Stage" on page 127 for a description of LOOKUP.) The LOOKUP stage compares the first 17 columns of the record with its reference list. If the record matches, LOOKUP writes it to its primary output stream. This record is passed to the primary input stream of SPECS, the next stage. It adds the word VALID to the beginning of the record. The record is then combined with INVALID-prefixed records from the secondary output of FANINANY and LOOKUP and leaves the subroutine pipeline (CHKAUTH) to return to the calling stage.

LOOKUP expects a reference list as its secondary input. The second pipeline (beginning with < valid users) provides the records that LOOKUP uses to build the reference. The file VALID USERS is read and the records are translated to uppercase. VALID USERS should contain a list of valid node/user ID pairs. For example:

```
vmnode2 lisa
yournode ted
yournode denise
vmfar1 annette
```

After the records are translated to uppercase, a SPECS stage moves the node and user IDs to specific columns. This is necessary for comparisons with the message records. The label l connects the output stream of SPECS to the secondary input stream of LOOKUP. LOOKUP builds its reference from the records that SPECS supplies.

The label l also connects the secondary output stream of LOOKUP to the last SPECS stage in CHKAUTH REXX. LOOKUP writes records that do not match to its secondary output stream. SPECS adds the word INVALID to the beginning of these records and passes them back to the first pipeline through FANINANY.

Referring back to Figure 227 on page 169, you'll see that REQUEST checks for the VALID/INVALID keyword. If the record is not valid it sends a message to the user. Otherwise, it strips off the VALID keyword and calls CHKFILE.

CHKFILE verifies that the requested file exists. Like CHKAUTH, it expects a standard-format record. It writes the record to its output stream, with a word

prefixed to it indicating whether the file exists.

```
/* CHKFILE REXX -- Verify the existence of a file */

'callpipe (name chkfile endchar ?)',
   '*:',
   '| a: fanout',                     /* Send a copy of the record */
   '| f: faninany',      /* Collect original record & VALID|INVALID */
   '| specs 1-* 10 read 1-9 1',             /* Combine records */
   '| *:',
   '?',
   'a:',
   '| specs 1-* 1 / A A A / next',    /* Protect against bogus msgs */
   '| specs words 4.3 1',            /* Extract the file identifier*/
   '| b: state',                     /* Check existence            */
   '| specs /VALID / 1',             /* Exists: create VALID rec.  */
   '| f:',
   '?',
   'b:',
   '| specs /INVALID / 1',           /* Does not exist...          */
   '| f:'
exit rc
```

Figure 229. CHKFILE REXX: Verify the Existence of a File

The STATE stage is used to check the existence of the file. Unfortunately, the input that STATE needs is the file identifier, *not* the user ID and node identifier, which are also contained in the record. So, to use STATE, we must modify the record. Yet, we need to preserve the original record because that is what CHKFILE must write to its output stream. In other words, we must destroy the record to check it, but we still need to keep the original record.

This problem happens frequently when coding pipelines. There are many instances when you want to save your original record for use later, but must alter it to test it. While each of these cases is different, consider using FANOUT. FANOUT lets you make a copy of the record on a different stream. With some ingenuity, you can combine the streams later in a way that will suit your needs.

Notice that a similar problem occurs in CHKAUTH. In CHKAUTH, however, LOOKUP is used. LOOKUP accepts a range, which saved us from having to destroy the record to test it.

In CHKFILE, the record must be modified for STATE, so we use the FANOUT technique. FANOUT writes a copy of the record to its secondary stream. The stages connected to that stream pull apart the record and test it. The original record continues on to FANINANY, which is also used to collect the result of the test. In our example, the result of the test is a record containing one word: VALID or INVALID.

The SPECS stage following the FANINANY combines the two records, which yields the desired result.

Running the File Server

To run the file server, create a file named VALID USERS that contains the node and user IDs of valid users. Then enter the MYSERV command:

```
myserv
```

MYSERV is ready to process requests. You can disconnect from the server machine by entering:

```
#cp disc
```

Log on to an authorized user ID and use the FGET EXEC to request a file:

```
fget all notebook
```

The file, if it exists, will be sent to you. If the file does not exist, the server will send you a `File not found` message.

To enter a CMS command while the server is running, prefix the command with `cms` as shown in this example:

```
cms query disk a
LABEL  VDEV M  STAT   CYL TYPE BLKSIZE   FILES   BLKS USED-(%) BLKS LEFT  BLK TOTAL
BAR191 191  A    R/W     3 3380 4096         99        121-27         329       450
```

To stop the server, reconnect to the server machine and enter the STOP immediate command:

```
stop
```

CMS will display a `Ready;` message containing the return code from the pipeline. When you stop the server, examine the file MYSERV OTHERMSG to see if the server machine received messages other than those created by FGET.

Chapter 8. Using Unit Record Devices

CMS Pipelines has several stages that work with unit record devices. CMS supports three unit record devices: one virtual reader at address 00C, one virtual punch at address 00D, and one virtual printer at address 00E. Any output that you direct to your virtual printer or punch, or any input you receive from your reader, is controlled by the spooling facilities of the control program (CP). Each output unit is known to CP as a spool file.

CMS Pipelines lets you work with these spool files. You can write (create) spool files to the virtual printer or virtual punch. You can read spool files from the virtual reader. To give you complete control, CMS Pipelines does not issue CP commands to the virtual device. One exception is the READER stage. It issues a CLOSE command when it is done. In all other cases, you must issue SPOOL, TAG, and CLOSE commands as required. Spool files are created by CP when you issue the CLOSE command.

In this chapter, we assume that you have some familiarity with unit record devices and spool files. If you want more information about unit record devices and spool files, refer to *z/VM: General Information* and to the *z/VM: CMS User's Guide* before reading this chapter. We also assume that you are familiar with channel command codes used in device I/O. Channel commands are described in the *IBM ESA/370* Reference Summary*.

Writing to the Virtual Punch (PUNCH, URO)

CMS Pipelines includes two device drivers that write to the virtual punch: PUNCH and URO. The important difference between the two is their handling of channel command codes. CMS Pipelines requires a one-byte channel command code at the beginning of each record in a punch spool file. PUNCH puts the channel command code on the records for you before writing the spool file. URO does not add a channel command code. The records it processes must already have channel command codes.

One thing to remember when using either stage is that the maximum length for a punch record is 80 bytes. CP truncates punch records after column 80 without issuing a message or giving other indication of error.

Let's look at PUNCH first. PUNCH puts a one-byte channel command code at the beginning of each record it reads from its input stream. The channel command code that PUNCH adds is X'41', which indicates that the record is a data record. A X'03' is also allowed for the channel command code. X'03' is a no-operation. X'03' is sometimes used for records that describe the file (such as the file name or the source of the file). If you want to punch records with X'03' channel command codes, you must use URO instead of PUNCH.

Figure 230 on page 174 shows an example of PUNCH. The CP SPOOL PUNCH HOLD command prevents the file from being released to the CP spooling queue. The first PIPE command displays the contents of the file MYBOOK SCRIPT. Ordinary text records are in the file. The second PIPE command punches the file. Next, a CP CLOSE command is entered to create the spool file. (Remember that CMS Pipelines does not issue a CLOSE.) Finally, a QUERY PUNCH command is entered to show the status of the spool file.

```
spool punch hold
Ready;
pipe < mybook script | console
This is a test file.
It is used in examples in
the "z/VM: CMS Pipelines User's Guide."
Ready;
pipe < mybook script | punch
Ready;
close punch name mybook1 script
PUN FILE 0047 SENT FROM YOURID   PUN WAS 0047 RECS 0003 CPY  001 A HOLD   NOKEEP
Ready;
query punch all
ORIGINID FILE CLASS RECORDS  CPY HOLD DATE  TIME      NAME      TYPE      DIST
YOURID   0047 A PUN 00000003 001 USER 01/16 16:08:22 MYBOOK1   SCRIPT    111/AAA
Ready;
```

Figure 230. Example of the PUNCH Stage

The next example (Figure 231) shows how to use URO to write to the virtual punch. By default, the URO stage writes to the virtual printer (device address 00E). To make URO write to the virtual punch, specify 00D as an operand. In the example, a SPECS stage is used to add channel command codes to the records from the file MYBOOK SCRIPT. SPECS puts X'41' in the first column of every record.

```
pipe < mybook script | specs x41 1 1-* 2 | uro 00d
Ready;
close punch name mybook2 script
PUN FILE 0048 SENT FROM YOURID   PUN WAS 0048 RECS 0003 CPY  001 A HOLD   NOKEEP
Ready;
query punch all
ORIGINID FILE CLASS RECORDS  CPY HOLD DATE  TIME      NAME      TYPE      DIST
YOURID   0047 A PUN 00000003 001 USER 01/16 16:08:22 MYBOOK1   SCRIPT    111/AAA
YOURID   0048 A PUN 00000003 001 USER 01/16 16:15:34 MYBOOK2   SCRIPT    111/AAA
Ready;
```

Figure 231. Punching a File with URO

Writing to the Printer (PRINTMC, URO)

Two stages write to the virtual printer (device address 00E): PRINTMC and URO. They have identical functions. However, because URO can write to the virtual punch as well as the virtual printer, you might find it more useful in execs.

Both stages expect a channel command as the first byte of every record. If the records you want to print do not contain channel commands, you must add them.

Some files created by programs (such as compiler listings) have *carriage control* characters in the first byte of each record. There are two kinds of carriage control associated with listing files: *machine carriage control* and *ASA carriage control*. Machine carriage control happens to be identical with the channel commands needed by PRINTMC and URO. You can use these records without alteration. ASA carriage control, however, is not the same—it must be converted.

To convert ASA carriage control to machine carriage control, use the ASATOMC stage. ASATOMC reads records from its input stream, converts the carriage control, and writes the changed records to its output stream. If the input records already

have machine carriage control, ASATOMC writes them unchanged to its output stream. (To convert from machine carriage control to ASA, use the MCTOASA stage.)

You can tell whether a file has ASA carriage control by looking at it. If the first byte of a record with carriage control contains any of the following characters, it has ASA carriage control:

1   (X'F1') Skip to new page and print the line. The line is printed at the top of the next page. The numbers 2 through 9 and the letters A through C are defined for the other channels, but are seldom used.

(blank)   Print on the next line.

0   Skip one line and print. That is, print one blank line and then the data part.

–   Skip two lines before printing.

+   Overprint the line on the previous one.

The example in Figure 232 prints a file using PRINTMC. The file being printed, PIPDUMP LISTING, contains carriage control. The filter ASATOMC is used because we are not sure what kind of carriage control is used.

```
spool printer hold
Ready;
pipe < pipdump listing | asatomc | printmc
Ready;
close printer name stall1 listing
PRT FILE 0041 SENT FROM YOURID   PRT WAS 0041 RECS 0072 CPY  001 A HOLD   NOKEEP
Ready;
query printer all
ORIGINID FILE CLASS RECORDS  CPY HOLD DATE  TIME      NAME     TYPE     DIST
YOURID   0041 A PRT 00000072 001 USER 01/16 13:30:37 STALL1   LISTING  111/AAA
Ready;
```

Figure 232. Printing a File with PRINTMC

Figure 233 shows how to print the same file with the URO stage.

```
pipe < pipdump listing | asatomc | uro
Ready;
close printer name stall2 listing
PRT FILE 0042 SENT FROM YOURID   PRT WAS 0042 RECS 0072 CPY  001 A HOLD   NOKEEP
Ready;
query printer all
ORIGINID FILE CLASS RECORDS  CPY HOLD DATE  TIME      NAME     TYPE     DIST
YOURID   0041 A PRT 00000072 001 USER 01/16 13:30:37 STALL1   LISTING  111/AAA
YOURID   0042 A PRT 00000072 001 USER 01/16 13:31:08 STALL2   LISTING  111/AAA
Ready;
```

Figure 233. Printing a File with URO

An easy way to print a file without carriage control is by putting blanks in the first column of each record. Then use ASATOMC to convert those blanks to machine carriage control. Figure 234 on page 176 shows an example.

```
pipe < mybook script | specs 1-* 2 | asatomc | printmc
Ready;
close printer name mybook script
PRT FILE 0043 SENT FROM YOURID   PRT WAS 0043 RECS 0003 CPY  001 A HOLD   NOKEEP
Ready;
query printer all
ORIGINID FILE CLASS RECORDS  CPY HOLD DATE  TIME      NAME      TYPE     DIST
YOURID   0041 A PRT 00000072 001 USER 01/16 13:30:37 STALL1    LISTING  111/AAA
YOURID   0042 A PRT 00000072 001 USER 01/16 13:31:08 STALL2    LISTING  111/AAA
YOURID   0043 A PRT 00000003 001 USER 01/16 13:32:07 MYBOOK    SCRIPT   111/AAA
Ready;
```

Figure 234. Printing a File without Carriage Control

Reading Spool Files (READER)

Reading a spool file from your virtual reader involves two operations:

- Reading the records from the virtual reader
- Filtering and deblocking those records as necessary.

The READER stage reads a spool file and writes a record to its primary output stream for each channel command word (CCW) in the spool file (unless the MONITOR or 4KBLOCK parameters are specified). The first character of a record is the channel command code; it is followed by the data part of the record.

READER does not filter and deblock the data. Filtering and deblocking is often necessary because different facilities create spool files in different formats. The general formats are:

- Virtual punch format.

  This is the simplest format. The first byte of every record contains either X'03' or X'41'. X'03' indicates no-operation. X'41' marks data records. The maximum record length is 80 columns. Shorter records are usually padded with blanks.

  This format is often used for electronic mail. Some mail facilities do not block records—one line in the mail file is one data record in the spool file. However, other mail facilities block the data records before punching them. Not all of the mail facilities block records in the same way. Two commonly used blocking formats are NETDATA format and DISK DUMP format.

- Virtual printer format.

  In this format, the file contains data records possibly interspersed with control information (such as forms control buffers). The longest data record is 204 characters for a virtual file for an IBM 3800 printer. A record having a X'5A' carriage control can be longer. (X'5A' indicates an *oversize* record with data for an all-points-addressable printer).

- CP formats.

  CP generates spool files with specialized formats (for instance VMDUMP files or monitor data files).

- Real card reader format.

  Files created by a real card reader have a format similar to a virtual punch file. Such files often have channel command codes of X'42'. Few real card readers are still being used, so it is not likely that you will have to process this format.

So, although READER reads the records of a spool file, additional processing is usually needed to unravel them. To determine the format of a virtual reader file, use the CMS RDR command. RDR generates a return code and displays a message for each type of file recognized.

Virtual Reader Characteristics

The characteristics of your virtual reader affect the results you'll get when executing a READER stage. Before using the READER stage, enter CP SPOOL commands to change the virtual reader characteristics to your liking. The keywords on the SPOOL command for a virtual reader control several things:

CLASS   Sets the class of spool files that can be read by the reader. A spool file has a class (A through Z or 0 through 9) associated with it; likewise, a reader can have one of the 36 classes associated with it or it can be set to read files irrespective of their class (indicated by setting class * for the reader).

NOCONT   Only one spool file is read for each call to READER. This is the most common way of reading reader files.

CONT   All available files are read by a single call to READER.

The files in your reader queue are controlled through a combination of READER options and the options in effect for the virtual reader on which they are processed. Table 1 summarizes the disposition of a reader file after it has been processed, depending on the hold/keep settings specified for the reader, and the READER options.

Table 1. Disposition of Reader Files Depending upon HOLD/KEEP Settings

READER Options	Device Options	Disposition
PURGE	Any setting	The file is purged.
KEEP	Any setting	The file is retained in user hold status. KEEP status remains on the file.
HOLD	KEEP	The file is retained in user hold status.
HOLD	NOKEEP	The file is retained and remains eligible for processing on your virtual reader.
NOHOLD/NOKEEP	KEEP	The file is retained in user hold status.
NOHOLD/NOKEEP	HOLD	The file is retained and remains eligible for processing on your virtual reader.
NOHOLD/NOKEEP	NOHOLD/NOKEEP	The file is purged.

The READER stage reads the first file in the reader queue. Enter a CP ORDER command to put one or more spool files at the beginning of the reader queue so that they are processed by the next call to READER. (You can also use the FILE operand of READER to read a specific spool file.) Use the CP PURGE command to remove one or more spool files from the system.

READER does not read files in HOLD status; it simply reads the next file if there is one. Use the CP CHANGE command to change the hold status of a reader file.

READER does not read files having classes that do not match the class setting for the reader. Use the CP SPOOL command to change the reader class or the CP CHANGE command to change the classes of the spool files.

READER closes the reader after reading the file. CP purges the file unless the reader is spooled hold.

See "Processing Reader Files" on page 252 for an example of an exec that issues the necessary CP commands before reading the virtual reader.

Reading Printer Files

When reading a printer file residing in a reader, you don't have to worry about deblocking data. However, printer files do contain carriage control, so you must decide whether you want to keep it or remove it. Often you'll want to retain the carriage control so you are able to print the file after storing it, but remember to delete non-translatable channel command codes before printing. If you don't want to keep the carriage control, use a SPECS stage to remove the first column of the record, or the STRNFIND stage to delete the entire record. You can also convert the carriage control to ASA.

The example in Figure 235 shows how to read a printer file.

```
spool printer to * nohold
Ready;
pipe < mybook script | specs 1-* 2 | asatomc | printmc
Ready;
close printer
RDR FILE 0057 SENT FROM YOURID   PRT WAS 0057 RECS 0003 CPY  001 A NOHOLD NOKEEP
Ready;
pipe reader file 57 hold | console


 This is a test file.
 It is used in examples in
 the "z/VM: CMS Pipelines User's Guide."
Ready;
pipe reader file 57 hold | mctoasa | console
 This is a test file.
 It is used in examples in
 the "z/VM: CMS Pipelines User's Guide."
Ready;
```

Figure 235. Reading a Printer File

The first PIPE command prints the file with machine carriage control. The second PIPE command reads the file and displays its contents without alteration. (Some nondisplayable characters are in the response.) The third PIPE command also reads the file, but it converts the carriage control to ASA before the file is displayed.

Reading Punch Files

This section describes how to read punch files. For our first example, let's create a plain punch file and read it. Figure 236 on page 179 shows the screen dialog.

```
spool punch to * nohold
Ready;
pipe < mybook script | punch
Ready;
close punch
RDR FILE 0053 SENT FROM YOURID   PUN WAS 0053 RECS 0003 CPY  001 A NOHOLD NOKEEP
Ready;
pipe reader | console
*
*This is a test file.
*It is used in examples in
*the "z/VM: CMS Pipelines User's Guide."
Ready;
```

Figure 236. Reading a Plain Punch File

The SPOOL command directs punched files to your virtual reader and ensures that there is no hold on the file. The first PIPE command punches the file. The second PIPE command reads and displays the file.

Notice that there is an extra line in the console display and that there are nondisplayable characters in the first column. In the example, nondisplayable characters are represented by asterisks (*). The extra line is a record that does not contain data (channel command X'03') and the nondisplayable characters are the channel commands.   Figure 237 shows a stage that filters the records.

```
/* PLAIN REXX -- Select data records and strip command codes */
'callpipe',
   '*:',
   '| find' '41'X||,    /* Keep only data records              */
   '| specs 2-* 1',     /* Remove channel command              */
   '| *:'
exit rc
```

Figure 237. A Filter for Reading Plain Punch Files: PLAIN REXX

PLAIN REXX restores the original file:

```
pipe reader | plain | console
This is a test file.
It is used in examples in
the "z/VM: CMS Pipelines User's Guide."
Ready;
```

A variation of this format is created by the CMS PUNCH command. The PUNCH command adds a header record (Figure 238 on page 180). PLAIN REXX is used to remove channel commands and records that do not contain data.

```
punch mybook script
RDR FILE 0054 SENT FROM YOURID   PUN WAS 0054 RECS 0004 CPY  001 A NOHOLD NOKEEP
Ready;
pipe reader file 54 | plain | console
:READ  MYBOOK   SCRIPT   A1 BAR191 01/16/92 10:17:50
This is a test file.
It is used in examples in
the "z/VM: CMS Pipelines User's Guide."
Ready;
```

Figure 238. Reading a File Created by CMS PUNCH

To get rid of the header record, add a DROP 1 stage to the pipeline (or specify the NOHEADER option on CMS PUNCH).

If the records are blocked, you need to add deblocking stages. An example of deblocking a punch file in NETDATA format is shown in Figure 246 on page 188. To deblock other formats you need to learn the blocking scheme of the facility that created the file. Refer to the documentation for the facility.

Chapter 9. Blocking and Deblocking

CMS Pipelines gives you the ability to convert plain records to *blocked* records and to deblock records. There are several reasons why you might want to block records. In a communications program, for example, you might want to send blocked records instead of individual records for better performance. Or, in a data management program, you might want to block records for efficient storage. Naturally, you would eventually need to deblock the records you have blocked.

You may also have a need to deblock data that another application or that another system has blocked. For example, you may need to deblock files that originated from a non-CMS system. If the blocking format matches a format supported by CMS Pipelines, you can use CMS Pipelines to deblock the data.

Commonly used blocking formats range from simple fixed records put one after the other to records wrapped in several layers of protocol. CMS Pipelines handles several blocking formats. Each of the following formats is discussed in its own section in this chapter:
- Fixed format
- CMS variable format
- MVS variable format
- Line-end character format
- NETDATA format
- IEBCOPY unloaded data set format
- Packed format.

The filters that support these blocking formats are BLOCK, DEBLOCK, IEBCOPY, PACK and UNPACK. At the end of the chapter we also discuss the FBLOCK stage. FBLOCK creates fixed-length output records from either variable- or fixed-length input records. Unlike the other blocking filters, FBLOCK either blocks or deblocks depending on the relationship of the block size requested to the lengths of the input records.

When blocking records, stages put records together in a buffer that usually has room for more than a single record. When the block is created, the filter writes that block to its output stream. Although it is convenient to say that the filter writes a block, the filter actually writes a record (just as other filters do). The record just happens to contain logical records within it. A blocking filter might, for example, read three input records but write only one output record that contains the three records read.

Some of the blocking filters let you specify the block size. You can select a block size that is appropriate for your application. The CMS file system, for instance, uses block sizes of 512, 1 KB, 2 KB, or 4 KB. OS/MVS simulation access methods support blocks with up to 32 KB of data.

In some formats, a record can *span* blocks. This means that part of the record is in one block and part of it is in the next. In other formats, records cannot span blocks, which means that the entire record must fit in the block.

Another characteristic of blocking formats, in addition to block size and spanning, is the technique used to indicate the end of a record. The end of a record can be defined by a fixed length, a line-end character, or a record descriptor word.

Blocking and Deblocking

When the end of a record is defined by a fixed length, you know where a record ends because each record is the same known length. When line-end characters are used, a special character (one that is not in the data itself) is used to mark the end of a record. With this scheme, the lengths of the records can vary. When a *record descriptor word* is used, one or more bytes (known as the record descriptor word) are added to the data portion of the record. The record descriptor word describes the record and defines its end.

Blocking operations are *reversible* when it is possible to recover the original format of the file. This is the case for instance when blocking variable length records in the MVS variable format, but not in general when deblocking fixed format records that span blocks.

Fixed Format

In fixed-format blocking, records having fixed lengths are abutted without any control information. The records cannot span blocks, and the block length must be a multiple of the record length. Figure 239 shows a 240-byte block that contains three 80-byte records. Notice that all three records fit neatly into the block.

```
240-Byte Block
```

Record 1 (80 bytes)	Record 2 (80 bytes)	Record 3 (80 bytes)
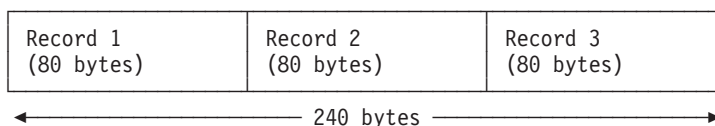
←——————— 240 bytes ———————→

Figure 239. 80-Byte Records in a 240-Byte Block

To block records in fixed format, use the BLOCK stage. BLOCK requires the block size as an operand. Following the block size, you can optionally specify the FIXED operand. (BLOCK creates fixed-format blocks by default.)

BLOCK FIXED expects fixed-length records in its input stream. The block size you specify must be a multiple of the record length. Conceptually, BLOCK FIXED reads records from its input stream, concatenating them until the block size is reached. Then BLOCK FIXED writes the block (actually a record containing one or more logical records) to its output stream.

In the example in Figure 240 on page 183, an existing file (NUMBER LIST A) is converted by a CMS COPYFILE command to F-format with a record length of 10. The first PIPE command displays the contents of the file. Note that each record has some trailing blanks. The last PIPE command shows the use of BLOCK FIXED. It blocks the records from the file and then displays the blocked records. Notice that the specified block size 30 is a multiple of the record length 10.

```
copy number list a (recfm f lrecl 10
Ready;
pipe < number list | console
one
two
three
four
five
Ready;
pipe < number list | block 30 fixed | console
one      two        three
four      five
Ready;
```

Figure 240. Blocking Records in Fixed Format

To deblock the records, use the DEBLOCK stage. Specify the FIXED operand after the DEBLOCK keyword, followed by the length of the original records. FIXED is the default for DEBLOCK, so you can omit the FIXED operand if you wish.

Figure 241 shows two examples of deblocking. In the first PIPE command, the records are blocked by BLOCK and are correctly deblocked using a record length of 10. The second PIPE command shows what happens if you specify the wrong record size.

```
pipe < number list | block 30 fixed | deblock fixed 10 | console
one
two
three
four
five
Ready;
pipe < number list | block 30 fixed | deblock fixed 11 | console
one      t
wo        th
ree
four        f
ive
Ready;
```

Figure 241. Deblocking Fixed-Format Blocks

CMS Variable Format

In CMS variable format, records are blocked with a half-word (two bytes) record descriptor indicating the length of the record. The record descriptor is located immediately before the record. This 2-byte descriptor restricts the length of the records to be blocked to one less than 64 KB. In general, records are spanned over blocks; even the record descriptor can be spanned.

The CMS variable format also has an end-of-data (or end-of-file) indication. End-of-file is indicated with a halfword (record descriptor) of binary zeros if there are two or more bytes available in the last block. If there is only one byte available, a single byte of zero indicates end of data. If there are no bytes available on the last block, the end of the last record indicates end of file.

Blocking and Deblocking

The end-of-file indication is optional. Blocks created by the BLOCK CMS stage do not have the additional zeros. The DEBLOCK CMS stage successfully deblocks records even if the end-of-file indication is omitted.

To block records in CMS variable format, use the BLOCK CMS stage. Specify a block size followed by the CMS operand. Use DEBLOCK CMS to deblock the records. Do not specify a block size on DEBLOCK CMS. Figure 242 shows an example of BLOCK CMS and DEBLOCK CMS. When displaying files blocked in CMS, there are bound to be nondisplayable characters in the record descriptors. In Figure 242, they appear as quotation marks (").

```
pipe < records script | console
This is the first record.
This is the second.
And this is the third.
Ready;
pipe < records script | block 30 cms | console
""This is the first record.""T
his is the second.""And this i
s the third.
Ready;
pipe < records script | block 30 cms | deblock cms | console
This is the first record.
This is the second.
And this is the third.
Ready;
```

Figure 242. Blocking Records in CMS Variable Format

MVS Variable Format

MVS has four kinds of variable-format records. All four have a 4-byte *block descriptor word* at the beginning of each block that describes the block. The first two bytes contain the length of the complete block including the block descriptor word. The last two bytes are zero. Blocks are limited to 32,760 bytes including the block descriptor word. The four kinds of variable-format records are:

V   Deblocked variable records.

     In this format, the records are blocked, but with only one record per block. Before each record there is a 4-byte *record descriptor word*. In the first two bytes is the length of the record plus the length of the record descriptor. The second two bytes contain binary zeros. Thus, the longest record possible is 32,756 bytes.

VB   Variable blocked records.

     In this format the records are blocked with more than one record per block. Before each record there is a 4-byte record descriptor word. Each block contains as many complete records (with their record descriptors) as can fit within the block. Logical records are not spanned across block boundaries.

VBS Variable block spanned records.

     In this format, the records are blocked and records can span blocks. Each record or part of a record is a *segment*. Instead of record descriptor words, there are *segment descriptor words*. The 4-byte segment descriptor word contains a two-byte segment length and *segmentation flags*. The segmentation flags define whether the segment is:
     • A complete record,

- The first part,
- The last part, or
- An intermediary part of a record that is neither first nor last.

The length of the record is the sum of the lengths of the segments. Because the length is not written explicitly, this format supports logical records longer than 32 KB. In MVS, the user has to handle the segmentation; in z/VM, CMS Pipelines does it for you.

VS   Variable spanned records.

In this format, the records are blocked, but with only one record or a part of one record per block. Each new record is placed in a new block, even if there is space available in the previous block. Each record or part of a record is a segment. Instead of record descriptor words, there are segment descriptor words. The 4-byte segment descriptor word contains a two-byte segment length and segmentation flags. The segmentation flags define whether the segment is:
- A complete record,
- The first part,
- The last part, or
- An intermediary part of a record that is neither first nor last.

The length of the record is the sum of the lengths of the segments. Because the length is not written explicitly, this format supports logical records longer than 32 KB. In MVS, the user has to handle the segmentation; in z/VM, CMS Pipelines does it for you.

DEBLOCK supports all four input formats, but IBM recommends using the V format (specify with the V operand). DEBLOCK V determines the structure of the blocks from record descriptor words and segment descriptor words.

```
pipe tape | deblock v | take 20 | console
```

When blocking, you must decide the format you want. Code the desired blocking format as a keyword, for instance BLOCK VBS.

Line-End Character Format

In *line-end character* format, the end of each logical record is indicated by a specific character. A line-end character does not follow the last record. The records can span blocks.

Use the BLOCK filter with the LINEND operand to block records in line-end character format. You also need to specify a block size and a line-end character. The character you choose should not occur in the data being blocked (otherwise, it cannot be deblocked properly). Often the best choice is a hexadecimal value that cannot be typed on a terminal.

Figure 243 on page 186 shows an example in which X'F0' is used as the line-end character. X'F0' happens to be a displayable character: 0. A block size of 80 is used.

```
pipe < legumes script | console
Peas
Bush beans
Pole beans
Lima beans
Ready;
pipe < legumes script | block 80 linend f0 | console
Peas0Bush beans0Pole beans0Lima beans
Ready;
```

Figure 243. Blocking Data with Line-End Characters

Notice how the hexadecimal value is specified following `linend`. The hexadecimal value is not enclosed by single quotation marks. Also notice in the output that a line-end character is not placed after the last logical record.

Let's take another example in which the records span blocks. We'll use the same LEGUMES SCRIPT file and choose a block size of 5. In the example, the character `0` is specified instead of its hexadecimal value `f0`; you can use either method to specify the line-end character. Figure 244 shows the result.

```
pipe < legumes script | block 5 linend 0 | console
Peas0
Bush
beans
0Pole
 bean
s0Lim
a bea
ns
Ready;
```

Figure 244. Spanning Blocks with Line-End Characters

Each displayed line is one block of 5 characters. Since the block size is so small, most of the records span at least one block.

To deblock records in line-end character format, use the DEBLOCK stage with the LINEND operand. DEBLOCK reads each record from its input stream, and writes a record to its output whenever it finds a line-end character (or when it has read the last record). It does not write the line-end character with the data.

Figure 245 on page 187 shows PIPE commands that block and deblock files. Do not specify the block size as an operand on DEBLOCK. Instead, specify a LINEND operand that identifies the line-end character.

```
pipe < legumes script | block 80 linend f0 | deblock linend f0 | console
Peas
Bush beans
Pole beans
Lima beans
Ready;
pipe < legumes script | block 5 linend 0 | deblock linend 0 | console
Peas
Bush beans
Pole beans
Lima beans
Ready;
```

Figure 245. Deblocking Data with Line-End Characters

NETDATA Format

In the NETDATA format, records are segmented with a 1-byte segment length, a flag byte, and up to 253 bytes of data. The logical record can be any length. In general, segments are spanned across blocks.

The flag byte indicates if the record contains data (X'C0') or control information (X'E0'). Control records have *text units* to encode the attributes of a data set (such as the name of the data set). For a complete description of NETDATA format, see the *z/VM: CMS Macros and Functions Reference*.

CMS Pipelines supports this format with the NETDATA operand on the BLOCK and DEBLOCK filters. DEBLOCK TEXTUNIT further deblocks text units in control records.

NETDATA format is also supported by the CMS NETDATA command, which processes spool files that are in NETDATA format. In MVS, the INMRCOPY utility processes files in NETDATA format.

If you want to block records in NETDATA format, you'll need to learn more about NETDATA format in the *z/VM: CMS Macros and Functions Reference*. The BLOCK stage with the NETDATA operand expects input records with an appropriate flag byte. The BLOCK stage also expects appropriate control records.

To deblock records in NETDATA format, use the DEBLOCK stage with the NETDATA operand. Figure 246 on page 188 shows how to deblock a NETDATA file that is in your reader. (A spool file in NETDATA format that does not contain any data, such as an acknowledgement, will produce no output.)

```
/* DEBNET REXX -- Deblock a reader file in NETDATA format.         */
'callpipe',
  'reader',                /* From reader                          */
  '| find' '41'x||,        /* Only data records                    */
  '| specs 2-* 1.80',      /* Discard channel command and pad to 80 */
  '| deblock netdata',     /* Deblock                              */
  '| find' 'c0'x||,        /* Only data records                    */
  '| specs 2-* 1',         /* Remove control character             */
  '| *:'
```

Figure 246. Deblocking NETDATA-Format Files

The READER stage reads the spool file and writes the blocked records to its output stream. We can't read these records directly with a DEBLOCK stage because records from your reader have channel commands in column 1. (See Chapter 8, "Using Unit Record Devices," on page 173.) In fact, some of the records may not contain actual file data. These records have X'03' in column 1. We're interested in only data records, which have a X'41' in column 1. So, we use a FIND stage to select only the data records, and a SPECS stage to create records without channel commands.

After the records are selected and adjusted, DEBLOCK NETDATA deblocks them. The output from DEBLOCK also consists of control records and data records. The last two stages select only the data records and create records without a flag byte. These records are the original, deblocked records.

IEBCOPY Unloaded Data Set Format

The MVS utility, IEBCOPY, unloads disk data sets in a format that can be restored to a device other than the one that originally housed the data set. IEBCOPY often unloads these data sets in VBS format. Each logical record contains data from one or more physical disk blocks, including record identifiers. End-of-file is indicated by a record with zero key and data (also on the disk).

There are several sub-files in an unloaded partitioned data set (PDS). The first file is the directory for the PDS. The remaining files are the members. The directory is sorted by member name, but the files are ordered by their position in the data set and not necessarily in the order of the directory.

The sample file OSPDS REXX is shipped with CMS Pipelines (usually it resides on the MAINT user ID's 193 disk). It unravels this format by first deblocking the variable format records (usually with DEBLOCK V). It discards the first two records defining data set attributes, removes the record identification (and optionally the key) with IEBCOPY. It processes the directory and sorts it in the order the members are unloaded. The output is a stacked file with *COPY separators or individual disk files.

There is no built-in filter to create an unloaded data set. If you write a stage to do it, remember to skip record zero on each track.

Packed Format (PACK, UNPACK)

The packed format supported by CMS Pipelines is the same format that CMS uses when storing packed files. (You can create packed files by using the PACK option on the CMS COPYFILE command or the SET PACK subcommand of XEDIT.) Packed records have a fixed-length record format and a record length of 1024. They contain logical records in which multiple occurrences of a character are replaced by a count. Logical records can be 64KB or longer.

To pack records, use the PACK stage. To unpack records, use the UNPACK stage. First, let's see how to pack records.

When packing records, you must know whether the records have variable lengths or a fixed length. For our first example, let's create packed fixed-length records. Figure 247 shows how. The < stage reads records from an F-format file named MYCODE ASSEMBLE. When the input records to PACK have a fixed length, specify the FIXED operand on PACK. PACK reads the fixed-length records from its input stream, compresses the records and writes the compressed records to its output stream. Then the records are written to a file named MYCODE PACKED A.

```
pipe < mycode assemble | pack fixed | > mycode packed a fixed
Ready;
```

Figure 247. Packing Fixed-Length Records

All the records in the input stream to the > stage have a length of 1024. (PACK fills its last output record with zeros if necessary.)

Now let's pack variable-length records. This time we'll read a file named MYBOOK SCRIPT, which is a V-format file. Assume that the length of the longest record in the file is 110 bytes. Figure 248 shows how to pack the file.

```
pipe < mybook script | pack variable 110 | > packbook script a fixed
Ready;
```

Figure 248. Packing Variable-Length Records

The < stage reads the file MYBOOK SCRIPT and writes those variable-length records to its output stream. PACK reads these records, compresses them, and writes the compressed records to its output stream. Then the > stage writes the packed records to a file named PACKBOOK SCRIPT A.

Notice that the operands `variable` and `110` are specified on the PACK stage. What if you don't know the length of the longest record? It is still possible to pack the records, but you must use a multistream pipeline. See the *z/VM: CMS Pipelines Reference* for more information about packing variable-length records.

Now that we have packed files, how do we unpack them? Use the UNPACK stage. UNPACK has no operands. It reads records from its input stream and determines whether the records are packed. If they are packed, UNPACK converts them to regular records and writes the converted records to its output stream. If the records are not packed, UNPACK simply copies the records to its output stream.

Figure 249 shows a PIPE command that uses the UNPACK stage to unpack the file if it is packed. The stage unpacks the file INPUT FILE A and displays the last 20 lines.

```
pipe < input file a | unpack | take last 20 | console
```

Figure 249. Unpacking Records

It's important to keep all packed records intact if you plan to unpack them. Do not use filters that select records, such as TAKE or DROP, before the UNPACK stage. If you do, the remaining records may not look like a packed file to UNPACK. In that case, UNPACK copies the records to its output stream without converting them. In Figure 249, for example, the TAKE stage is after the UNPACK stage. Reversing the order of the stages would likely cause problems.

Creating Fixed-Format Records with FBLOCK

In an earlier section we saw how to block and deblock fixed-length records. One restriction with the BLOCK stage, however, is that all input records must have the same length. A second restriction is that the block size must be a multiple of the record length; that is, the records cannot span blocks.

The FBLOCK stage does not have those restrictions. Like the BLOCK stage, FBLOCK writes output records having fixed lengths, but that is where the similarity ends. FBLOCK accepts both variable- and fixed-length records as input. The block size specified for output records does not have to relate to the size of the input records in any way.

FBLOCK reads records from its input stream and writes output records of the requested length. Whenever FBLOCK's internal buffer is filled, it writes an output record and begins refilling the buffer. Imagine that FBLOCK concatenates all its input records into one long string. Then it chops that long string into pieces of the size you request without regard to the original record boundaries.

Figure 250 on page 191 shows several examples of FBLOCK. The size of the desired output record is specified as an operand.

```
pipe literal bbbb| literal aaa| fblock 3 | console
aaa
bbb
b
Ready;
pipe literal bbbb| literal aaa| fblock 1 | console
a
a
a
b
b
b
b
Ready;
pipe literal bbbb| literal aaa| fblock 10 | console
aaabbbb
Ready;
```

Figure 250. Creating Fixed-Length Records with FBLOCK

The preceding examples showed FBLOCK working with input records of variable lengths. FBLOCK also works with input records having a fixed length. In fact, if you specify a block size that is a multiple of the fixed-length input records, FBLOCK behaves the same as BLOCK FIXED. If, however, you specify a block size that is not a multiple of the record length, FBLOCK will span the records as necessary; BLOCK FIXED will display an error message.

It's worth looking at an example of spanned fixed-length records. You may encounter files in this format that you want to deblock. Figure 251 shows an example in which the block size is 200, but the record length is 80. In this case some records (record 3 in the example) begin on one block and end on the next.

```
200-Byte Blocks

┌──────────────┬──────────────┬─────────┐
│ Record 1     │ Record 2     │ Half of │ ◄─── Block 1
│ (80 bytes)   │ (80 bytes)   │ Record 3│
├─────────┬────┴──────────────┴─────────┤
│ Half of │ Null Data                   │ ◄─── Block 2
│ Record 3│                             │
└─────────┴─────────────────────────────┘
◄─────────────── 200 bytes ───────────────►
```
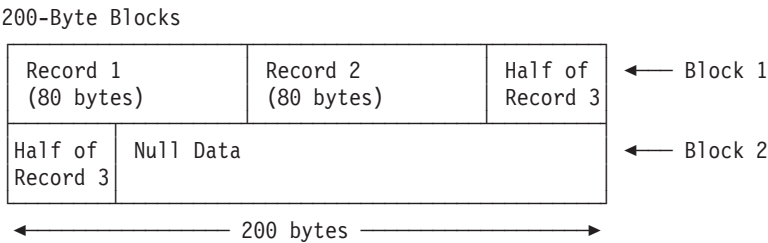
Figure 251. 80-Byte Records in a 200-Byte Block

Figure 252 on page 192 shows PIPE commands that create fixed blocks having spanned records. The second PIPE command shows how to specify a pad character: type it after the block size. The example uses a hyphen (-), but you can use any character or a hexadecimal value (such as `f0`).

```
pipe literal 12345| literal 12345| fblock 7 | console
1234512
345
Ready;
pipe literal 12345| literal 12345| fblock 7 - | console
1234512
345----
Ready;
```

Figure 252. Creating Spanned Records with FBLOCK

Let's see what happens when we try to deblock these by adding a second FBLOCK stage. Specify the original record length, which we know to be 5. Figure 253 shows what happens.

```
pipe literal 12345| literal 12345| fblock 7 | fblock 5 | console
12345
12345
Ready;
pipe literal 12345| literal 12345| fblock 7 - | fblock 5 | console
12345
12345
----
Ready;
```

Figure 253. Deblocking Spanned Records with FBLOCK

As you can see, it is not always possible to reverse the blocking operation when fixed, spanned records are involved. Usually you also need to know the number of records that were originally blocked and add a TAKE stage to discard the extras.

Chapter 10. Using SQL in CMS Pipelines

The SQL device driver lets you use DB2® Server for VM (formerly SQL/DS) from CMS Pipelines. Before you can use DB2 Server for VM from CMS Pipelines, several tasks must be done:

- DB2 Server for VM must know about CMS Pipelines. This process is called *preparing the access module*, and is documented in the *z/VM: CMS Planning and Administration*. It is done once by your system support staff.

- You must be registered as an DB2 Server for VM user. Contact your database administrator if you are not already registered. Your installation may have granted everyone connect authority; you can query tables once you have connect authority.

- To create tables you must have a DBSPACE or write privileges to a space owned by someone else. Your database administrator can allocate a space to you.

- Issue the SQLINIT command to create the modules required by DB2 Server for VM.

Basic DB2 Server for VM education is beyond the scope of this book. Refer, instead, to the DB2 Server for VM book listed in the bibliography at the end of this book.

SQLSELEC - An Example Program to Format a Query

CMS Pipelines provides an example program named SQLSELEC REXX. You may need to get access to this program from your system administrator. (Usually it resides on the MAINT machine's 193 minidisk.) The filter takes a query as an operand. It describes the query and formats the result (see the example in Figure 254). The first line of the response contains the names of the columns padded with hyphens to their maximum length. The remaining lines are the result of the query.

```
pipe sqlselec project_name from sqldba.projects | console
PROJECT_NAME--—
BLUE MACHINE
GREEN MACHINE
ORANGE MACHINE
RED MACHINE
WHITE MACHINE
Ready;
pipe sqlselec salary, name from q.staff where years is null | console
SALARY--- NAME-----
+16086.30 JONES
+13504.60 SMITH
+12954.74 NAUGHTON
+11508.60 SCOUTTEN
Ready;
```

Figure 254. SQLSELEC Examples

Creating, Loading, and Querying a Table

The SQL device driver lets you create and maintain DB2 Server for VM tables. Two ways to create a table are shown in Figure 255. The first example shows how to issue a single DB2 Server for VM statement as an operand on the SQL device driver. Specify the CREATE TABLE statement after the EXECUTE operand, as shown in Figure 255. The second example shows that the SQL EXECUTE reads statements from its primary input stream.

```
pipe sql execute create table jtest (kwd char(8), text varchar(80))
Ready;
pipe literal create table jtest (kwd char(8), text varchar(80)) | sql execute
Ready;
```

Figure 255. Creating an DB2 Server for VM Table

Use SQL INSERT to load data into the table (Figure 256). The first eight characters of each record are stored in the column kwd; the remainder of the record is loaded into the column text. SPECS is used with a conversion option to generate the halfword length required for the variable character string.

```
/* Insert rows in an SQL table */
signal on novalue
address command
'PIPE',
   'literal DMS       Conversational Monitor System'||,
   '| literal HCP      Control Program'||,
   '| literal DMT      Remote Spooling Communications Subsystem'||,
   '| specs 1.8 1',
      '9-* v2c 9',
   '| sql insert into jtest'
exit RC
```

Figure 256. Inserting Rows in an SQL Table

When SQL INSERT is used without other operands, all columns defined for the table are loaded with data from the input record. SQL gets the length of each column from DB2 Server for VM. Data must be loaded in the format used by DB2 Server for VM, which usually involves conversion. The SPECS stage copies the first eight characters of each record without change. It then inserts a halfword field with the number of bytes remaining in the input record and copies the rest of the input record after this halfword. This is the format DB2 Server for VM requires for a row with a fixed and a variable-length character variable.

Figure 257 shows how to use SQL DESCRIBE SELECT to see the format of the input record or the result of a query.

```
pipe sql describe select * from jtest | console
453     CHAR        8       8 KWD
449     VARCHAR    80      82 TEXT
Ready;
```

Figure 257. Describing a Query

Each line describes a column in the table. The first column of the record is the numeric DB2 Server for VM field code. It is decoded in the next column. A column with the length (or precision) of the field as perceived by DB2 Server for VM is next. The following number is the number of characters required to represent the field when loading with SQL INSERT and when queried by SQL SELECT. Note that the varying character field (VARCHAR) has two bytes reserved for the length prefix. Finally, the name of the column is shown.

To query a table, use SQL SELECT. Figure 258 shows an example.

```
pipe sql select * from jtest | console
""DMT     """"Remote Spooling Communications Subsystem
""HCP     """"Control Program
""DMS     """"Conversational Monitor System
Ready;
```

Figure 258. Querying a Table

The double quotation marks in Figure 258 represent nondisplayable binary data. (On your terminal, they may appear as blanks or as other characters.) The first two positions of each column is the DB2 Server for VM indicator word that tells whether the column is null. This information may be needed to process the result of a query in a table with columns that can be null. Figure 259 shows how to suppress these indicator words. Type the NOINDICATORS operand after SQL. The query seen by DB2 Server for VM is the same in both cases.

```
pipe sql noindicators select * from jtest | console
DMT     ""Remote Spooling Communications Subsystem
HCP     ""Control Program
DMS     ""Conversational Monitor System
Ready;
```

Figure 259. Suppressing Indicator Words

The remaining two nondisplayable bytes contain the length, in binary, of the varying character field. Use SPECS to discard the columns. An alternative is to use SPECS to format the binary data (see Figure 260).

```
/* SQLFORM EXEC -- query the test table with formatting. */
signal on novalue
'PIPE',
   'sql noindicators select * from jtest',
   '| specs 1.3 1',
      '9.2 c2d 5.2 right',
      '11-* 8',
   '| console'
exit rc
```

Figure 260. Formatting the Field Length

Figure 261 on page 196 shows the results of running SQLFORM.

```
sqlform
DMT 40 Remote Spooling Communications Subsystem
HCP 15 Control Program
DMS 29 Conversational Monitor System
Ready;
```

Figure 261. Running SQLFORM EXEC

SPECS supports conversion between character and binary or floating point. It also lets you construct varying length character fields. The PIPE command in Figure 262 formats a query against the sample table.

```
pipe sqlselec * from jtest | console
KWD---- TEXT----------------------------------------------------------------------
DMT     Remote Spooling Communications Subsystem
HCP     Control Program
DMS     Conversational Monitor System
Ready;
```

Figure 262. Another SQLSELEC Example

Using SPECS to Convert Fields

Input and output records from SQL have data in the format that is defined for the table. For instance, when a column is specified as SMALLINT, the corresponding field in a record is a two-byte binary integer.

Use SPECS to convert from readable formats to the internal ones. The sample program SQLSELEC shows how to format DB2 Server for VM data on output. Table 2 shows how to convert some DB2 Server for VM data types. The input record is assumed to contain a single field.

Table 2. Formatting SQL Data

Data Type	Conversion
Fixed-length character string	This example pads the field to eight bytes: `specs 1-* 1.8`
Variable-length character string	`specs 1-* v2c 1`
Large integer	`specs 1-* d2c 1`
Small integer	`specs 1-* d2c 1.2 right`
Floating point	`specs 1-* f2c 1`
Decimal	There is no SPECS conversion to do this directly. See the following text.

To convert decimal, use XLATE to change leading blanks to zeros, a plus sign to C, and a minus sign to D. Use CHANGE to remove the period if the number of decimals is constant. For instance, if a decimal(5,2) field is always positive and presented right-aligned in columns 1 to 6 with two decimals (period in column 4):

`...xlate *-* space 0|specs 1.3 1 5.6 next /C/ next|spec 1-* x2c 1...`

The subroutine pipeline in Figure 263 processes a number that has a leading sign and a variable number of decimals (or none). The argument is the number of digits (including decimals) and the number of decimal digits. The first argument must be odd.

```
/* DEC2PACK REXX -- convert from decimal to packed.              */
/*                  For example: dec2pack 5 2                    */
signal on novalue
parse arg digits decimals

'callpipe (endchar ? name DEC2PACK)',
  '| *:',
  '| strip any /+ /',           /* Remove positive sign            */
  '| p: nfind -',               /* Select nonnegative ones         */
  '| change //+/',              /* Put default plus sign           */
  '| xi: faninany',             /* Join with negative ones         */
  '| xlate - D + C',            /* Change sign to /360 code points */
  '| c: chop .',                /* Split integer and decimals      */
  '| specs pad 0 2-* 1.' || (digits-decimals) 'right 1' digits+1,
  '| o: overlay',               /* Overlay decimals                */
  '| specs 1-* x2c 1',          /* Convert to hexadecimal          */
  '| *:',
  '?',
  'p:',                         /* Copy negative numbers           */
  '| xi:',
  '?',
  'c:',                         /* Process decimals                */
  '| specs 2-*' (digits-decimals+1)'.'decimals 'left',
  '| o:'
exit rc
```

Figure 263. Converting to Packed Decimal

About Units of Work

DB2 Server for VM commits changes to the database at the end of the unit of work. The unit of work ends with an explicit COMMIT or by CMS reaching the end of command. Unless instructed by an option, SQL does an explicit commit and relinquishes the connection to the database virtual machine when processing is complete. Use the option COMMIT when you wish the unit of work to be committed without releasing the connection to the database machine. Use NOCOMMIT in concurrent SQL stages, and to treat a subsequent SQL stage as the same unit of work.

The unit of work can also be rolled back. That is, the database is restored to the state before the unit of work began. SQL automatically rolls the unit of work back when it receives an error code from DB2 Server for VM. Use SQL ROLLBACK to do an explicit rollback, possibly in response to a CMS or pipeline error condition.

Using Multiple Streams with SQL

SQL EXECUTE processes multiple input and output streams when the primary input stream has multiple insert or query statements, or a mixture of these. Each insert statement causes SQL to read records from a separate input stream, starting with stream number 1. There must be as many additional input streams defined as there are insert statements.

The result of the first query is written to the primary output stream. If the secondary output stream is defined and connected, the result of the query is written there, and so on. More queries are allowed than there are streams defined. The output from the last queries are written to the highest numbered stream defined.

Using Concurrent SQL Stages

You can process the results of a query to construct DB2 Server for VM statements and queries processed in a subsequent SQL stage. As seen from DB2 Server for VM, all concurrent SQL stages are considered to be the same program using multiple cursors.

The option NOCOMMIT *must* be specified when multiple SQL stages are running concurrently. Each stage uses its own cursor; the module is prepared for up to ten cursors.

If one of the stages fails with an DB2 Server for VM error, the unit of work is rolled back and all other SQL stages fail if they access DB2 Server for VM after the error occurred. Use a buffer stage to isolate the programs when building DB2 Server for VM statements from the result of a query. This ensures that the initial query is complete before a subsequent stage starts processing. You can also process the query and store the result in a REXX stemmed array. Test the return code and issue the second SQL pipeline only when the first one ends successfully.

Getting HELP for DB2 Server for VM

DB2 Server for VM stores help information in tables. Your system administrator must install the DB2 Server for VM HELP text before you can use the HELP SQL stage to access these tables. You must also have connect privileges and have run SQLINIT, before using HELP SQL. Type the topic you wish help about as the argument. This may be a DB2 Server for VM statement or a numeric return code. Use HELP SQLCODE to get help for the last return code received from DB2 Server for VM. HELP SQLCODE 1 displays help for the second-to-last return code received, and so on. Figure 264 shows several HELP stage examples.

```
pipe help sql select
pipe help sql 105
pipe help sqlcode
```

Figure 264. Using the HELP Stage to Get DB2 Server for VM Help

Figure 265 shows a session in which a user accesses DB2 Server for VM for the first time. The user runs the initialization procedure (SQLINIT) and then enters a PIPE command that gets help.

```
sqlinit db(sqldba)
Ready;
pipe help sql select
Ready;
```

Figure 265. Running SQLINIT

Chapter 11. Using TCP/IP with CMS Pipelines

This chapter introduces you to the use of TCP/IP with CMS Pipelines. Basic TCP/IP education is beyond the scope of this book. Refer, instead, to the TCP/IP books listed in the bibliography at the end of this book.

Introduction

At first you may question why you would want to use TCP/IP with CMS Pipelines. There are very good reasons to do so.

The TCP/IP stages allow you to use CMS Pipelines to send data to and receive data from other users. You can build clients that will operate with existing servers and build new servers to accommodate existing and new clients. z/VM users already have the capability to use popular TCP/IP applications, such as Telnet, FTP and others, from within REXX programs and CMS Pipelines. The new TCP/IP stages in CMS Pipelines offer a new interface between CMS Pipelines and the socket library of the TCP/IP virtual machine. New stages are also provided that support the UDP (User Datagram Protocol) stage. This allows z/VM users to create their own TCP/IP network applications that are known as **clients** and **servers**. This can be done without extensive knowledge about socket level communications of TCP/IP.

The socket library supports two protocols for network applications:
- UDP (user datagram protocol), which allows the flow of datagrams between network hosts.
- TCP (transmission control protocol), which sets session communications between two network hosts, one local and one remote, and establishes two data streams or TCP pipelines, to and from the local and remote hosts.

Using the CMS Pipelines interface, the z/VM user can put CMS Pipelines records into and retrieve data from TCP data streams or UDP datagrams. This allows z/VM users to create TCP/IP client/server applications of any desired complexity.

There are two basic methods of accomplishing this:
- Use the CMS Pipelines UDP stage to put CMS Pipelines records into network datagrams and then get datagrams from the network and put them into CMS Pipelines using the UDP communication protocol of TCP/IP. The CMS Pipelines IP2SOCKA and SOCKA2IP stages assist the UDP stage with creating datagrams.
- Use the CMS Pipelines TCPCLIENT, TCPLISTEN and TCPDATA stages to create network data streams from CMS Pipelines records. And vice versa, you can get network data and convert it into CMS Pipelines records. The HOSTBYADDR, HOSTBYNAME, HOSTID and HOSTNAME stages assist the TCPCLIENT, TCPLISTEN and TCPDATA stages.

It is important to note that when using TCP/IP with CMS Pipelines, network data flow and the way records flow in a pipeline are similar. Unlike CMS Pipelines records, network data streams are not split into records. They are transmitted in groups of varying size, which is determined by the TCP/IP interface. Network data streams flow in both directions between network host computers, which differs from the way data flow through CMS Pipelines stages. The TCPCLIENT, TCPLISTEN and TCPDATA stages help to exchange network data and pipeline records. These stages will also connect a local host to a remote host and establish a session with it

for further data transfer through CMS Pipelines. As a result, z/VM applications can use this interface to create network servers that develop requests composed of a stream of records rather than a stream of bytes and create network clients that send such record requests.

Network communication in TCP/IP is not made from application to application, but rather from one host with its specific network address to another host. The locations of network hosts are specified as IP (internet protocol) addresses. Hosts may also have names that map to IP addresses. The CMS Pipelines TCPCLIENT, TCPLISTEN, TCPDATA and UDP stages can use either a host name or the IP address of a host.

The recipient of TCP/IP data, which is the application running on a network host, is "listening" and "reading" data from a socket bound to a specific receiving port number. Symmetrically, the application running on a network host can "write" data into a socket bound to a sending port number. An application that runs CMS Pipelines can become a client of a network server by using the TCPCLIENT stage. It specifies the server's IP address and a port number that the server is using for listening and reading. A network client connects to the listening server and then exchanges data with it. The pipeline in this case provides records for TCPCLIENT to send to the server and retrieves records received from the server. TCPCLIENT connects to the server and converts records to and from the data stream that the network server develops. Unless a network server has already started and bound to its listening port, a client should not start. If it does, the request to connect is rejected.

Figure 266 shows how a VM host becomes a client:
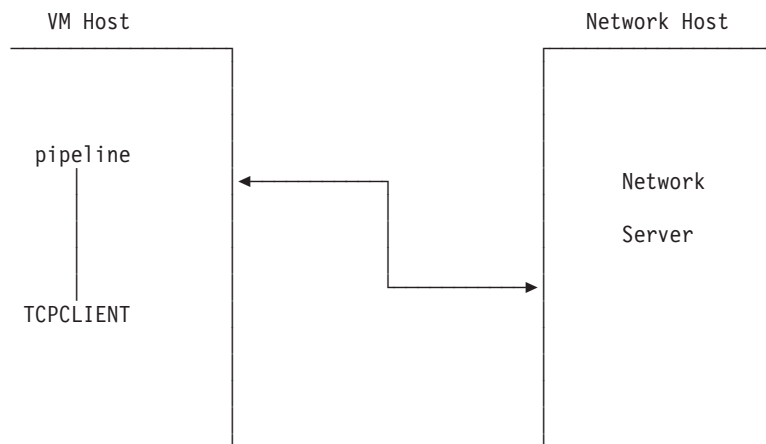
Figure 266. VM Host as a Client

An application can be a server in a network by using CMS Pipelines with the TCPLISTEN and TCPDATA stages. TCPLISTEN must be provided with a port number to which it will bind the socket to accept a connection request from a client. It will then create a special record for further communication so the TCPDATA stage can pick this record and become responsible for communicating with the client. Then TCPLISTEN is free to listen for more clients.

```
         pipeline


       TCPLISTEN


       TCPDATA
```

Figure 267. VM Host as a Server

Creating a Network Client

The TCPCLIENT stage has two required operands:

IPaddress
> This is either the host name of the server or a dotted-decimal IP address.

portnumber
> This advises which port number the server is bound to on its host.

It is assumed that a client knows which server it is going to connect to so it knows the location of the server in the network and the port number that the server is using.

The TCPCLIENT optional operands are used to specify which way the records should be sent and received from the network, and how to convert them to and from the stream of bytes. These optional operands are:

ONERESPonse
> Specifies that every input pipeline record is to be sent separately to the server, after which the client should wait for the response from the server. If this optional operand is not used, the client will not wait for the response and will keep sending input records as they arrive. The client will also retrieve network data, convert the data into records and put them into the output stream as network data arrive.
>
> **Note:** TCP does not guarantee arrival of all bytes of a single client message in one group. So unless a server has the capability to collect split messages back into one group, this option should not be used.

GREETING
> Specifies to the client that the first message received from the server is not a response, but rather a "greeting", which is a message sent immediately after the connection is established and before the first record is sent to the server. If GREETING is used when a server does not have a greeting to send, two conditions can occur:

- If ONERESPONSE is used, the client will wait for a greeting, even if one does not exist, until a time-out occurs that will disconnect the client from the server.
- If ONERESPONSE is not used, the client will not wait for the greeting, but it may incorrectly treat the first response from the server as a greeting message.

SF

Specifies that the client should convert the record into a data block before sending the pipeline record to the server and deblock the data TCPCLIENT receives; the length precedes the block in a two-byte prefix. The block length includes the length of the prefix.

SF4

Is similar to SF, but the block length is put in a four-byte prefix.

Note: Using SF or SF4 allows a client to help a server collect a split message in case TCP fails to deliver it in one group of data. Then it is the responsibility of the server to collect the split message.

DEBLOCK

Specifies how the pipeline records should be cut out of the network data stream that is received from the server. If the response from the server consists of lines that end with some specific code, then the DEBLOCK operands CRLF, LINEND and STRING enable the client to find this code in order to cut the records. If the response is expected as a number of blocks of variable length, then the conversion into a record is made from each block. The block is preceded by a two- or four-byte prefix providing the block length. The prefix length is included in the total block length. The DEBLOCK operand **SF** is used to inform the client to cut the records out of blocks when the prefix is two bytes long. The **SF4** operand is used when the prefix is four bytes long.

The communication related optional operands are used to control the connection. These options are:

TIMEOUT *number*

Specifies the time limit in seconds for the client, after which it will disconnect unless the server sends the response or disconnects.

LINGER *number*

Forces the client to delay closing the connection after the last record from the client is sent. This allows the client an opportunity to get the last response from the server. If not specified, the effective value will probably be 0 (zero).

KEEPALIVe
OOBINLINe
REUSEADDR

These options allow the experienced programmer to turn ON the TCP/IP communications options, which are used for writing reliable network applications. For more information about the TCP/IP communications options, refer to *z/VM: TCP/IP Programmer's Reference*, SC24-6239.

GETSOCKName
LOCALIPaddress *IPaddress*
LOCALport *portnumber*
USERID *tcpipuserid*

These are used for tracing and debugging needs.

Note: The primary input and output streams are used by TCPCLIENT if the TCP connection remains satisfactory. If an error occurs and the server disconnects, other errors occur in the TCP connection, or if the client times out, the TCPCLIENT stage uses its secondary output stream (if it is defined and connected) to report the problem with either a CMS Pipelines or network error message.

Example of TCPCLIENT Sending Records to an ECHO Server

For this example, assume that an ECHO server exists on the network and a program called ECHONET has been started with the default parameters in effect. (Refer to Appendix D, "ECHONET C Source Code," on page 271 for the C source code for this ECHONET server.) It runs on a host named `eagle.company.com`, and is listening on port 45678.

This ECHO server can be compiled and run on a z/VM or AIX® system to be called by the TCP/IP client. Figure 268 is a client REXX program that uses the ECHO server. The ECHOC user-written stage reads text from a command line and sends it, word-by-word, to the ECHONET server, expecting every word to be echoed.

```
/*  ECHOC REXX */

host     = 'eagle.company.com'  /* Server assumed host name    */
srv_port = 45678                /* Provide default server port # */

parse arg text  "(" port .      /* Get port # from server      */
if text='' then exit            /* No text; no work to do      */
if port='' then port=srv_port   /* If no port #, use the default */

text = text "EOD"               /* Concatenate EOD to end of text*/

'pipe (endchar ?)'          ,
'   var text'               ,   /* Get text to echo            */
' │ split'                  ,   /* Place each word in a record  */
' │ q: tcpclient' host port,    /* Have client set up connection */
'     'greeting'            ,   /* Expect greeting message      */
'     'oneresp'             ,   /* Expect one response per input */
' │ drop first 1'           ,   /* Drop a greeting message      */
' │ drop last 1'            ,   /* Drop end-of-data token       */
' │ console'                ,   /* Display the echoed records   */
' ?'                        ,   /* Provide errors to REXX       */
' q:'                       ,   /* Obtain error                */
' │ append literal',           /* Provide a blank if no error  */
' │ var errno'             ,   /* Save error in a variable     */

if (  errno <> '',              /* If errno is present and      */
   & left(errno,4) <> '0 OK' ) /*  is not "0 OK"               */
  then say 'connect err:' errno /*  display it                 */

Exit                            /*  Exit                       */
```

Figure 268. ECHOC REXX User-written Stage on the Client

After the text that is to be sent is read from the command line, and optionally, the port of the server set, an end-of-data token (EOD) is added at the end of the text. Then the text is split to create one pipeline record for each word. All records are sent to the listening server using the TCPCLIENT stage. The client will wait for a response after every record is sent because the ONERESPONSE operand was used. It is important to note that neither the SF, SF4 nor DEBLOCK operands were

used. So every piece of data returned is converted into a record and is put into a 'drop 1' stage, which is used to eliminate the greeting statement from the server and read just echoes. The next stage, 'drop last', eliminates the last end-of-data record that was returned. The echo records are displayed and then the client terminates. If an error occurs and the TCPCLIENT stage terminates prematurely, an error message is sent to the secondary output stream connected at the label "q:" and is displayed.

If the client was started in this way:

```
echoc This is a line of text.
```

The following is displayed on the console of the client:

```
This
is
a
line
of
text.
Ready;
```

Note that the client program is a simple application that sends records to a server and converts bytes it receives into records. But remember, data in a network are just streams of bytes. The arrival of these records in order is guaranteed by the use of the TCP protocol. However, there is no guarantee that these records arrive on the server grouped in the same way when they were sent by the client. There is also no guarantee that the data bytes from the server arrive at the client grouped in the same way that they were sent by the server. Groups of records may arrive contiguously or may be separated. In other words, this client is not a reliable application. The ONERESPONSE operand was used in Figure 268 on page 203 so each record sent would be echoed by the server before the next record is transmitted. (This is very important for clients in general and for this example in particular.) The fact that the client waits to perform a read after each record is sent allows the response to be received and converted back into a record before the next record is sent to a server. This guarantees that records in a network do not arrive contiguously. However, there is still no guarantee that client records will not be split. That is why network clients should send requests in blocks of a predetermined size or separated with a special code. The TCPCLIENT operands SF and SF4 are used to create blocks of a predetermined size before sending and deblocking data after it is received. Thus, it is assumed that the server has the capability to deblock such data upon receiving it, and then blocking that data before it is sent. Alternatively, the user can imbed code separators between requests if the use of SF or SF4 is not sufficient, nor desirable.

It may be beneficial to change this ECHOC example in Figure 268 on page 203 by removing the ONERESPONSE operand so the client will not wait for every individual response. If this is done, ensure that the LINGER operand is added to the TCPCLIENT stage providing a specified number of seconds, such as `linger 3`. This will help the modified client to wait for all responses to arrive before closing the connection. Also, ensure that the `drop last` stage is removed because the end-of-data token will probably not remain as a separate record. It is also recommended that the TIMEOUT operand be used with the TCPCLIENT stage because the default timeout value may not be satisfactory.

Start the client again by entering:

```
echoc Check if returned data arrives contiguously.
```

The client output in this case is not as predictable. It could look like this:

```
Checkifreturneddataarriv
escontiguously.EOD
```

Or like this:

```
Checkifreturneddataarrivescontiguously.EOD
```

The SPLIT stage was used in the ECHOC REXX user-written stage (Figure 268 on page 203) to define each word of text as a pipeline record. However, when ONERESPONSE is not used, the client records are no longer perceived as individual pipeline records, but instead, as a stream of bytes. Client records are converted into network bytes while traveling through the network. So responses from the server may be received as one large record or smaller ones with no predetermined way of how returned data is to be converted into pipeline records.

Note: It is recommended that the requests from the client should always be put into blocks, or that special code separators are used to ensure the requests are recognized.

Add the SF operand to the TCPCLIENT stage and ensure ONERESPONSE is not used. After all modifications, the client ECHOC user-written stage appears as in Figure 269 on page 206:

```
/*  ECHOC REXX */
 host     = 'eagle.company.com'  /* Server assumed host name     */
 srv_port = 45678                /* Provide default server's port */

 parse arg text  "(" port .      /* Get port # from server       */
 if text='' then exit            /* No text; no work to do       */
 if port='' then port=srv_port   /* If no port #, use the default */

 text = text "EOD"               /* Concatenate EOD to text tail  */

 'pipe (endchar ?)'           ,
 '   var text'                ,  /* Get text to echo             */
 ' | split'                   ,  /* Place each word in a record  */
 ' | q: tcpclient' host port, ,  /* Have client set up connection */
       'greeting'             ,  /* Expect greeting              */
       'SF'                   ,  /* Block/deblock data           */
       'linger 3'             ,  /* Linger on closing for 3 sec  */
       'timeout 10'           ,  /* Set timeout                  */
 '| drop first 1'             ,  /* Drop greeting                */
 '| drop last  1'             ,  /* Drop end-of-data token       */
 '| console'                  ,  /* Display the echoed records   */
 '?'                          ,  /* Provide errors to REXX       */
 'q:'                         ,  /* Obtain error                 */
 '| append literal'           ,  /* Provide a blank if no error  */
 '| var errno'                ,  /* Save error in a variable     */

 if (  errno <> '',             /* If errno is present and       */
    & left(errno,4) <> '0 OK' ) /* is not "0 OK"                */
   then say 'connect err:' errno /* display it                   */

 Exit                            /* Exit                         */
```

Figure 269. Enhanced ECHOC REXX User-written Stage on the Client

Stop and restart the ECHO server with the SF operand this time. Run the new ECHOC user-written stage and notice that every record is returned. For example, start ECHOC by entering:

```
echoc The text of line is separated by words, guaranteed.
```

The client output will look like this:

```
The
text
of
line
is
separated
by
words,
guaranteed.
```

In the examples above, it was assumed that the ECHO server was running on the same platform as the client. So the server understood the end-of-data token (EOD) from the client. In fact, the greeting message from the server is the record that the client can use as a token to request the end of communication. A change could be made to ECHOC that would accept the greeting from the server and use it as an end-of-data token as the last word sent after the text from the client. Now the client is platform independent because the text from the client is echoed without translation and the client gets the end-of-data token that the server provides. This final version is found in Figure 270 on page 207.

```
/*  ECHOC REXX */
 host     = 'eagle.company.com'  /* Server assumed host name      */
 srv_port = 45678                /* Default server's port #       */

 parse arg text  "(" port .      /* Get port # from server        */
 if text='' then signal Exit     /* No text; no work to do        */
 if port='' then port=srv_port   /* If no port #, use the default */

 /* Set pipeline that establishes connection; sends split text
     followed by end-of-data token received from the server as a
     greeting message, receives echoes, and displays them.        */

 'pipe (endchar ?)',
  'var text',                    /* Get text to echo             */
  '| split',                     /* Place each word in a record  */
  '|s:fanin',                    /* Get text, then end-of-data    */
  '| q: tcpclient' host port,    /* Have client set up connection */
      'greeting',                /* Greeting is expected          */
      'sf',                      /* Get records blocked/deblocked*/
      'linger 3',                /* Linger on connect closing     */
      'timeout 10',              /* Disconnect if 10 sec idle     */
  '| f:fanout',                  /* Fan to f: to pick greeting    */
  '| drop first 1',              /* Drop greeting                 */
  '| drop last  1',              /* Drop end-of-data token        */
  '| console',                   /* Display the answer            */
  '?',                           /* Provide errors to REXX        */
  'q:',                          /* Obtain error                  */
  '| append literal',            /* Provide a blank if no error   */
  '| var errno',                 /* Save error in variable        */
  '?f:',
  '| take 1',                    /* Pick just greeting to use     */
  '| elastic',                   /*  it as end-of-data token      */
  '|s:'                          /*  after 'text' records         */

 /* Report if problem occurs. */

 if (  errno <> '',             /* If errno is present and       */
     & left(errno,4) <> '0 OK' ) /*  is not "0 OK"                */
    then say 'connect msg:' errno /*  display it                  */

Exit:
  exit rc
```

Figure 270. Further Enhanced ECHOC REXX User-written Stage on the Client

In this modified client, the FANIN stage occurs immediately after the text is split. This takes all TEXT records from the primary input stream. When the primary input stream disconnects, all TEXT records are taken from a secondary input stream that is fed from the ELASTIC stage. The only record that ELASTIC sends to FANIN is the first record received from TCPCLIENT (the TAKE stage is used), which is a greeting record that was sent from a server. The client output does not differ from the one presented above, but this time the client can operate with servers running on non-EBCDIC platforms.

Note: If the client must translate transmitted data to network code, this is easily accomplished by adding the stage `'xlate 1-* from 1047 to 819'` before TCPCLIENT and `'xlate 1-* from 819 to 1047'` after TCPCLIENT.

Creating a Network Server

The most common way to create a network server is to perform two separate tasks:

- Establish a communication session with the client.

• Provide data exchange with the client.

CMS Pipelines presents two stages to build a server that performs these tasks:

1. TCPLISTEN listens to a port and when the connection request is accepted, establishes the session with the client and then passes on session credentials to the next stage.
2. TCPDATA reads the session credentials, sustains the session and communicates with the client until the client stops or disconnects.

TCPLISTEN continues to wait for new client requests, allowing one server to serve multiple clients.

TCPLISTEN has one required operand and optional operands that are used to perform the first task—establishing sessions with clients. The only required operand is a *port number* to which the server will be listening. This port number must be known by clients before they are started. Some of the most popular servers use "well-known" port numbers. Other, less popular servers must use port numbers from a pool of available port numbers that do not have conflicts. One way to assign a port number to a server is to have it assigned by TCP/IP each time the server is started. This port number is then transmitted to and used by the clients. Any port number can be assigned to the server unless other servers on the same host use that same port number. Using port number 0 (zero) forces the TCP/IP virtual machine to assign a port number from the pool of available ports, which will be a different port number than those used by other servers on the same host.

All other operands of TCPLISTEN are optional:

BACKLOG *maxpending*
> Specifies to the server the maximum number of clients that can simultaneously establish a connection with the server without being rejected.
>
> **Note:** The implementation of TCPLISTEN provides an unlimited number of simultaneous connections to a server. If BACKLOG *maxpending* is not specified, the default value is 10.

GETSOCKName
> Specifies that a special record is put into the primary output stream of the TCPLISTEN stage that describes the initial binding of the server after the server starts and before the first client connection request is accepted. This is in binary, not in EBCDIC. It is information that includes a bound port number, which can be very important if a user starts a server the way the port is assigned by TCP/IP. Using this option helps to find out the port number assigned so it can be transmitted to customers who will be using the server.

Note: Other TCPLISTEN optional operands are used for tracing and debugging.

TCPLISTEN has two very important features:

• As soon as the client connection request is accepted and a session between partners is established, TCPLISTEN writes a special 84 byte record to its primary output stream that starts with the *pipetcp* token, so it is easily recognized in a stream of records. This record, which describes the session that is established, is created for the TCPDATA stage to take control of the session for further communication. TCPLISTEN is now ready to listen for the next client and the TCPDATA stage assumes responsibility for exchanging data with the connected client.
• TCPLISTEN is always the first stage in a pipeline. So the only way to stop it is to disconnect its primary output. The other reasons why it stops are due to network

errors, but this is beyond the control of the user unless PIPMOD STOP was issued or if a corresponding record was passed into PIPESTOP.

All operands of the TCPDATA stage are optional (previously described under the TCPCLIENT stage). In fact, running the TCPCLIENT and TCPDATA stages represent communication peers in the network. Therefore, they use the same parameters; although the server cannot disconnect from an idle client in a timeout period of its choice and cannot specify a greeting from the client.

The TCPDATA stage has one very important feature: it consumes records of two different types from its primary input stream:

• One record with a *pipetcp* token that TCPLISTEN produces.
• Records that are destined to be sent over the session to a client.

The way the TCPDATA stage communicates with its client is as follows:
1. Receives data from the client over the session.
2. Deblocks the data into records and puts them onto its primary output stream.
3. Reads the records from its primary input stream and then writes the response back into network over the session.

Therefore, the TCPDATA stage must be put in a data loop where records can be serviced after being received by TCPDATA and before they are returned to the network by the same TCPDATA stage.

Figure 271 shows the data flow for the server:
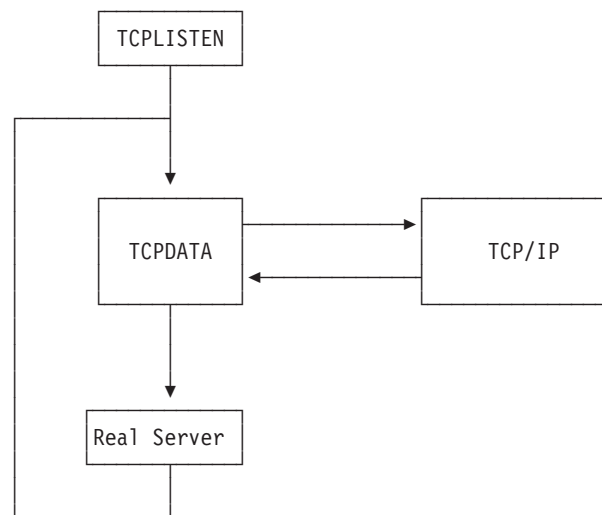
Figure 271. Data Flow for the Server

The examples below describe how the server must find a way to disconnect TCPDATA from TCPLISTEN after the *pipetcp* token record is received and changes the TCPDATA input stream to receive records from the "Real Server" (shown in the diagram above).

Note: If a server is only receiving data from clients, a data loop is not required, but servers seldom just consume data.

A Simple Server

A server must be written using CMS Pipelines that is capable of replacing the ECHONET server used above. As with ECHONET, the server will listen on a port for a connection and then it will echo the data that is received. For simplicity, the first server has been written to take only one connection request and terminate when it is done with the client.

For clarity of explanation, the server has been written to operate using two parts, ECHOS (Figure 272) and ECHOD (Figure 273 on page 211). ECHOS starts the job and just listens to the port in order to establish the connection as soon as the connection request is received and accepted. ECHOD then takes control of the session and exchanges data with the client.

Note: Note that the client ECHOC presented above is capable of communicating with ECHOS.

```
/* ECHOS EXEC - This server echoes the information that is sent to it.
   This is done by establishing a TCP session between a client and this
   server.

   Note:
   If not provided, the port number will be requested from TCP/IP   */
 parse arg port .

 if port='' then port=0            /* Get port # from TCP/IP       */

'pipe (endchar ? name echoS)',
 'tcplisten' port,                /* Obtain the initial request    */
    'getsockname',                /* Requests 2nd record with port#*/
 '| take 2',                      /* Ensure only one request       */
 '| s: locate /pipetcp/',         /* Picks out token record        */
 '| echoD',                       /* Take socket and echo data     */
 '?',                             /* Displays port number          */
 's:',                            /* Obtains port number           */
 '| specs x0000 1 3-4 3',         /* Puts 2 zero bytes before port#*/
 '| specs 1-4 c2d 1',             /* Changes port # into characters*/
 '| strip',                       /* Removes leading blanks        */
 '| literal Started on port',     /* Places text pipeline          */
 '| join 1',                      /* Combines the two records      */
 '| console',                     /* Displays it                   */
```

Figure 272. ECHOS EXEC

ECHOS has only one argument, which is optional. That argument is the port number on which the server is to listen. If the argument is not present, TCP/IP assigns the port number. In either case, ECHOS displays the port number to which it is listening so the client can be informed of what port number it should use.

The first CMS Pipelines stage in the ECHOS EXEC is TCPLISTEN. It requires a port number as its first argument. If the port number is 0 (zero), TCP/IP assigns an unused port number. In the example, a TCP/IP or user-assigned port number is possible. To display the port number to which ECHOS is listening, the optional operand GETSOCKNAME is specified. It causes TCPLISTEN to put the first record, which contains the port number that is being used, on its primary output stream immediately after the server starts. This record is converted to decimal and displayed. The second record contains *pipetcp* and it will be passed to the TCPDATA stage that has been placed in ECHOD.

TCPLISTEN terminates when its output stream is disconnected. Therefore, to limit the application to only one response, a TAKE 2 stage was added. This terminates after both of the output records from TCPLISTEN are received, which causes TCPLISTEN to terminate. TAKE 2 also provides an end-of-file specification that ECHOD needs.

The LOCATE stage separates the record with the token *pipetcp* from the record that contains the port number. This allows the exec to pass this record to ECHOD, and the other record, the port number record, to the secondary output stream designated by the label `s:`.

When the port number record arrives at `s:`, the two SPECS stages extract the port number and convert it to characters. This conversion produces leading blanks that make the display look strange. The STRIP stage eliminates these blanks. The LITERAL stage introduces the text that is to be displayed along with the port number so the JOIN stage can combine them. The CONSOLE stage can then display the result.

The main purpose of the ECHOS EXEC was to pass the record containing a token to the ECHOD user-written stage. The ECHOD user-written stage takes the connection with the client, receives data from the client, and echoes the data back to the client in the loop, which was explained above.

```
/* ECHOD REXX - This is the TCPDATA portion of the server.      */
 greeting = 'EOD'                    /* Closing statement this uses  */

'callpipe (endchar ? name echoD)',
  '*:',              /* Obtain the token record      */
  '| append literal' greeting,   /* Send greeting                */
  '| i: fanin',                  /* Allow reply to be sent out   */
  '| q: tcpdata',                /* Get and send data to client  */
  '       sf',                   /* Make them blocked/deblocked  */
  '| hextype',                   /* Display it in both char & hex */
  '| elastic',                   /* Prevent stalling             */
  '| i:',                        /* Send to TCPDATA for client   */
  '?',                           /* Handle errors from TCPDATA   */
  'q:',                          /* Get error message            */
  '| append literal',            /* Provide a blank if no error  */
  '| var errno'                  /* Save error in a variable     */

 if (errno <> '') then           /* Check if error is present    */
   ername = word(errno,2)
   if (ername <>  ECONNRESET) then
     say 'error no:' errno       /* Report other errors          */
   else
     say 'Server finished after client connection reset.'
 exit
```

Figure 273. ECHOD REXX

This CMS Pipelines stage accomplishes this by using the TCPDATA stage. The TCPDATA stage consumes two different types of input records on its primary input stream. The first record must be the token that was created by TCPLISTEN. Do not modify this record. The greeting is sent first. The other records that arrive at the primary input stream of TCPDATA will be sent over the session to the client. FANIN is used to accomplish the sequencing of records to TCPDATA. This works because FANIN will read from only one input stream until that stream reaches end-of-file. Then it will read from another input stream.

The TAKE 2 stage of ECHOS is needed by ECHOD to cause FANIN to disconnect from its one input stream so that it can start reading its second input stream. This completes the loop in the diagram above and the server can start receiving data and sending responses to the network.

When ECHOD is called, a token is passed to it from TCPLISTEN. That record enters CALLPIPE at the connection stage, *:. It is passed to TCPDATA as the first record. Having the token requirement met, TCPDATA is now ready to receive data that will be sent to the client. The APPEND LITERAL statement must receive an end-of-file before it writes the greeting. This is provided by the TAKE 2 stage in ECHOS EXEC. This TAKE 2 stage not only terminates TCPLISTEN, but it also terminates the LOCATE stage that followed it. LOCATE in turn severs the input stream to ECHOD REXX, which passes it onto the primary input stream of the APPEND stage. This greeting is the first record sent to the client providing the information that must be returned to end the conversation. FANIN receives an end-of-file on its primary input stream and starts reading from its secondary input stream. When FANIN receives this end-of-file, it starts reading data from its secondary input stream and passing that data to TCPDATA.

Figure 274 on page 213 is an enhanced version of the diagram in Figure 271 on page 209. Notice that the secondary input stream is attached to more stages that are tied to the primary output stream of TCPDATA. This enables data to flow from the client to TCPDATA, then through the other stages, and then back to TCPDATA to be sent back to the client. This is how the echo is accomplished.

```
                    ┌──────────┐
                    │    *:    │
                    └──────────┘
                         │
                         ▼
                    ┌──────────┐
                    │  append  │
                    │ greeting │
                    └──────────┘
                         │                    │
                         ▼                    ▼
                    ┌──────────────────┐
                    │    i:fanin       │
                    └──────────────────┘
                         │
                         ▼
                    ┌───────────────────┐      ┌─────────────┐
                    │                   │─────▶│             │
                    │   q:tcpdata       │      │   TCP/IP    │
                    │                   │◀─────│             │
                    └───────────────────┘      └─────────────┘
                         │            │
                         ▼            ▼
                    ┌─────────┐  ┌─────────┐
                    │ hextype │  │ literal │
                    └─────────┘  └─────────┘
                         │            │
                         ▼            ▼
                    ┌─────────┐  ┌─────────┐
                    │ elastic │  │  var    │
                    └─────────┘  └─────────┘
                         │
```
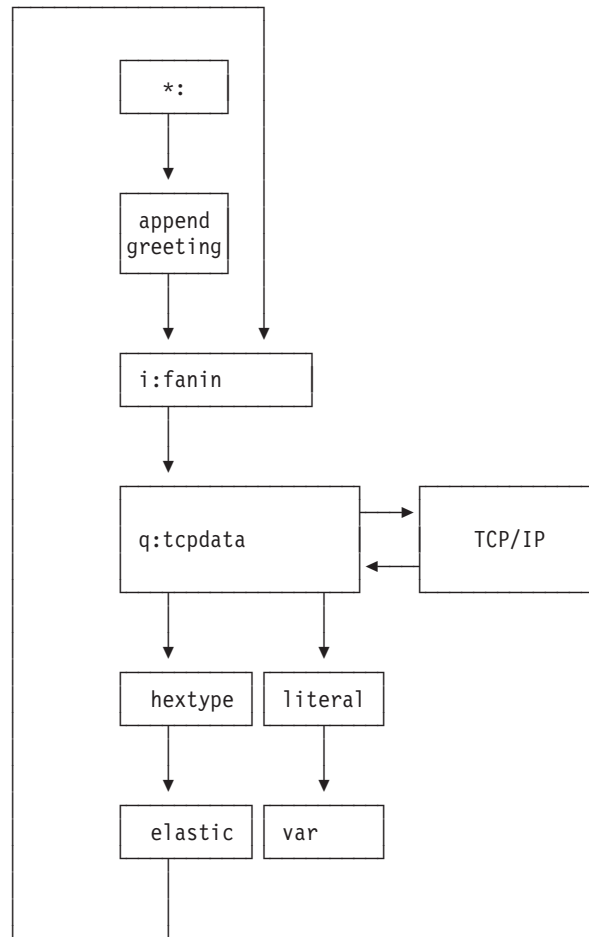
Figure 274. Complete Data Flow for the Server

When the data leaves TCPDATA, it is sent through a user-written stage called HEXTYPE (Figure 275 on page 214). HEXTYPE passes any record it receives to its output without changing it. It also displays the contents of the record in both character and hexadecimal forms. HEXTYPE is shown in Figure 275 on page 214:

Note: The HEXTYPE user-written stage is only used for tracing the data flow in the server.

```
/* ********************************************************* */
/* HEXTYPE REXX Pipeline User-written stage                  */
/*                                                        */
/* Being transparent to records, this stage prints them and  */
/* the hexadecimal representation on a console.           */
/* Note: This stage terminates when the primary IN or OUT    */
/* streams are not defined or disconnected, so it can be used */
/* between stages to trace the record flow.              */
/* ********************************************************* */

h='1de8'x; l='1d60'x              /* Set highlight codes     */

 do forever                       /* Read in record loop     */
   'STREAMSTATE INPUT 0'          /* Check prime IN state     */
   if (rc > 0) then do            /* Does OUT need verifying? */
      'STREAMSTATE OUTPUT 0'      /* Check prime OUT          */
      if (rc > 8) then leave      /* Leave if disconnected    */
   end                            /* Otherwise, keep looping  */
   'PEEKTO  record'               /* Peek next record         */
   if (rc <> 0) then leave        /* Leave if trouble         */
   dsp_line  = ''                 /* Compose for displaying    */
   do j = 1 to length(record)     /*  in every other column   */
     dsp_line = dsp_line || substr(record, j, 1) || " "
   end
   say h dsp_line l               /* Display the IN record    */
   say " "c2x(record)             /*  and its hex form         */
   'OUTPUT'  record               /*  and put it to OUT stream*/
   if (rc <> 0) then leave        /* Leave if trouble         */
   'READTO  record'               /* Consume it finally        */
 end /* do forever */

Exit  rc*(rc <> 12)               /* Convert EOF back to RC=0 */
```

Figure 275. HEXTYPE REXX

When records are received from the primary output stream of the TCPDATA stage, HEXTYPE may not be ready to use them on its primary input stream yet because it may still be busy with the previous output. A buffer must be created that will consume records until the next stage is ready for data. To solve this, the CMS Pipelines ELASTIC stage is used. ELASTIC holds as many records as necessary to prevent the pipeline from stalling.

ECHOD keeps reading any new data arriving and cannot yet tell the end-of-data token from other client records sent. So when the client sends this last token and disconnects, TCPDATA gets the ECONNRESET signal and puts the ERRNO value to label q:. This is where the VAR ERRNO stage places the error in a REXX variable called ERRNO and analyzes it with REXX code. To allow for the case when an error does not exist, a blank line is appended. This allows the REXX variable to have a null value if no error is present. When ERRNO contains an expected ECONNRESET value, successful termination is reported. If any other connection error occurs, the ERRNO value is reported as is.

Here is an example of running these execs:

1. From the server user ID, enter:

   echos

   Note: If a port number was assigned to this application, then this could have been entered as:

   echos *portnumber*

The ECHOS EXEC immediately displays the port number provided by TCP/IP (or the port number that was entered):

```
     Started on port 9882
```

2. From the client user ID, enter:

```
echoc This is a line of text ( 9882
```

Note that the port number that was entered was the port number that ECHOS provided.

The following is displayed on the console of the server:

```
      T h i s
     E38889A2
      i s
     89A2
      a
     81
      l i n e
     93899585
      o f
     9686
      t e x t
     A385A7A3
      E O D
     C5D6C440
     Server finished after client connection reset.
     Ready;
```

The following is displayed on the console of the client:

```
      T h i s
      i s
      a
      l i n e
      o f
      t e x t
     Ready;
```

Notice that both execs terminated. The client ECHOC EXEC terminated because it accomplished the work and then lingered for three seconds. The server ECHOS EXEC terminated because it was set up to handle only one client and then to terminate. This was done by the TAKE 2 stage. Notice also that the same client (ECHOC) was used with the new server that was used in a previous example with the ECHONET server written in C.

A Way to Stop One Client/Server Conversation

The normal ways to stop a server are to use `PIPMOD STOP` or to pass a record to a PIPESTOP stage. However, if just one conversation between a client and a server is to be terminated, the following describes a way to accomplish that.

To contact a client, the server sends the client a record. The client uses this record as the last record it sends back to the server. For explanatory purposes, this record will be called the "greeting record".

The ECHOD REXX user-written stage (Figure 273 on page 211) can be altered to quit when this greeting record is received. For ECHOD REXX to function similar to the server written in C code (Appendix D, "ECHONET C Source Code," on page 271), it must also write the greeting back to the client before it quits. The new logic is added to the previous diagram (Figure 274 on page 213) between the ELASTIC stage and the secondary input stream of FANIN.

Using TCP/IP with CMS Pipelines

A revised diagram with this new logic added can be found in Figure 276:

```
        ┌──────┐
        │  *:  │
        └──────┘
            │
            ▼
        ┌──────────┐
        │ append   │
        │ greeting │
        └──────────┘
            │              │
            ▼              ▼
        ┌─────────────────────┐
        │ i:fanin             │
        └─────────────────────┘
            │
            ▼
(F)  ┌──────────────┐      ┌──────────┐
     │ q:tcpdata sf │─────▶│  TCP/IP  │
     │              │◀─────│          │
     └──────────────┘      └──────────┘
         │          │
         ▼          ▼
(E)  ┌────────┐ ┌─────────┐
     │hextype │ │ literal │
     └────────┘ └─────────┘
         │          │
         ▼          ▼
(E)  ┌────────┐ ┌─────────┐
     │elastic │ │  var    │
     └────────┘ └─────────┘
(A)      │
         ▼
     ┌──────────────────────┐
     │ l:locate /'greeting'/│
     └──────────────────────┘
         │  (D)          │
         ▼               │
(C)  ┌──────────────┐    │
     │ d:not fanout │    │
     └──────────────┘    │
        │      │    (B)  │
        ▼3     ▼2        ▼1
     ┌──────────────────────┐
     │ g:gate               │
     └──────────────────────┘
              │      │
              ▼      ▼
        ┌──────────────┐
        │ m:faninany   │
        └──────────────┘
```
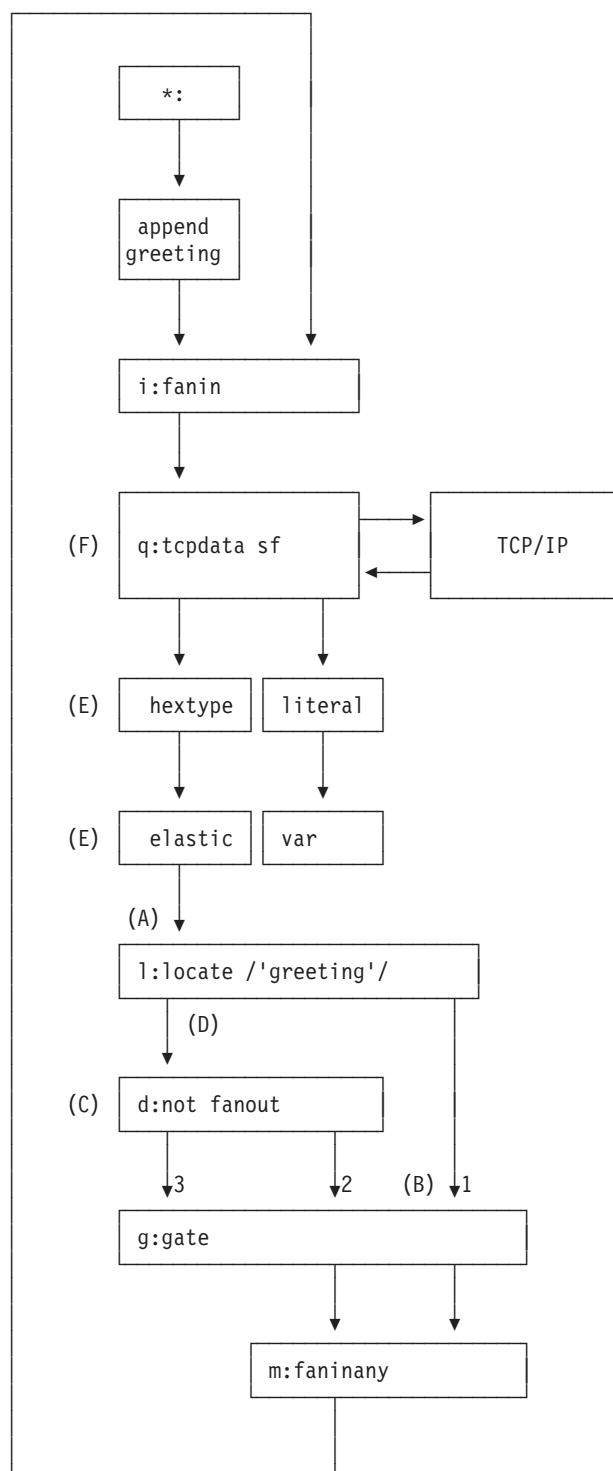
Figure 276. Data Flow for the Server to End a Conversation

The flow of this pipeline is as follows; refer to the alphabetic characters in Figure 276:

(A)   The first step is to use the LOCATE stage to find all records.

(B)   Records other than the greeting record are passed through the GATE stage, collected by FANINANY, and then given to the FANIN stage.

(C)   FANOUT makes a copy of the greeting record. However, the first copy of the greeting record must be written to the secondary output stream. The NOT stage is used to switch the two output streams of FANOUT. A copy of the greeting record is sent through GATE, collected by FANINANY, and then sent to the FANIN stage. Finally, NOT FANOUT writes a record to the primary input stream of GATE and the GATE stage is closed. When GATE closes, it disconnects all of the streams that are connected to it. NOT FANOUT closes because its two output streams are disconnected.

(D)   LOCATE then closes because its two output streams are disconnected.

(E)   Likewise, ELASTIC and HEXTYPE close, which causes the primary output stream of TCPDATA to be disconnected.

(F)   TCPDATA closes.

Theory of Operation

When records leave the ELASTIC stage they are provided to the primary input stream of TCPDATA without delay. This is even true for the greeting record. However, to stop TCPDATA, its primary output stream must be severed when the greeting is presented by the ELASTIC stage. To sever the primary output stream of TCPDATA, all of the other streams derived from TCPDATA must be severed. To accomplish this, the GATE stage was added. The technique is to write the greeting record to the primary input stream of the GATE stage after all records have been sent to the secondary input stream of FANIN. All of the derivative output streams are run through the GATE stage, which is closed after the greeting record has been provided to TCPDATA. To ensure that TCPDATA has written all of its data before it closes, none of the stages between the primary output stream of ELASTIC and the primary input stream of TCPDATA can delay the record.

The ECHOD REXX user-written stage with these modifications is in Figure 277 on page 218:

```
/* ECHOD REXX */

/* This is the TCPDATA portion of the server that will stop when it
receives the greeting back from the client.                      */

 greeting = 'EOD'                  /* Closing statement - use this  */
 errno    = ''                     /* Initially, no TCP/IP error    */

'callpipe (endchar ? name echoD)',
  '*:',                            /* Obtain the token record       */
  '| append literal' greeting,     /* Send greeting (quit text)     */
  '| i: fanin',                    /* Allow reply to be sent out    */
  '| q: tcpdata',                  /* Get and send data to client   */
      'sf',                        /* Block data with length        */
  '| hextype',                     /* Display it in both char & hex */
  '| elastic',                     /* Prevent stalling              */
  '| l:locate /'greeting'/',       /* Time to quit?                 */
  '| d:not fanout',                /* Write to secondary output     */
  '| g:gate',                      /*   first, then stop streams    */
  '? d:',                          /* Write greeting here           */
  '| g:',                          /* Is gate still open?           */
  '| m:faninany',                  /* Collect for input to FANIN    */
  '| i:',                          /* Send to TCPDATA for client    */
  '? l:',                          /* Here if not greeting          */
  '| g:',                          /* Is gate still open?           */
  '| m:',                          /* Send to TCPDATA for client    */
  '?',                             /* Handle errors from TCPDATA    */
  'q:',                            /* Get error message             */
  '| append literal',             /* Provide a blank if no error   */
  '| var errno'                    /* Save error in variable        */

 if (errno <> '') then             /* If error is present           */
   say 'error no:' errno           /*   report it                   */
 exit RC
```

Figure 277. ECHOD REXX to End a Conversation

A Server that Handles Multiple Clients

There are many ways to set up the server exec to handle several clients. One has been selected to illustrate some of the principles involved.

First, create an ECHOSND EXEC by copying the ECHOS EXEC (Figure 272 on page 210) and then make two key changes:

1. To allow TCPLISTEN to accept many clients, the TAKE 2 stage that terminated it has been eliminated. However, because the TAKE 2 stage also provided an "end-of-file" to the primary input stream of FANIN this function must be provided in another place. Notice that removing the TAKE 2 stage also removed the previous method used to stop the server. To keep this simple, stop the server by using this command:

       PIPMOD STOP

   The server could also be stopped by passing a record to a PIPESTOP stage.

2. The second change is to code the server so it will establish multiple executions of the ECHOD REXX user-written stage. When this is done, there is an occurrence of ECHOD REXX for each client that connects to this server. This second change is accomplished by the TCPDEAL REXX user-written stage (Figure 280 on page 221). This takes a stage name as an argument and hands

out the new TCP/IP tokens to a specific number of waiting TCPDEALT user-written stages (Figure 281 on page 222). Each TCPDEALT user-written stage runs a copy of the ECHOD user-written stage. Therefore, the second change to ECHOSND EXEC is to add TCPDEAL ECHOD in the location where ECHOD was called before.

The general strategy remains the same, however, one TCPLISTEN stage will have several TCPDATA stages to which it can pass tokens.

Figure 278 provides the general flow for this example.
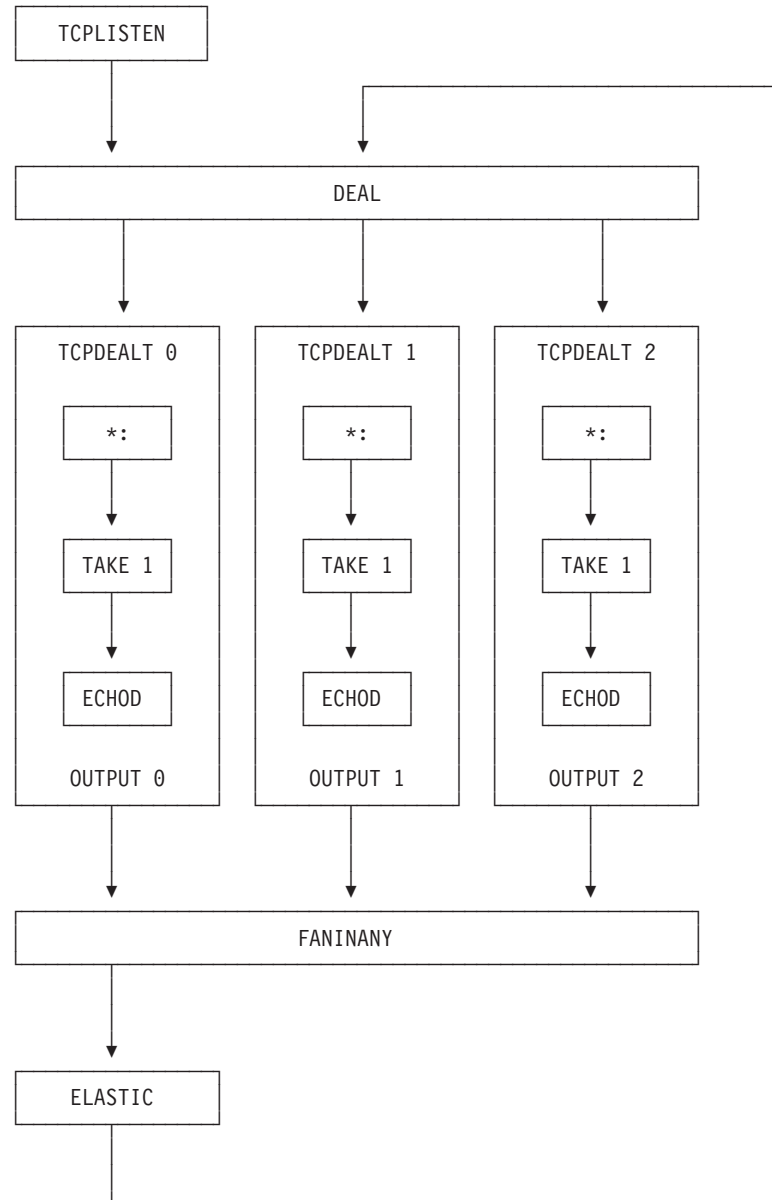
Figure 278. Data Flow for the Server to Handle Multiple Clients

1. When TCPLISTEN is contacted by a client, it writes a token to its primary output stream.

2. This token is delivered to one of three (in this example) TCPDEALT user-written stages.
3. Each TCPDEALT user-written stage passes the token to an ECHOD user-written stage and then writes its stage number to tell the DEAL stage that it is available.
4. The stage numbers are collected by FANINANY and presented to DEAL when DEAL needs them.

The resulting ECHOSND EXEC is in Figure 279:

```
/* ECHOSND EXEC */
/*
** Note: if not provided, the port number will be requested
   from TCP/IP.                                              */
 parse arg port .                   /* Allow port # as argument      */
 if port='' then port=0             /* If not, ask TCP/IP for port #*/

'pipe (endchar ? name echoSnd)',
 'tcplisten' port,                  /* Obtain the initial request    */
    'getsockname',                  /* Requests 2nd record with port#*/
 '| l: locate /pipetcp/',          /* Picks out token record        */
 '| tcpdeal echoD',                 /* Take socket and echo data     */
 '?',                               /* Displays port number          */
 'l:',                              /* Obtains port number           */
 '| specs x0000 1 3-4 3',           /* Puts 2 x00 before port number */
 '| specs 1-4 c2d 1',               /* Changes port # into characters*/
 '| strip',                         /* Removes leading blanks        */
 '| literal Started on port',       /* Places text pipeline          */
 '| join 1',                        /* Combines the two records      */
 '| console'                        /* Displays it                   */
 exit RC
```

Figure 279. ECHOSND EXEC

The TCPDEAL user-written stage allows three clients to be connected at one time, but this may be expanded; this stage uses DEAL SECONDARY. The DEAL stage passes the tokens that were produced by TCPLISTEN to waiting TCPDEALT user-written stages. Specifying SECONDARY tells DEAL that its secondary input stream will be presented with a record that contains the stream number that the next token should be dealt to. Because DEAL has three output streams, the secondary input stream must be either 0, 1, or 2. Each of the output streams of the DEAL stage are connected to a TCPDEALT user-written stage that receives the token and continues the conversation. TCPDEALT needs to know which of the output streams of the DEAL stage it is connected to and the name of the stage to which it hands each token. These are provided as arguments. Each TCPDEALT user-written stage also has the responsibility of producing the record that is sent to the secondary input stream of the DEAL stage. In fact, each TCPDEALT user-written stage must initially write one of these records to seed the process. The stream identifier records that are created by the TCPDEALT user-written stages are collected by a FANINANY stage and held in an ELASTIC stage until they are needed.

The TCPDEAL REXX code is shown in Figure 280 on page 221:

```
/* TCPDEAL REXX */
 arg echod .                              /* Obtain the stage to run  */

'callpipe (endchar ? name TCPDEAL  )',
   '|*:',                                 /* Get tokens to be handed out */
   '|d:deal secondary',                   /* Hand out tokens            */
   '|  tcpdealt 0' echod,                 /* Activate sub-server        */
   '|f:faninany',                         /* Collect available stream #s */
   '|  elastic',                          /* Hold until needed          */
   '|d:',                                 /* Tell DEAL - stream available*/
   '|  tcpdealt 1' echod,                 /* Activate sub-server        */
   '|f:',                                 /* Write stream 1 available    */
   '?d:',                                 /* Tell DEAL- stream available */
   '|  tcpdealt 2' echod,                 /* Activate sub-server        */
   '|f:'                                  /* Write stream 2 available    */
 exit rc
```

Figure 280. TCPDEAL REXX

The token from TCPLISTEN is passed to each TCPDEALT user-written stage. TCPDEALT must accept one token at a time and pass it to the TCPDATA stage in ECHOD REXX. TCPDEALT also must write a record to its primary output stream when it is ready to accept a token. This record must contain the stream to which DEAL writes the token. Therefore, the arguments must be the stream number to be written and the stage to which the token is to be passed. When TCPDEALT is started, it must first write the stream number to its output stream and wait for a token to arrive. The stream number is written using OUTPUT, and PEEKTO forces the stage to wait until a token is available. When the token arrives, CALLPIPE accepts it, passes it through the TAKE 1 stage and then to the specified stage. The specified user-written stage in this case is ECHOD REXX. The TAKE 1 stage disconnects the primary input stream to ECHOD. When ECHOD terminates, CALLPIPE terminates and the next OUTPUT writes its stream number telling the DEAL stage that it is ready for more work. PEEKTO makes TCPDEALT wait until another token is available.

The TCPDEALT REXX code is shown in

```
/* TCPDEALT REXX */
 arg number echod              /* Obtain stream number and        */
                              /*  stage to handle connection     */
'output' number              /* Write the number of this stage */
                              /*  telling DEAL that it can send   */
                              /*  tokens to this stream of TCPDEAL */
                              /*   to this stream of TCPDEAL      */
                              /* Mention this on the console      */
 say 'Server' number 'is available.'
'peekto'                      /* Wait for first token to arrive   */
 do while rc=0                /*  while there are tokens available */
  'callpipe (name TCPDELAT)', /* Process each token               */
     '|*:',                   /* Get the token                    */
     '|  take 1',             /* After one token, terminate stream */
     '|' echod                /* Run the stage for each token     */
   if rc <> 0 then exit rc    /* End if bad RC from CALLPIPE      */
  'output' number             /* Ask DEAL for more work           */
                              /* Mention this on the console      */
   say 'Server' number 'is available.'
  'peekto'                    /* Wait for another token to arrive */
 end                          /* End when DEAL terminates         */
 exit rc*(rc<>12)             /* Tell if RC not 0 or 12           */
```

Figure 281. TCPDEALT REXX

TAKE 1 accomplishes the second function of the TAKE 2 stage that is in ECHOS. For more information on CALLPIPE, refer to the CALLPIPE discussion in "Using CALLPIPE to Write Subroutine Pipelines" on page 94.

Here is an example of running these execs:

1. From the user ID that is to be the server, enter:

   ```
   echosnd 10000
   ```

   Note: The port number that is to be used was specified.

   This exec will immediately display the port number.

   ```
   Started on port 10000
   ```

2. From the user ID that is the client, enter:

   ```
   echoc First line of text ( 10000
   ```

   Note: The port number that was entered was the port number that ECHOS provided.

   The following is displayed on the console of the client:

   ```
   First
   line
   of
   text
   Ready;
   ```

3. From the user ID that is the client, enter:

   ```
   echoc Second line of text providing more fun (10000
   ```

   The following is displayed on the console of the client:

```
       Second
       line
       of
       text
       providing
       more
       fun
       Ready;
```

The following is displayed on the console of the server:

```
       Started on port 10000
       Server 0 is available.
       Server 1 is available.
       Server 2 is available.
        F i r s t
       C68999A2A3
        l i n e
       93899585
        o f
       9686
        t e x t
       A385A7A3
        E O D
       C5D6C440
       Server 0 is available.
        S e c o n d
       E28583969584
        l i n e
       93899585
        o f
       9686
        t e x t
       A385A7A3
        p r o v i d i n g
       979996A58984899587
        m o r e
       94969985
        f u n
       86A495
        E O D
       C5D6C440
       Server 1 is available.
```

To stop the server, enter the following from the console of the server:

```
       pipmod stop
```

Other TCP/IP Related Stages

There are six more TCP/IP related stages that support servers and clients:

HOSTID
HOSTNAME
> These stages generate a single output record that is either the host IP address (HOSTID) where the client or server is currently running or the actual host name (HOSTNAME). For more information about these stages, refer to the *z/VM: CMS Pipelines Reference*.

HOSTBYADDR
> This stage converts a host IP address into the host name known to the name server that serves the local TCP/IP virtual machine.
>
> For more information about this stage, refer to the *z/VM: CMS Pipelines Reference*.

HOSTBYNAME
> This stage converts any host name known to the name server serving the local

TCP/IP virtual machine into a host IP address. This stage accomplishes the reverse function of HOSTBYADDR, which converts an IP address to a host name. For more information about these stages, refer to the *z/VM: CMS Pipelines Reference*.

IPSOCKA
SOCKA2IP

These stages are helpful when using the UDP stage. IP2SOCKA converts a EBCDIC port number and host IP address into a hexadecimal record that UDP needs for datagrams. This hexadecimal record received by UDP is converted back to a EBCDIC form by SOCKA2IP For more information about these stages, refer to the *z/VM: CMS Pipelines Reference.*

Chapter 12. Filter Packages

Would you like to improve the performance of stages you have written? Would you like to share a set of your stages with other users? If so, consider building and using a *filter package*.

A filter package is a MODULE or TEXT file that contains user-written stages. These user-written stages do not necessarily have to be classified as filter stages; they can be any type of stage. A filter package also contains an entry point table defining the stages.

A filter package helps improve performance because it can be loaded into virtual storage as a nucleus extension. Once a filter package is loaded, it is considered to be an extension to the main pipeline module. Instead of searching for your stages on disk and loading them into storage, CMS Pipelines finds them immediately in virtual storage.

If you share your stages with others, you can also share the benefits of a filter package with them. Usually stages are shared by placing them on a shared minidisk or on a shared SFS directory. By placing a filter package on the shared space instead, other users can also load your stages into virtual storage.

There are two ways to load a filter package:
- By entering the name of the MODULE file after CMS Pipelines itself is loaded. (CMS Pipelines is loaded the first time a PIPE command is executed during a CMS session.)
- By letting CMS Pipelines do it for you

To have CMS Pipelines automatically load a filter package, enter on the command line one of four available special names. These special names are discussed in the next section.

Filter Package Names

CMS Pipelines has four special filter package names. The first time (and only the first time) a PIPE command is executed during a CMS session, it searches for any MODULE files having these names. For each name, the PIPE command loads the first filter package (if any) in the CMS search order that has a matching name. (Avoid inadvertently using a filter package you don't want.)

The names, along with their intended uses, are:

PIPPTFF
> This filter package can be used to replace built-in stages. If the name of a stage in this package has the same name as a built-in stage, the stage in the filter package is used.
>
> **Attention:** A stage placed in this package should not have a name that is identical with the name of a pipeline subcommand. If the stage does have an identical name, it will be used instead of the pipeline subcommand having the same name. The results are not predictable.

PIPSYSF
> PIPSYSF is intended to house stages that are to be shared across the entire system. Usually this filter package is placed on a system minidisk that is accessed by all users.

PIPLOCF

> PIPLOCF is intended to house stages that are to be shared by a local group of users (not all users of the system). Usually this filter package is placed on a minidisk or SFS directory that is locally shared (perhaps within a department).

PIPUSERF

> PIPUSERF is a user filter package. This filter package is used for private stages that are used often and should, therefore, remain in storage. Usually this filter package resides on a user's personal disk storage.

A filter package having another name is not automatically loaded.

Search Order

When CMS Pipelines searches for a stage, it first searches in virtual storage through any filter packages that have been loaded. Then it searches for the stage on disk, using the standard CMS search order. The following list summarizes the search order:

1. Search through virtual storage as follows:

   a. PIPPTFF

   b. Built-in stages

   c. PIPUSERF

   d. PIPLOCF

   e. PIPSYSF

   f. User filter packages without special names (in the order that they were loaded).

2. Search on disk using the CMS disk search order.

CMS Pipelines uses the first stage that matches. If, for instance, ADD REXX exists in PIPUSERF and in PIPSYSF, CMS Pipelines will use the one in PIPUSERF.

Building a Filter Package

This section describes how to build a filter package. Here is a summary of the steps we'll be following:

1. Create an input file listing all the REXX and Assembler user-written stages that will be in the filter package

2. Create a TEXT file from the input file

3. Create a load module

4. Load the filter package.

When building a filter package, you will use the PIPGFTXT EXEC and the PIPGFMOD EXEC which are on the S-disk.

The details of building a filter package follow. Read all the steps before trying to build a filter package:

1. Create a file listing the user-written stages.

   First, you must create a file that lists the file names and file types of all the stages you want to include in the filter package. This file may include both REXX and Assembler user-written stages. REXXES is the default file type for this input file. However, any file type can be specified.

For example, to build a filter package that contains The REXX stages PROGRAM1 REXX and PROGRAM2 CREXX, and the Assembler stage PROGRAM3 TEXT, create a file that contains the following:

```
PROGRAM1 REXX * = 3 REXX
PROGRAM2 CREXX * = 2 CREXX
PROGRAM3 TEXT * MFILTIA 4
```

The first two fields are the file name and file types of the source program. The other fields are described as part of the PIPGFTXT EXEC Input File Format section in the *z/VM: CMS Pipelines Reference*.

The name you select for the input file will become the name of the filter package. Suppose you want the filter package to be named MYFILTER. You would name the file MYFILTER REXXES. Now you are ready to build the TEXT file.

2. Create a TEXT file.

   To create a TEXT file from your input file, use the PIPGFTXT EXEC. PIPGFTXT by default generates an entry point table as part of the TEXT file.

   The TEXT file created has the same name as the input file. If the TEXT file already exists on your A-disk it will be written over.

   The format and operands of the PIPGFTXT EXEC are described in the *z/VM: CMS Pipelines Reference*.

   For example, to build the filter package MYFILTER TEXT from an input file named MYFILTER REXXES that contains various user-written stages, enter the following command:

   ```
   pipgftxt myfilter
   ```

   PIPGFTXT EXEC uses the default file name of REXXES and default file mode of * (asterisk), and by default creates an entry point name of PIPEPT for the local directory as part of the TEXT file.

   This TEXT file is a filter package that can be loaded into storage without creating a load module. By entering the CMS LOAD command with the name of the TEXT file, you can use the LDRTBLS stage to run any compiled stage built into the filter package. This allows you to test a new version of a user-written stage loaded in virtual storage while still retaining the original stage program in a filter package loaded as a MODULE file.

   If the user-written stages in the filter package (TEXT file) are the only ones that should be available, or are new, you are ready to create the load module.

3. Create a load module.

   To create a load module from the TEXT file and to remove a load module from storage, if it exists, use PIPGFMOD EXEC.

   The format and operands of the PIPGFMOD EXEC are described in the *z/VM: CMS Pipelines Reference*.

   For example, to create MYFILTER MODULE, enter the following:

   ```
   pipgfmod myfilter
   ```

   MYFILTER is the file name of the TEXT file generated by PIPGFTXT EXEC. In this example the name of a nucleus extension to be dropped is the same as the specified TEXT file name `myfilter`. The file MYFILTER MODULE is created or replaced if it already exists and was loaded as a nucleus extension. It is now ready to be loaded and used by CMS Pipelines.

   When CMS Pipelines loads a filter package having a special name (PIPPTFF, PIPSYSF, PIPLOCF, or PIPUSERF), it prefixes an asterisk to the name. That is, CMS Pipelines uses the names *PIPPTFF, *PIPSYSF, *PIPLOCF, and *PIPUSER. You can use any one of these filter package names as the name of the nucleus extension to be dropped (the second operand on PIPGFMOD).

For example, suppose you are creating a filter package named PIPUSERF. This new module will replace PIPMOD, the main pipeline module that has already been loaded as a nucleus extension. The command NUCXDROP PIPMOD drops the PIPMOD nucleus extension so that the PIPUSERF filter package of stages can be loaded and used instead. Issue the following PIPGFMOD and NUCXDROP commands:

```
pipgfmod pipuserf *pipuser
nucxdrop pipmod
```

Next, the filter package must be loaded.

4. Load the filter package.

The filter package is now a MODULE file and can be accessed by CMS Pipelines.

If the file name of the MODULE is PIPPTFF, PIPUSERF, PIPLOCF, or PIPSYSF, CMS Pipelines loads it automatically the first time a PIPE command is executed during a CMS session (assuming it is first in the CMS search order).

If the file name of the MODULE is not one of the special names, it is not loaded automatically. Before loading it, ensure that CMS Pipelines is loaded. CMS Pipelines is loaded the first time a PIPE command is executed during a CMS session. To load the module, invoke it by entering its name. For example, to load MYFILTER as a nucleus extension for CMS Pipelines to use, enter:

```
myfilter
```

MYFILTER MODULE loads itself as a nucleus extension. It remains loaded until the CMS session ends or until a CMS NUCXDROP command is entered for MYFILTER.

Replaced Filter Package Execs

This chapter described how to use the PIPGFTXT EXEC and the PIPGFMOD EXEC to build a filter package containing REXX or Assembler stages. These execs replace previous execs which built a filter package of only REXX stages. Specifically:
- PIPGFTXT EXEC replaces PIPGREXX EXEC
- PIPGFMOD EXEC replaces PIPLNKRX EXEC.

An existing input file containing only REXX stage names can be updated to include Assembler stage names and the PIPGFTXT and PIPGFMOD execs will create a new filter package with all the stages listed in that input file.

Chapter 13. Debugging Pipelines

If your pipelines are not working, there are several things you can do to unclog them. First, ensure the syntax of the pipeline is correct. The PIPE command scans your pipeline for syntax errors before processing. When PIPE finds syntax errors, it displays error messages. If you receive a nonzero return code, but no error messages, CP EMSG may be set off. See "Displaying Pipeline Messages" on page 237.

The messages often provide enough information for you to correct a syntax problem. To see an explanation of a message, use the HELP command. Also, use HELP (or the *z/VM: CMS Pipelines Reference*) to see a complete syntax description. All pipeline stages are in the PIPE help component. To see a description of LOCATE, for instance, enter:

```
help pipe locate
```

Once the pipeline syntax is corrected and your pipeline runs, you may face other problems:
- The pipeline does not produce output.
- The pipeline produces the wrong output.
- You receive an unexpected message on the terminal.

Several tools are available to help determine if the pipeline is coded incorrectly or if there is an error in CMS Pipelines. The tools are described in this chapter. Most problems can be solved by inspection or by tracing. However, before debugging your problem, browse through this chapter to see if one of the other techniques are appropriate.

Tracing Pipelines

This section uses an example to show how tracing and other tools are used to solve a problem. The example involves a pipeline that is supposed to read an employee file and list all employees starting with Jack Brown and ending with Betty Thomas. The employee file contains:

```
ALBERT, TOM     40   12904
BROWN, JACK     45   13784
BUTLER, JOE     42   13652
MARKS, SAM      40   17246
SMITH, SUE      37   16222
THOMAS, BETTY   46   15623
WHITE, JOHN     40   14523
```

We compose a pipeline that uses a BETWEEN stage to select the records:

```
pipe < employee file a | between Brown Thomas|console
FPLBTW338E Not binary data: rown
FPLSCA003I ... Issued from stage 2 of pipeline 1
FPLSCA001I ... Running "between Brown Thomas"
Ready(338);
```

The messages indicate that we have not correctly written stage 2, which contains the BETWEEN stage. After checking the BETWEEN stage syntax, we rewrite the expression with the syntax corrected as follows:

```
pipe < employee file a | between /Brown/ /Thomas/ |console
Ready;
```

Debugging Pipelines

This time the pipeline runs, but we do not get any output to the terminal. Because the reason is not obvious, we will trace the pipeline.

To trace a pipeline, specify the TRACE option on the PIPE command or the TRACE operand on the RUNPIPE stage. In both cases, the generated trace shows the stages as they are executed and the data that is passed from one stage to the next.

Figure 282 shows an excerpt of a trace specified with the TRACE option on the PIPE command. Notice how the trace option is specified.

```
pipe (trace) < employee file a | between /Brown/ /Thomas/ | console
FPLFPI402I Calling Syntax Exit
FPLSCA003I ... Issued from stage 1 of pipeline 1
FPLSCA001I ... Running "< employee file a"
FPLDSQ028I Starting stage with save area at X'03DA1388 03F6F5B8 00000000'
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "between /Brown/ /Thomas/"
FPLDSP538I Query state of INPUT stream 1
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "between /Brown/ /Thomas/"
FPLDSQ031I Resuming stage; return code is -4
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "between /Brown/ /Thomas/"
FPLDSP537I Commit level 0
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "between /Brown/ /Thomas/"
FPLDSQ028I Starting stage with save area at X'03DA1228 00EBA408 00000000'
FPLMSG003I ... Issued from stage 1 of pipeline 1
FPLMSG001I ... Running "< employee file a"
FPLDSP035I Output 80 bytes
FPLMSG003I ... Issued from stage 1 of pipeline 1
FPLMSG001I ... Running "< employee file a"
FPLDSP039I ... Data: "ALBERT, TOM 40 12904"
FPLDSQ031I Resuming stage; return code is 0
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "between /Brown/ /Thomas/"
FPLDSP034I "Locate" called
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "between /Brown/ /Thomas/"
:
```

Figure 282. Example of Trace Output

You would get the same result if you used the TRACE operand on RUNPIPE. See Figure 285 on page 232 for an example of using the TRACE operand on RUNPIPE.

Tracing a PIPE command that contains a user-written stage may produce slightly different output. For example, if you use CALLPIPE or ADDPIPE with the TRACE option in your user-written stage, the numbering of the stages in the trace output for CALLPIPE or ADDPIPE begins with one regardless of the stage numbering in the original PIPE command. However, if the CALLPIPE or ADDPIPE begins with a connector, the connector is counted as the first stage. When CALLPIPE or ADDPIPE has finished processing, the original connection of the stages in the PIPE command are restored and numbering of the stages continues as if the CALLPIPE or ADDPIPE was not issued.

In Figure 283 on page 231 FIXED is a user-written stage that contains the following:

```
/*  FIXED REXX  */
'callpipe *: | CHOP 80 | PAD 80 | *:'
```

The numbers in parentheses indicate the stage numbers that are displayed in the trace output if both the PIPE command and the CALLPIPE pipeline subcommand are traced.
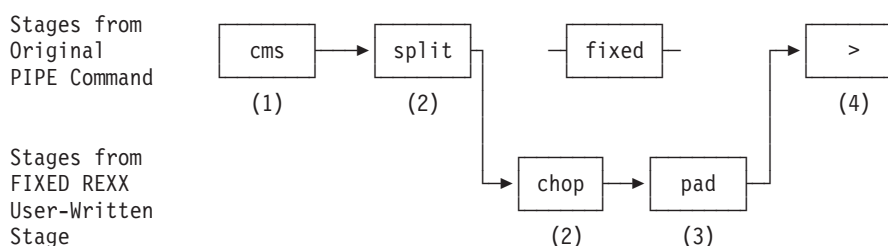
Figure 283. Numbering Stages of a Pipeline When CALLPIPE Is Running

Note that the number of the CHOP stage is 2 because the input connector counts as stage 1.

Usually the trace output is displayed on the console. However, because the trace output can be lengthy, it is often better to direct it to a file instead.

Tracing to a File

To direct trace output to a file, use the RUNPIPE stage. You can specify a trace by using the TRACE option on the PIPE command that is issuing the RUNPIPE stage (see Figure 284) or you can use the TRACE operand on RUNPIPE (see Figure 285 on page 232).

```
/* Directing Trace Output to a File with RUNPIPE                 */
mypipe = '(trace) < employee file a',
       '| between /Brown/ /Thomas/|console'
address command
'PIPE',
  'var mypipe',        /* Write variable MYPIPE to output stream */
  '| runpipe',         /* Run it                                 */
  '| > trace file a'   /* Put resultant trace in file            */
```

Figure 284. Directing Trace Output to a Data Set (RUNPIPE with PIPE TRACE Option)

This is another way to accomplish the same thing:

```
/*       -- Tracing to a file using TRACE argument of RUNPIPE       */
mypipe = '< employee file a',
         '| between /Brown/ /Thomas/|console'
address command
'PIPE',
  'var mypipe',        /* Write variable MYPIPE to output stream */
  '| runpipe trace',   /* Run it                                 */
  '| > trace file a'   /* Put resultant trace in file            */
```

Figure 285. Directing Trace Output to a Data Set (Using RUNPIPE TRACE)

The RUNPIPE stage reads its input stream and executes the records as pipelines. RUNPIPE writes any terminal output from the execution of the pipeline to its output stream. The stage following RUNPIPE writes the output to the file TRACE FILE A. In our examples, the output from the pipeline consists of trace records.

Figure 286 shows an excerpt of the trace file output. Line numbers are shown for illustration (they do not appear in the actual trace records):

```
  ⋮
                                                            Lines
FPLDSQ031I Resuming stage; return code is 0                   1
FPLMSG003I ... Issued from stage 1 of pipeline 1             2
FPLMSG001I ... Running "< employee file a"                   3
FPLDSP035I Output 80 bytes                                   4
FPLMSG003I ... Issued from stage 1 of pipeline 1             5
FPLMSG001I ... Running "< employee file a"                   6
FPLDSP039I ... Data: "BROWN, JACK     45   13784"            7
FPLDSQ031I Resuming stage; return code is 0                   8
FPLMSG003I ... Issued from stage 2 of pipeline 1             9
FPLMSG001I ... Running "between /Brown/ /Thomas/"           10
FPLDSP033I Input requested for 0 bytes                      11
FPLMSG003I ... Issued from stage 2 of pipeline 1            12
FPLMSG001I ... Running "between /Brown/ /Thomas/"           13
FPLDSQ031I Resuming stage; return code is 0                  14
FPLMSG003I ... Issued from stage 2 of pipeline 1            15
FPLMSG001I ... Running "between /Brown/ /Thomas/"           16
FPLDSP034I "Locate" called                                 17
FPLMSG003I ... Issued from stage 2 of pipeline 1            18
FPLMSG001I ... Running "between /Brown/ /Thomas/"           19
FPLDSQ031I Resuming stage; return code is 0                  20
FPLMSG003I ... Issued from stage 1 of pipeline 1            21
FPLMSG001I ... Running "< employee file a"                  22
FPLDSP035I Output 80 bytes                                  23
FPLMSG003I ... Issued from stage 1 of pipeline 1            24
FPLMSG001I ... Running "< employee file a"                  25
FPLDSP039I ... Data: "BUTLER, JOE     42   13652"           26
  ⋮
```

Figure 286. Console Output from Directing Trace Output to a Data Set

Figure 286 shows the processing for a record we believe should have been selected. To find out why there is not any output at the terminal, look at the output from stage to stage. The highlighting in the excerpt in Figure 287 on page 233 shows what pattern to focus on when scanning this sort of trace:

```
                                                             Lines
FPLDSQ031I Resuming stage; return code is 0                    1
FPLMSG003I ... Issued from stage 1 of pipeline 1              2
FPLMSG001I ... Running "< employee file a"                     3
FPLDSP035I Output 80 bytes                                    4
FPLMSG003I ... Issued from stage 1 of pipeline 1             5
FPLMSG001I ... Running "< employee file a"                     6
FPLDSP039I ... Data: "BROWN, JACK    45   13784"              7
FPLDSQ031I Resuming stage; return code is 0                    8
FPLMSG003I ... Issued from stage 2 of pipeline 1             9
FPLMSG001I ... Running "between /Brown/ /Thomas/"             10
FPLDSP033I Input requested for 0 bytes                        11
FPLMSG003I ... Issued from stage 2 of pipeline 1            12
FPLMSG001I ... Running "between /Brown/ /Thomas/"             13
FPLDSQ031I Resuming stage; return code is 0                   14
FPLMSG003I ... Issued from stage 2 of pipeline 1            15
FPLMSG001I ... Running "between /Brown/ /Thomas/"             16
FPLDSP034I "Locate" called                                   17
FPLMSG003I ... Issued from stage 2 of pipeline 1            18
FPLMSG001I ... Running "between /Brown/ /Thomas/"             19
FPLDSQ031I Resuming stage; return code is 0                   20
FPLMSG003I ... Issued from stage 1 of pipeline 1            21
FPLMSG001I ... Running "< employee file a"                    22
FPLDSP035I Output 80 bytes                                   23
FPLMSG003I ... Issued from stage 1 of pipeline 1            24
FPLMSG001I ... Running "< employee file a"                    25
FPLDSP039I ... Data: "BUTLER, JOE    42   13652"             26
```

Figure 287. Highlighted Console Output from Directing Trace Output to a Data Set

Stage 1, which reads EMPLOYEE FILE A, has OUTPUT messages, but stage 2 does not. Stage 2, which contains the BETWEEN stage, is the culprit.

We see on line 7 that the output from stage 1 is BROWN, JACK   45   13784. The output from stage 1 is the input to stage 2. So, why does the BETWEEN stage not select the record?

A hint is on line 10. In the statement between /Brown/ /Thomas/, mixed case is used, but the data is in uppercase. Remembering that CMS Pipelines is case sensitive, we found the answer to our problem. Rerunning the command with the names in uppercase produces the output that we want.

```
pipe < employee file a | between /BROWN/ /THOMAS/| console
BROWN, JACK    45   13784
BUTLER, JOE    42   13652
MARKS, SAM     40   17246
THOMAS, BETTY  46   15623
Ready;
```

Figure 288 on page 234 is part of the trace from when the correct pipeline expression was entered:

```
                                                                 Lines
        FPLDSQ031I Resuming stage; return code is 0                1
        FPLMSG003I ... Issued from stage 1 of pipeline 1           2
        FPLMSG001I ... Running "< employee file a"                 3
        FPLDSP035I Output 80 bytes                                 4
        FPLMSG003I ... Issued from stage 1 of pipeline 1           5
        FPLMSG001I ... Running "< employee file a"                 6
        FPLDSP039I ... Data: "BROWN, JACK 45 13784"                7
        FPLDSQ031I Resuming stage; return code is 0                8
        FPLMSG003I ... Issued from stage 2 of pipeline 1           9
        FPLMSG001I ... Running "between /BROWN/ /THOMAS/"          10
        FPLDSP035I Output 80 bytes                                11
        FPLMSG003I ... Issued from stage 2 of pipeline 1          12
        FPLMSG001I ... Running "between /BROWN/ /THOMAS/"         13
        FPLDSP039I ... Data: "BROWN, JACK 45 13784"               14
        FPLDSQ028I Starting stage with save area at X'03DA14E8 03F6F700 00000000'  15
        FPLMSG003I ... Issued from stage 3 of pipeline 1          16
        FPLMSG001I ... Running "console"                          17
        FPLDSP034I "Test if PIPRUNEVENTS active" called           18
        FPLMSG003I ... Issued from stage 3 of pipeline 1          19
        FPLMSG001I ... Running "console"                          20
        FPLDSQ031I Resuming stage; return code is 0               21
        FPLMSG003I ... Issued from stage 3 of pipeline 1          22
        FPLMSG001I ... Running "console"                          23
        FPLDSP034I "Locate" called                               24
        FPLMSG003I ... Issued from stage 3 of pipeline 1          25
        FPLMSG001I ... Running "console"                          26
        FPLDSQ031I Resuming stage; return code is 0               27
        FPLMSG003I ... Issued from stage 3 of pipeline 1          28
        FPLMSG001I ... Running "console"                          29
        BROWN, JACK    45   13784                                 30
        FPLDSP035I Output 80 bytes                                31
        FPLMSG003I ... Issued from stage 3 of pipeline 1          32
        FPLMSG001I ... Running "console"                          33
        FPLDSP039I ... Data: "BROWN, JACK 45 13784"               34
```

Figure 288. Console Output of Trace with Correct Pipeline Expression Set

Notice that all of the stages now have output for a record that satisfies the
BETWEEN criteria.

Tracing Individual Stages

Tracing generates very large trace files, therefore, you may want to trace just a few stages. Put the TRACE option at the beginning of the stage you wish to trace, as shown in Figure 289.

```
pipe literal test | (trace) literal trace selectivity | console
FPLDSQ028I Starting stage with save area at X'03DA1390 03F6F6B0 00000000'
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "literal trace selectivity"
FPLDSP035I Output 18 bytes
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "literal trace selectivity"
FPLDSP039I ... Data: "trace selectivity"
trace selectivity
FPLDSQ031I Resuming stage; return code is 0
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "literal trace selectivity"
FPLDSP034I "SHORT" called
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "literal trace selectivity"
FPLDSQ031I Resuming stage; return code is 0
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "literal trace selectivity"
FPLDSP020I Stage returned with return code 0
FPLMSG003I ... Issued from stage 2 of pipeline 1
FPLMSG001I ... Running "literal trace selectivity"
test
Ready;
```

Figure 289. Tracing Individual Stages (TRACE Option)

You cannot trace an individual stage using the TRACE operand on the RUNPIPE stage.

Controlling Trace Messages

You can suppress some trace messages by specifying the NOMSGLEVEL option. NOMSGLEVEL lets you selectively suppress certain levels of messages. Figure 290 shows an example in which some messages are suppressed for the stage being traced. See the *z/VM: CMS Pipelines Reference* for more information about NOMSGLEVEL.

```
pipe literal test | (trace nomsglevel 7) literal trace selectivity | console
FPLDSQ028I Starting stage with save area at X'03DA1388 03F6F6B0 00000000'
FPLDSQ001I ... Running "literal trace selectivity"
FPLDSP035I Output 20 bytes
FPLDSP039I ... Data: "trace selectivity"
trace selectivity
FPLDSQ031I Resuming stage; return code is 0
FPLDSP034I "SHORT" called
FPLDSQ031I Resuming stage; return code is 0
FPLDSP020I Stage returned with return code 0
test
Ready;
```

Figure 290. Using NOMSGLEVEL to Suppress Messages

You can also control trace messages for the entire pipeline by using either the TRACE NOMSGLEVEL option on the PIPE command or by using the MSGLEVEL operand and the TRACE operand on RUNPIPE. See the *z/VM: CMS Pipelines Reference* for more information about NOMSGLEVEL and MSGLEVEL.

Taking Snapshots of Data

As an alternative to tracing, consider taking snapshots of your data as it passes through stages. Tracing lets you follow the data a record at a time, but snapshots let you see all the data at once.

For example, suppose you want to see how the data changes after each stage in this PIPE command:

```
/* Display list of SCRIPT files sorted in descending order     */
'pipe',
   'cms listfile * * a',      /* Execute LISTFILE command        */
   '| locate 10.8 /SCRIPT/',  /* Find those with SCRIPT file type */
   '| sort descending',       /* Sort them                       */
   '| console'
```

Figure 291 shows how to take snapshots from an exec. Stages that write to files are added to the pipeline.

```
/* Display list of SCRIPT files sorted in descending order     */
'pipe',
   'cms listfile * * a',      /* Execute LISTFILE command        */
   '| > test1 snapshot a',    /* ...take snapshot                */
   '| locate 10.8 /SCRIPT/',  /* Find those with SCRIPT file type */
   '| > test2 snapshot a',    /* ...take snapshot                */
   '| sort descending',       /* Sort them                       */
   '| console'
```

Figure 291. Taking Snapshots

Naming Pipelines (NAME Option)

When writing execs that call other execs that issue PIPE commands, it can be difficult to find a failing PIPE command. The error messages reveal the problem, but you cannot figure out which pipeline issued the messages. In these situations, use the NAME option on your PIPE commands.

NAME lets you give a pipeline a name. The name is displayed in any error messages caused by the pipeline.

The following example shows how you might keep track of each time an exec is run by writing a record in a log file. The TIME EXEC writes a record to a log file:

```
/* TIME EXEC -- print out the time to a log file each time an */
/* exec is run                                                */
'PIPE (NAME TIME) CP Q TIME | >> TIME FILE E'
```

TIME EXEC is called by the MYEXEC EXEC:

```
 /* MYEXEC EXEC                                               */
 /* This is an example of how one could keep track of each time an */
 /* exec is run in a log file                                 */

   'TIME'    /* Write time to the log file */
 /*********************************************/
 /*                                           */
 /*      BODY OF THE EXEC.                     */
 /*                                           */
 /*********************************************/
```

The only problem is that you do not have access to the disk where the log file is located. This causes a CMS Pipelines error. Because the NAME option is used, the error messages will include the name of the failing pipeline.

Displaying Pipeline Messages

CMS Pipelines issues messages when it detects errors. These messages are not displayed if the CP EMSG setting is OFF. If you receive a nonzero return code from your pipeline, but no messages, enter the CP SET EMSG ON command:

```
pipe < test file j | console
Ready(119);
set emsg on
Ready;
pipe < test file j | console
FPLDSR119E Mode J not available or read only
FPLSCA003I ... Issued from stage 1 of pipeline 1
FPLSCA001I ... Running "< test file j"
Ready(119);
```

In the above example, the TEST FILE was on file mode J, which was released before the PIPE command was issued.

See the *z/VM: CP Commands and Utilities Reference* for more information about the CP SET EMSG command.

Displaying All Nonzero Return Codes (LISTERR Option)

The PIPE command returns the most severe return code from all its stages. (Any negative return code is more severe than any positive return code.) If several stages give a nonzero return code, you see only the most severe code. To see the others, use the LISTERR option.

For example, the following PIPE command produces two errors:

```
pipe (listerr) cms listfile xxx scrpit a|append cms listile * exec a |console
```

Because the file XXX SCRPIT A does not exist, the CMS LISTFILE stage gives a nonzero return code. (Naturally, if you have a file named XXX SCRPIT A, the return code is zero.) The second nonzero return code is caused by a typing error in the APPEND stage. Instead of `listfile`, the string `listile` was entered.

With the LISTERR option, messages are displayed for both stages with the return codes. Without the LISTERR option, you see only one of the return codes. Try the command both ways to see the differences.

Appendix A. Additional Examples

This appendix contains additional examples that show how stages can be combined effectively.

Listing Frequently-Used Execs

POPEXECS (Figure 292) lists the ten execs in storage that are called most often. The CMS EXECMAP command writes a list, with usage statistics, of execs in storage. POPEXECS processes the response from the CMS EXECMAP command.

```
/* POPEXECS EXEC -- Find 10 most popular execs loaded with EXECLOAD */
'PIPE',
   'cms EXECMAP',            /* Get list of everything       */
   '| drop 1',               /* Remove title                 */
   '| specs 1.29 1',         /* Retain name and count        */
   '| sort 22-* descending', /* Sort                         */
   '| take 10',              /* Take the top ten             */
   '| console'               /* Display them                 */
exit rc
```

Figure 292. Finding Frequently-Used Execs: POPEXECS EXEC

Here is an example:

```
popexecs
PROFILE    XEDIT       185
PI         REXX         99
XEDMAC     XEDIT        89
ACC29A     EXEC         22
XPNDDEST   REXX         22
RDRXVDST   XEDIT        21
EQSYN      EXEC         11
PFXTEST    EXEC         11
ALTER      XEDIT        10
UNRAVEL    XEDIT        10
Ready;
```

Listing Accessed File Modes

The example in Figure 293 lists accessed file modes. It displays the list as a string of letters.

```
/* TYACC EXEC -- Get list of accessed file mode letters      */
'PIPE',
   'cms QUERY ACCESSED',     /* List in-use file mode letters */
   '| drop 1',               /* Drop the title line          */
   '| specs 1.1 1',          /* Keep only the mode letter     */
   '| join *',               /* Put them together            */
   '| console'               /* Display them                 */
exit rc
```

Figure 293. Listing Accessed File Modes: TYACC EXEC

Here is an example:

```
query accessed
Mode  Stat  Files  Vdev  Label/Directory
A     R/W     45   191   BAR191
B     R/W     62   DIR   SERVER8:KIM.PUBS
C     R/O   1152   19C   ESA19C
G     R/O   4728   19F   NUGOOD
S     R/O    349   190   CMS11
Y/S   R/O    378   19E   19ESP4
Z/Z   R/O    823   19D   ES1HLP
Ready;
tyacc
ABCGSYZ
Ready;
```

Counting Reader Files

The example in Figure 294 counts the number of reader files that are from local users. Files from other systems are sent via RSCS. The origin user ID of such a spool file is the virtual machine named in the fifth token of the response to the CMS IDENTIFY command.

The NFIND stage contains an underscore to ensure that records from only the network virtual machine are discarded. For instance, the NFIND stage discards RSCS but keeps RSCS1.

```
/* NETRDRF EXEC -- count local reader files                     */
address command
'IDENTIFY (LIFO'
parse pull . . . . rscs .
'PIPE',
   'cp Q RDR * ALL',    /* Issue the QUERY READER command        */
   '| drop 1',          /* Discard the header record             */
   '| nfind' rscs'_'||, /* Discard records from the RSCS machine */
   '| count lines',     /* Count the remaining lines from local users */
   '| console'          /* Display the count                     */
exit rc
```

Figure 294. Counting Reader Files: NETRDRF EXEC

Here is an example:
```
pipe cp query files | console
FILES: 15 RDR,  NO PRT,  NO PUN
Ready;
netrdrf
14
Ready;
```

Displaying Block Comments

DISPBLKC EXEC (Figure 295 on page 241) displays the first block comment in an exec. Specify the exec to be processed as an argument to DISPBLKC.

DISPBLKC uses the FRLABEL and TOLABEL stages to select the lines required. It drops the first line matched to avoid selecting a null range.

```
/* DISPBLKC EXEC -- Get block comment from an exec              */
address command
arg fn ft fm .
if ft='' then exit 9999
'PIPE',
   '<' fn ft fm ,                    /* Read the requested file   */
   '| frlabel /******************'||, /* Copy from start of block  */
   '| drop 1',                       /* But drop the first comment */
   '| tolabel /*****************'||, /* Stop at end of block      */
   '| change -/*--',                 /* Discard comment delimiters */
   '| change -*/--',
   '| console'                       /* Display the comment text  */
exit rc
```

Figure 295. Displaying Block Comments: DISPBLKC EXEC

Here is an example:

```
dispblkc dispblkc exec h
Ready;
pipe < blksamp exec | console
/* Demonstrate a block comment */

/********************************************************************/
/* This is a block comment.                                       */
/*                                                                */
/* It has three lines.                                            */
/********************************************************************/

exit RC
Ready;
dispblkc blksamp exec
 This is the block comment.

 It has three lines.
Ready;
```

Adding Sequence Numbers to a File

Figure 296 on page 242 shows an example exec that creates a SCRIPT file having sequence numbers with leading zeros in columns 1 through 8. The first record number is 10. The sequence numbers are incremented by 10.

Note that the SPECS RECNO option generates a 10-character field, but we want only 8 of them. An explicit field length takes care of this. You could also generate a 10-byte sequence field and then discard the first two bytes in a subsequent stage.

```
/* CRTSCR EXEC -- Create Script file with sequence numbers        */

address command
'PIPE',
   'literal :egdoc.'||,              /* Some data lines              */
   '| literal    here is a line in the document'||,
   '| literal :body.'||,
   '| literal :gdoc.'||,
   '| specs',
     'recno 1.7 right',              /* Add sequence numbers         */
     '1-* 9',                        /* Load record into column 9 onward */
   '| xlate 1.8 40 0',               /* Make leading zeros           */
   '| > sample script a'
exit rc
```

Figure 296. Adding Sequence Numbers: CRTSCR EXEC

Because the RIGHT operand is used to position the record number (RECNO), the number is padded on the left with blanks. Notice that column 8 is also a blank. The XLATE stage changes the leading blanks and the blank in column 8 to zeros. This yields leading zeros and an increment by 10.

Here is an example:

```
crtscr
Ready;
pipe < sample script|console
00000010:gdoc.
00000020:body.
00000030   here is a line in the document
00000040:egdoc.
Ready;
```

Copying between XEDIT Files

The XEDIT COPY subcommand copies lines of any length within a file, but copying lines from one file to another one is, in general, via disk. The IFCOPY XEDIT macro in Figure 297 on page 243 copies from one file to another without using disk storage. The format is:

IFCOPY *number to-file*

The first argument is the number of lines to copy from the current line in the current file. The remaining argument is the name of the file to receive the copied records. The lines are inserted after the current line.

Because the XEDIT stage cannot insert lines into a file, we use it to add the lines to the end of the target file instead. Then we use an XEDIT subcommand to move the added text from the end of the file to the appropriate location (that is, after the current line). The XEDIT stage is also used to read the lines of the source file. It is assumed that there are only two files and that the argument is the number of lines to copy.

```
/* IFCOPY XEDIT -- Copy from one file to another without using disk    */

parse arg numlines tofile

'extract /fname/ftype/fmode/line/size'  /* Where are we?              */
fromfile=fname.1 ftype.1 fmode.1   /* Remember it...                  */
fromline=line.1

'xedit' tofile                     /* Edit the "to" file             */
'extract /fname/ftype/fmode/line'  /* Remember it...                 */
tofile=fname.1 ftype.1 fmode.1
toline=line.1

'+* extract /line'          /* Go to the end and learn where that is    */
insert=size.1               /* Set INSERT to number of first line inserted */
'bottom'

address command,
'PIPE',
   'xedit' fromfile,            /* Get the "from" file             */
   '| take' numlines,           /* Keep only the lines requested   */
   '| xedit' tofile             /* Append to the "to" file         */

':'insert 'move * :'toline      /* Move lines where they belong    */
```

Figure 297. Copying between XEDIT Files: IFCOPY XEDIT

Because of XEDIT size restrictions, your records are truncated if you copy to a file that cannot accommodate the length of the lines inserted.

Reversing the Order of Records

The REVLINES REXX stage in Figure 298 on page 244 reverses the order of records flowing through it. It accepts a number as an operand. The operand indicates the number of records that are to be reversed. After reversing the specified number of records, REVLINES REXX copies any remaining records in its input stream to its output stream.

To reverse the records, REVLINES stores them in REXX variables and writes them in reverse order (notice the DO instruction). If the requested number is greater than the number of records in the input stream, REVLINES reverses all the available records.

```
/* REVLINES REXX -- Reverse the order of the first N lines        */
parse arg number .                    /* Read number of lines to reverse */
if ¬datatype(number, 'Whole') then exit 999   /* Valid number?          */

'callpipe',
   '*:',                               /* Connect input stream          */
   '| take' number,                    /* Take number of records requested*/
   '| stem in.'                        /* Put them in a stemmed array    */

do  i=in.0 by -1 to 1                  /* Write them in reverse order    */
   'output' in.i
end

'short'                                /* Copy the rest                  */
exit rc
```

Figure 298. Reversing the Order of Records: REVLINES REXX

Examples:

pipe literal def|literal abc|console
abc
def
Ready;
pipe literal def|literal abc|revlines 3|console
def
abc
Ready;
pipe literal ghi|literal def|literal abc|revlines 2|console
def
abc
ghi
Ready;

An alternative method of doing what the REVLINES REXX stage does is to use the INSTORE stage with the REVERSE operand specified. Then the OUTSTORE stage can read all the records back into the pipeline in reverse order.

Isolating Words

ISOWORD REXX (Figure 299) isolate words of text by writing each word in its input records to a separate output record. It retains quotation marks within words (for example, Fred's), but not quotation marks at the beginning or end (for example, users').

```
/* ISOWORD REXX -- Isolate words.                           */
'callpipe',
   '*:',                      /* Connect input stream          */
   "| xlate 40-7f 40 ' ' 0-9 40",  /* No punctuation and numbers    */
   '| split',                 /* Isolate words                 */
   '| specs 1-* next / / next',  /* Pad a blank on end            */
   "| change 1 /'//",         /* Remove leading quotation mark */
   "| change /' //",          /* ...and trailing one           */
   '| strip',                 /* Just words, no blanks         */
   '| *:'                     /* Write to output stream        */
exit rc
```

Figure 299. Isolating Words: ISOWORD REXX

The XLATE stage removes all other punctuation and numbers, but leaves single quotation marks. The record is then split into words. The SPECS stage puts a blank at the end of each word so that a following CHANGE stage can remove trailing quotation marks. After leading and trailing quotation marks are removed, STRIP removes any leading or trailing blanks. Note that case is respected in the output records. Use XLATE LOWER if you want all output records in lowercase.

Listing Files on Accessed File Modes

LFD REXX (Figure 300) writes information about all files on one or more accessed minidisks or accessed SFS directories. Specify a string of file mode letters as the operand. Specify an asterisk to list files on all accessed file modes.

Like many built-in device drivers, the LFD REXX example also processes records in its input stream. It expects these records to contain a string of one or more file mode letters (just like the operand). LFD processes the records in the input stream after processing the operand. The operand is optional if file mode letters are supplied in the input stream.

To process each file mode, LFD REXX uses COMMAND LISTFILE. When an asterisk is specified, LFD uses another user-written stage named ACCESSED to get a list of accessed file mode letters. ACCESSED REXX is shown in Figure 301 on page 246.

```
/* LFD REXX -- List files on a minidisk or number of disks        */
signal on error
parse arg in
if in='' then 'readto in'     /* No operand?  Read from input stream   */
else rc=0
do while rc=0
   do while in¬=''
      in=translate(strip(in))  /* Make uppercase and remove blanks     */
      if left(in,1)='*'        /* If character is asterisk, put actual */
         then in=accessed() || substr(in,2) /* letters in string       */
      parse var in mode +1 in
      'callpipe command LISTFILE * *' mode '(NOH LABEL | *:'
   end
   'readto in'
end
error: exit rc*(rc¬=12)

accessed:                      /* Return all accessed disks */
   address command 'PIPE',
      'accessed',              /* Use user-written stage    */
      '| stem modes.'          /* Get value                 */
return modes.1
```

Figure 300. Listing Files on Accessed File Modes: LFD REXX

The user-written stage ACCESSED REXX (Figure 301 on page 246) finds the mode letters of all accessed disks and directories.

```
/* ACCESSED REXX -- List accessed file modes                     */
arg disktype
'callpipe',
   'command QUERY DISK' disktype,      /* Ask CMS                 */
   '| drop 1',                         /* Drop the heading        */
   '| specs word3 1',                  /* File mode is the third word */
   '| chop 1',                         /* Just the letter         */
   '| join *',                         /* Combine all             */
   '| *:'                              /* Write to output stream  */
exit rc
```

Figure 301. Listing Accessed File Modes: ACCESSED REXX

Examples:

```
query disk a
LABEL  CUU M  STAT  CYL TYPE BLKSIZE    FILES   BLKS USED-(%) BLKS LEFT  BLK TOTAL
SECURE 100 A   R/W   10 3380 4096          10        57-04        1443       1500
Ready;
pipe lfd a | count lines | console
10
Ready;
```

Ignoring Case on FIND

All built-in stages that compare strings (such as LOCATE and FIND) are case-sensitive. The FINDU REXX example (Figure 302) does the same function as the FIND stage, but it ignores case.

FINDU REXX first ensures that input records are at least as long as the search string. Records that aren't as long as the search string cannot possibly match, so they are discarded.

If a record is long enough, it is prefixed with an uppercase version of itself. This uppercase version is as long as the search string. Then FIND is used to compare an uppercase version of the search string with the uppercase prefixes. Records that do not match are discarded. Matching records flow to a SPECS stage that removes the uppercase prefix, thus restoring the original record.

```
/* FINDU REXX -- Case-ignoring find                   */
parse upper arg a
arglen=length(a)
'callpipe',
   '*:',                        /* Connect input stream        */
   '| locate' arglen,           /* Ensure long enough          */
   '| specs 1.'arglen '1',      /* Put key at beginning of record */
      '1-* next',               /* Put original record after key  */
   '| xlate 1.'arglen 'upper',  /* Translate only the key      */
   '| find' a||,                /* Look for it                 */
   '| specs' arglen+1'-* 1',    /* Remove key                  */
   '| *:'
exit rc
```

Figure 302. Ignoring Case on FIND: FINDU REXX

Here is an example:

```
pipe literal a  | literal A | literal b | findu a | console
A
a
Ready;
```

Writing the First Lines of Files

If your files have meaningful comments on their first lines, you may wish to have an index of first lines. LINE1 REXX (Figure 303) is an example that produces such an index.

LINE1 REXX does not have an operand. Instead, it expects a list of files in its input stream. The list can be in CMS EXEC format (that is, the format created by the CMS LISTFILE command when the EXEC option is specified).

For each file listed in the input stream, LIST1 REXX writes an output record containing the name of the file and the first line from it. LIST1 eliminates comment delimiters from the lines.

```
/* LINE1 REXX -- Process a list of files for first lines        */
signal on error

/* We use ADDPIPE it preprocess the input records               */
'addpipe (name LINE1)',
   '*.input:',                  /* Connect input stream          */
   '| nfind *',                 /* Discard comments              */
   '| specs w3 - * 1',          /* Remove EXEC prefix            */
   '| strip',                   /* Strip leading and trailing blanks */
   '| find ',                   /* Keep non-null lines           */
   '| *.input:'

do forever
   'readto in'                  /* Read record containing file ID  */
   parse var in fn ft fm .                     /* Parse it          */
   if find('MODULE',ft)>0 then firstline=''  /* Use null for MODULE */
   else                         /* Otherwise get the first line... */
      'callpipe',
         '|< ' fn ft fm,        /* Read the file                 */
         '| take 1',            /* Take only the first line      */
         '| change ?/*??',      /* Get rid of comment delimiters */
         '| change ?*/??',
         '| change 1 ?*??',
         '| change 1.2 ?.*??',
         '| var firstline'     /* Put line in variable FIRSTLINE  */
   /* Now write the output record                                 */
   'output' left(fn,8) left(ft,8) left(fm,2)':' firstline
end

error: exit rc*(rc¬=12)
```

Figure 303. Writing the First Line of Files: LINE1 REXX

Here is an example:

```
listfile * exec b (exec
Ready;
pipe < cms exec | take 5 | line1 | console
ACCSRCE  EXEC    B1: Access a source disk
ACC19B   EXEC    B1: Links and accesses
```

```
ACC29A   EXEC    B1:  Ensures that the log file disk is accessed
AC4250   EXEC    B1:  Access for 4250 and dcf3
ADDENDA  EXEC    B1:  Process an addenda file
Ready;
```

Creating a Word List from XEDIT

The example in Figure 304 shows how to create a list of the words in a file with usage counts.

```
/* WORDLIST XEDIT -- Count of word usage in the file being edited     */

'extract /fname/ftype/fmode'        /* Find out the file ID           */
'locate :0'                         /* Go to the top of the file      */
'xedit = wordlist s'                /* Edit a file that does not exist */
'preserve'                          /* Save the editing environment   */
'set msgmode off'                   /* Don't tell us about null file   */
'locate -*'                         /* Go to the top                  */
'delete *'                          /* Delete any lines               */
'restore'                           /* Restore message setting        */
'set recfm v'                       /* Don't want trouble with length  */
address command,
   'PIPE',
      'xedit' fname.1 ftype.1 fmode.1,          /* Get file           */
      "| xlate lower 00-7f 40",     /* lowercase and discard          */
         " ' ' ",                   /* retain single quotation mark   */
      '| split',                    /* single words                   */
      "| xlate 1 ' 40",             /* not leading quotation marks    */
      '| strip',                    /* and no lone ones               */
      '| sort count',               /* sort and count occurrences     */
      '| specs 7-10 1 11-* 6',      /* format records                 */
      '| xedit'                     /* write to last edited file      */
r=rc
'locate :0'                         /* Go to the top of the file      */
'set alt 0 0'                       /* Reset alteration count         */
exit r
```

Figure 304. WORDLIST XEDIT

WORDLIST processes the contents of the current file and stores the result in another file in the ring. When WORDLIST completes its processing, you see the word list.

Executing a Filter against XEDIT Lines

The XSTG XEDIT macro (Figure 305 on page 250) lets you execute filters against the lines of a file you're editing. It reads a specified number of lines from the file, executes stages on those lines, and replaces the original lines with the changed lines.

XSTG takes two operands: a number followed by the stages you want to run. The number indicates how many lines should be processed by the stages you specify. The XSTG macro starts processing with the current line. In the following example, cranberry is the current line. The XSTG command indicates that 3 lines will be processed by the LOCATE and XLATE stages.

```
  FRUITS   SCRIPT   A1  V 255  Trunc=255 Size=5 Line=3 Col=1 Alt=0



===== * * * Top of File * * *
===== apple
===== banana
===== cranberry
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
===== date
===== elderberry
===== * * * End of File * * *



====> xstg 3 locate /berry/ | xlate upper
                                                      X E D I T  1 File
```

The resultant display follows. The line date is removed by the LOCATE stage.
XLATE translates the selected lines (cranberry and elderberry) to uppercase.

```
  FRUITS   SCRIPT   A1  V 255  Trunc=255 Size=4 Line=3 Col=1 Alt=3



===== * * * Top of File * * *
===== apple
===== banana
===== CRANBERRY
      |...+....1....+....2....+....3....+....4....+....5....+....6....+....7...
===== ELDERBERRY
===== * * * End of File * * *



====>
                                                      X E D I T  1 File
```

XSTG XEDIT is shown in Figure 305 on page 250.

```
/* XSTG XEDIT  -- run one or more stages on lines of the file        */

parse arg number stage
if verify(number, '0123456789')>0 then do    /* Valid argument?        */
   'emsg' number 'not a number'
   exit 28
end
'extract /line/size'                /* Get position and size          */
if line.1=0 then '+1 extract /line' /* Move down 1 if at top of file  */
todo=min(number, size.1-line.1+1)   /* Make sure we don't go past end */
'preserve'                          /* Save user's XEDIT settings     */
'set escape off'                    /* Set up XEDIT the way we need it */
'set linend off'
'set image off'
'set case mixed'
'set tabs 1'
'set trunc *'
address command 'PIPE',
   'xedit',                        /* Read from file               */
   '| take' todo,                  /* Take the requested number     */
   '|' stage,                      /* Add the user's stages         */
   '| spec /input/ 1',             /* Put INPUT command in column 1 */
     '1-* nextword',               /* Tack on the filtered record   */
   '| literal up',                 /* Back up one line              */
   '| buffer',                     /* Wait for all done             */
   '| subcom xedit'                /* Put the lines back            */
r=rc
'restore'                          /* Restore user's XEDIT settings */
if r=0 then ':'line.1 'delete' todo /* Get rid of original lines     */
exit r
```

Figure 305. XSTG XEDIT

XSTG XEDIT works only with files that have records up to 254 bytes in length; XEDIT truncates longer input lines without issuing an error message.

The settings that could change the appearance of a line are disabled because processed lines are put back in the file with the INPUT XEDIT subcommand.

The first few stages get the desired number of records from XEDIT and perform the operation. INPUT commands are generated to insert the output lines in the file. A command to back up one line is also generated.

All these commands are buffered in BUFFER. This is important because the reading should finish before lines are added. If not, there might be a loop where the same line is processed over and over again.

Finally the macro deletes the lines that were read from the file, leaving just the newly inserted ones.

There are several approaches to overcome the problems arising from inserting the changed lines with SUBCOM XEDIT instead of using the XEDIT device driver.

You can run two PIPE commands: One to send the lines to a stemmed array or to a utility file you have added to the XEDIT ring (or to a disk file); another to copy these records into the file where they should go, after you have deleted or added sufficient lines to accommodate the lines added or deleted by the filters.

Another approach is to use the XEDIT stage to add the lines at the end of the file. After the pipeline finishes, move them to where they belong with an XEDIT subcommand.

Counting Files

Figure 306 shows an example exec named NUMTYPE that uses SORT COUNT. NUMTYPE lists the unique file types on file mode A. Preceding each file type is a count of the number of files having that file type. The list of counts and file types is displayed in descending order by count. If several file types have the same count, they are displayed in alphabetic order within that count.

```
/* NUMTYPE EXEC -- List the number of files having each file type    */
'pipe',
   'cms listfile * * a',                /* Process LISTFILE command  */
   '| specs word2 1',                   /* Keep only the file type   */
   '| sort count',                      /* Sort and count them       */
   '| sort 1-10 desc 11-18 ascending',  /* Reorder the records       */
   '| specs 1-10 1 11-* 15',            /* Space them out            */
   '| literal     Number    File Type', /* Add a heading line        */
   '| console'                          /* Display it                */
exit rc
```

Figure 306. Example Exec for SORT COUNT: NUMTYPE EXEC

A CMS LISTFILE command is processed to put a list of files in the pipeline. SPECS writes records containing only the file type. Then SORT COUNT orders the records, discards duplicates, and counts the number of duplicates of each record. The count is in columns 1 through 10, with the original record immediately following. If there were 140 SCRIPT files, for example, the record would have this format:

```
       140SCRIPT
```

The second SORT stage uses two sort ranges. The first range is the count (columns 1 through 10). The second range is the file type (columns 11 to the end of the record). The first range is sorted in descending order (so the file type having the greatest number of files is first). The second range is sorted in ascending order (so that file types having the same count are in alphabetic order).

After the sort, SPECS is used to space the records. LITERAL puts a header ahead of all incoming pipeline data. Finally, CONSOLE displays the results. Here is an example run:

```
numtype
     Number    File Type
        140    SCRIPT
        105    EXEC
         91    XEDIT
         34    C
         28    CIUOLIB
         25    OFSLOGfl
         16    MODULE
         15    PACKAGE
         14    $PMLIBS$
         12    REXX
         10    OFSMLIST
         10    SXEDIT
          9    LIST3820
          9    NOTE
```

```
9    NOTEBOOK
.    .
.    .
.    .
```

If you have many files, it might be best to direct the output of NUMTYPE to a file (using another pipeline, of course):

```
pipe cms numtype | > numtype output a
Ready;
```

Trapping the Responses to RSCS Commands

The following stage issues an RSCS command and writes the response from RSCS to the primary output stream. Messages from other sources are written to the secondary output stream if it is defined.

```
/* QRSCS REXX -- Query RSCS, waiting for a response */
address command 'CP SET MSG IUCV'     /* Tell CP to route messages via IUCV */
address command 'IDENTIFY (LIFO'      /* Identify tells us RSCS user ID     */
parse pull . . . . net .              /* Extract the RSCS user ID           */

'maxstream output'                    /* How many streams connected?        */
if rc=0 then sec=''                   /* No secondary, discard other stuff  */
else sec='?rscs: | *.output.1'        /* Arrange to write to secondary      */

'callpipe (endchar ? name QRSCS)',
   'literal +5',                      /* Five seconds max                   */
   '| delay',                         /* Wait                               */
   '| specs /PIPMOD STOP/ 1',         /* Build command to stop PIPE         */
   '| command',                       /* Hold here                          */
   '?',
   'starmsg CP SMSG' net arg(1),      /* Get messages from everywhere       */
   '| rscs: find 00000001'left(net,8,'_'),   /* Just messages from net    */
   '| specs 17-* 1',                  /* Remove prefix                      */
   '| *: ',                           /* Write to primary output            */
   sec                                /* Write to secondary if it exists    */
exit rc
```

Figure 307. Example Stage to Trap RSCS Responses: QRSCS REXX

Processing Reader Files

The READER stage takes the reader file CP gives to it. The RDR REXX stage in Figure 308 on page 253 looks for a specific file and ensures it can be read (correct type, not in hold, and reader spooled to the same class as the file). The argument to RDR is the spool file ID. The file will remain in the reader after it is read.

```
/* RDR REXX -- Ready a reader file for processing */

parse arg sfid .
if sfid=''
   then call err 28, 'No spool file ID specified.'

address command 'CP CLOSE RDR'    /* Ensure reader is closed */

address command 'PIPE',
   'cp QUERY RDR' sfid,
   '| drop 1',
   '| var file'
if RC¬=0
   then call err RC, 'Unable to locate spool file' sfid

parse var file . . class type . . hold .

if find('PUN PRT RDR',type)=0
   then call err 90, 'Unsupported spool file type' type

if hold¬='NONE'                   /* Not in hold status? */
   then
      if hold='USER'             /* In user hold?       */
            then call diag 8, 'CHANGE RDR' sfid 'NOHOLD'
            else call err 100, 'Unsupported hold status' hold

address command 'CP SPOOL RDR CLASS' class 'NOCONT'
call diag 8, 'ORDER RDR' sfid

'callpipe',
   'reader hold',              /* Read the reader but keep the file */
   '| nfind' '03'x||,               /* Discard no-ops       */
   '| *:'
r=RC
address command 'CP CLOSE RDR'    /* Close the reader     */

exit r

err: procedure
   parse arg retc, msgtext
   parse source . . fn ft .
   'message' space(fn ft)':' msgtext
exit retc
```

Figure 308. Example Using Reader Files: RDR REXX

To test RDR we generated a spool file by entering:

```
spool pun *
Ready;
pipe < vmletter script | punch
Ready;
close punch
PUN FILE 0002 SENT FROM YOURID   PUN WAS 0002 RECS 0204 CPY  001 A NOHOLD NOKEEP
Ready;
```

The first character of the line is the X'41' channel command code character. This file is read in the first sample in Figure 309 on page 254 (the file has one line); the second sample shows the response when there is no reader file with the spool ID specified.

```
pipe rdr 0002 | console
 DSMBEG323I STARTING SECOND PASS.
Ready;
pipe rdr 1 | console
RDR REXX: Unable to locate SPOOL file 1
Ready;
```

Figure 309. Running RDR REXX

See Chapter 8, "Using Unit Record Devices," on page 173 for more information about using CMS Pipelines with spool files.

Marking Selected Lines

Figure 310 shows MARKLINE REXX. MARKLINE scans its input records for a specified string. Records containing the string are prefixed with a pointer. Records not containing the target get a blank prefix. MARKLINE writes the prefixed records to its output stream.

```
/* MARKLINE REXX -- Mark lines with a locate target.              */
parse arg target
if target='' then target='/ /'

'callpipe (endchar ?)',
   '*:',                    /* Connect to input stream            */
   '| l: locate' target,    /* Look for target                    */
   '| specs /---> / 1',     /* Put mark at beginning of output record */
       '1-* next',          /* Put all of input record after the mark */
   '| f: faninany',         /* Join with other lines              */
   '| *:',                  /* Send to output                     */
   '?',
   'l:',                    /* Lines without the target flow here */
   '| specs /      / 1',    /* Put blanks at the beginning of output */
       '1-* next',          /* Put all of input record after the blanks */
   '| f:'                   /* Send to FANINANY to be merged with others */
exit rc
```

Figure 310. Marking Selected Lines: MARKLINE REXX

Example run:

```
pipe < fruits script | console
apple
banana
cranberry
date
elderberry
Ready;
pipe < fruits script | markline /berry/ | console
     apple
     banana
---> cranberry
     date
---> elderberry
Ready;
```

Creating Two-Column Output

The stage in Figure 311 displays lines 1 through *n* and lines *n*+1 through 2*n* in a two-column format on your terminal:

```
Line 1                     Line n+1
Line 2                     Line n+2
....                       ....
Line n                     Line 2n
```

The first operand is the number of lines per page; the second one is the indentation of the second column. The remaining operands are written as a header before each set of records.

```
/* TWO-UP REXX -- display records in two columns                    */
parse arg pl indent title
'peekto'
do while rc=0
   if title ¬='' then 'output' title  /* Write title if there is one   */
   'callpipe (listerr endchar ?)',
      '*:',                       /* Connect to input              */
      '| t: take' pl,            /* Take the first column         */
      '| buffer',                /* Buffer it                     */
      '| s: specs 1-* 1',        /* Put primary input in first column */
        'select 1',              /* Switch to secondary input     */
        '1-*' indent,            /* Put it in second column       */
      '| *:',                    /* Send it to output             */
      '?',
      't:',
      '| take' pl,               /* Take the second column        */
      '| s:'                     /* Send it to SPECS for merge    */
   if rc¬=0 then exit rc
   'peekto'                      /* Done?                         */
end

   exit rc*(rc¬=12)
```

Figure 311. Creating Two-Column Output: TWO-UP REXX

Example run:

```
pipe < fruits script | console
apple
banana
cranberry
date
elderberry
Ready;
pipe < fruits script | two-up 3 35 Fruits: | console
Fruits:
apple                          date
banana                         elderberry
cranberry
Ready;
```

Note: Another stage that can be used to accomplish this example is the SNAKE stage.

Putting First Last and Last First

The example in Figure 312 writes the first record of the input stream after it has written all the other records to the output stream:

```
/* FIRLAST REXX -- Write the first input record last            */
'callpipe (end ?)',
   '*:',
   '| d: drop 1',  /* Write first record to secondary output stream */
   '| f: fanin',   /* Read primary input, then secondary          */
   '| *:',
   '?',
   'd:',
   '| buffer',     /* Hold the first record until all are read     */
   '| f:'
exit rc
```

Figure 312. Writing the First Record Last: FIRLAST REXX

The example in Figure 313 puts the last record of the input stream first in the output stream file.

```
/* LASTFIR REXX -- Write the last input record first            */
'callpipe (end ?)',
   '*:',
   '| d: take last',  /* Write all but last record to secondary output */
   '| f: fanin',      /* Combine streams                              */
   '| *:',
   '?',
   'd:',
   '| buffer',        /* Hold records until the last is read          */
   '| f:'
exit rc
```

Figure 313. Writing the Last Record First: LASTFIR REXX

In the first example you need to buffer the first record of the file; in the second example you must buffer all except the last record of the file. Example runs of FIRLAST and LASTFIR follow:

```
pipe < fruits script | firlast | console
banana
cranberry
date
elderberry
apple
Ready;
pipe < fruits script | lastfir | console
elderberry
apple
banana
cranberry
date
Ready;
```

Tagging and Spooling

The example in Figure 314, TAGNSPL REXX, directs the punch to a destination (node ID and user ID) coded as the argument. Then it sends the pipeline records to the device.

```
/* TAGNSPL REXX -- Tag and spool the punch, then punch the file      */
arg node user .

address command                      /* Issue several CP and CMS commands */
'IDENTIFY (LIFO'                     /* Where am I?                       */
parse pull . . mynode . rscs .
if node=mynode                            /* Local or remote            */
   then spoolto=user
   else spoolto=rscs
'CP SPOOL D PURGE' spoolto 'NOHOLD CLASS A'  /* Spool the punch         */
'CP TAG DEV D' node user                     /* Tag the punch           */

address                              /* Revert to pipeline environment  */
'callpipe *: | punch'                /* Write the records               */
exit rc
```

Figure 314. Tagging and Spooling the Punch: TAGNSPL REXX

Create a Print File from a Reader File

Assuming the first reader file is a PRT file, the following PIPE command creates a copy of the file on the virtual printer without an intermediary disk file:

```
pipe reader | drop 1 | printmc
```

Figure 315 shows a better solution. It builds and executes a CP TAG command. The tag information is usually on the first record of a print file.

```
/* REPRINT EXEC -- Create a print file from a reader file            */
address command
'PIPE (listerr endchar ?)',
   'reader',                     /* Read from the card reader        */
   '| a: drop 1',                /* Drop tag -- send it on secondary stream */
   '| printmc e',                /* Print the rest of the file       */
   '?',
   'a:',                         /* Connect to secondary output of DROP   */
   '| specs /TAG DEV 00E / 1',   /* Build a tag command...           */
     '2-* next',                 /* ...and add tag from DROP to it   */
   '| cp'                        /* Have CP run the command          */
r=rc
'CP CLOSE 00C'
'CP CLOSE 00E'
exit r
```

Figure 315. Creating a Print File from a Reader File: REPRINT EXEC

Punching Files

The PUNFILES EXEC (Figure 316 on page 258) punches files in the format used by the CMS PUNCH command. The operand is a file name pattern. PUNFILES uses the pattern on a LISTFILE command to determine which files to punch. All the files are punched as a single spool file.

Additional Examples

On the LISTFILE command, the options LABEL and NOH are used to get the information necessary to build the :READ record.

The PUNFILES EXEC uses the CMS LISTFILE command to find the files to punch. It uses PUNFILES REXX to get a file with the proper header.

```
/* PUNFILES EXEC -- Pipeline punch */
parse arg fn ft fm  .

address command

'PIPE',
   'cms LISTFILE' fn ft fm '(LABEL NOH',
   '| stem fileinfo.'
if rc¬=0 then do
   r=rc
   'PIPE',
      'stem fileinfo.',
      '| emsg'
   exit r
end

'PIPE',
   'stem fileinfo.',
   '| punfiles',
   '| punch'
r=rc
if r=0 then 'CP CLOSE D NAME MANY FILES'
else 'CP SPOOL D PURGE'
exit r
```

Figure 316. Punching Files: PUNFILES EXEC

PUNFILES REXX (Figure 317) gets the file and puts the proper header on it.

```
/* PUNFILES REXX -- Build header and read file into pipeline */
signal on error
do forever
   'readto fileinfo'
   parse var fileinfo file+20 57 date time label .
   hdr=':READ ' file left(label,6) right(date,8,0) right(time,8,0)
   'callpipe',
      'var hdr',                /* Get header      */
      '| append <' file,        /* then the file   */
      '| *:'                    /* To output       */
end
error: exit rc*(rc¬=12)
```

Figure 317. Building Punch File Headers: PUNFILES REXX

Appendix B. CMS Pipelines Summary

Table 3. What Each Stage Does

Stage	Task Performed
Assembler Files	
ASMCONT	Joins multiline assembler statements.
ASMFIND	Selects assembler statements that begin with a specified text.
ASMNFIND	Selects assembler statements that do not begin with a specified text.
ASMXPND	Splits assembler statements.
STRASMFIND	Selects assembler statements that begin with a specified string of characters.
STRASMNFIND	Selects assembler statements that do not begin with a specified string of characters.
Blocking and Deblocking	
BLOCK	Blocks records.
DEBLOCK	Converts blocked records back into their original format or creates logical records from an external data format.
FBLOCK	Reformats the primary input stream records to blocks of a specified size.
IEBCOPY	Processes an MVS unloaded data set.
PACK	Compacts data.
UNPACK	Converts primary input stream records compressed by PACK back to their original format.
Changing Records	
3270BFRA	Converts a 2-byte unsigned integer to the 12-bit buffer address required for some 3270 devices, or vice versa.
APLDECODE	Translates characters in the same way that the CMS command SET TEXT ON or SET APL ON translates characters read from a 3270 display capable of displaying APL/TEXT characters.
APLENCODE	Translates characters in the same way that the CMS command SET TEXT ON or SET APL ON translates characters written to a 3270 display capable of displaying APL/TEXT characters.
ASATOMC	Converts ASA carriage control to machine carriage control.
BUILDSCR	Converts print files with machine carraige control characters, such as those produced by OVERSTR or XPNDHI, to records containing 3270 character attributes.
CHANGE	Replaces a string of characters with another string of characters.
CHOP	Selectively truncates records.
COMBINE	Combines several records into one record according to a specified logical operator.
C14TO38	Replaces a set of overstruck characters with a single character.
DATECONVERT	Performs date format conversion and date validation.
JOIN	Concatenates groups of records.
JOINCONT	Joins records marked with a continuation string.
MCTOASA	Converts machine carriage control to ASA carriage control.
OPTCDJ	Generates a Table Reference Character (TRC) byte.
OVERSTR	Processes overstruck lines.
PAD	Extends records with one or more occurrences of a specified character.
REVERSE	Reverses the contents of records on a character-by-character basis.
SNAKE	Builds a multicolumn page layout.

CMS Pipelines Summary

Table 3. What Each Stage Does (continued)

Stage	Task Performed
SPECS	Rearranges the contents of records.
SPILL	Splits lines longer than a specified number into multiple output lines.
SPLIT	Splits records into multiple records.
STRIP	Removes leading or trailing characters from records.
TOKENIZE	Parses records according to a specified token.
UNTAB	Expands tab characters (X'05') to blanks for lining up data into columns.
VCHAR	Recodes characters to a different length.
XLATE	Translates characters based on a specified translation table.
XPNDHI	Highlights spaces between words.
Device Drivers	
3270ENC	Prepares a 64-character translate table used to convert binary values in the range B'000000' through B'111111' (64 values) to displayable 1-byte graphic characters for placement in a 3270 data stream.
APPEND	Writes primary input stream records to the primary output stream followed by records from a specified stage or subroutine pipeline.
BFS	Reads from an existing byte stream file when BFS is first in a pipeline, otherwise writes its input records to (appends to or creates) a BFS file.
BFSDIRECTORY	Reads from an existing BFS directory file and writes one record to the primary output stream for each directory entry.
BFSQUERY	Obtains information from OpenExtensions™ about the current working BFS directory, or the contents of the symbolic links, or the current operating system.
BFSREPLACE	Reads records from its primary input stream and writes those records to its connected primary output stream, replacing the contents of the specified byte stream file.
BFSSTATE	Writes records containing status information about byte stream files to its primary output stream.
BFSXECUTE	Reads a record containing a request from its primary input stream, and sends that request to OpenExtensions services. The record containing the request is written to the primary output stream, if it is connected.
CMS	Issues CMS commands with full command resolution.
COMMAND	Issues CMS commands as if they were invoked using ADDRESS COMMAND from REXX/VM.
CONSOLE	Reads from or writes to the terminal in line mode.
CP	Issues CP commands.
EMSG	Displays each record as an error message.
FULLSCREEN	Writes 3270 data streams to the virtual console in fullscreen mode or to a 3270 device.
FULLSCRQ	Queries 3270 device characteristics.
FULLSCRS	Formats output from FULLSCRQ.
ISPF	Accesses ISPF tables.
LISTPDS	Reads the directory of a CMS simulated partitioned data set, such as a macro library, discards the header record, and writes one record for each member of the library.
LITERAL	Writes the specified data to the primary output stream and then copies primary input stream records to the primary output stream.
MEMBERS	Extracts members from a MACLIB, TXTLIB, or a file with a similar format.
PDSDIRECT	Writes directory information from a CMS simulated partitioned data set.

Table 3. What Each Stage Does (continued)

Stage	Task Performed
PREFACE	Writes records from a specified stage or subroutine pipeline to the primary output stream followed by primary input stream records.
PRINTMC	Writes records to a virtual printer.
PUNCH	Writes records to a virtual punch.
READER	Reads data from a virtual card reader.
SQL	Issues SQL statements.
SQLCODES	Writes return codes received from DB2 Server for VM.
STORAGE	Reads from or writes to virtual machine storage.
STRLITERAL	Writes the specified data to the primary output stream and then copies primary input stream records to the primary output stream.
SUBCOM	Passes specified commands to a specified subcommand environment.
TAPE	Reads from or writes to a tape at its current position.
URO	Writes records to a virtual printer or virtual punch.
VMC	Sends messages over the Virtual Machine Communications Facility (VMCF) to a service machine.
XAB	Reads or writes an external attribute buffer from a virtual printer or file.
XRANGE	Creates one record containing a specified range of characters.
Event-Driven	
DELAY	Waits until a particular time of day or until a specified interval of time has passed to copy the record.
IMMCMD	Writes argument strings entered on specified immediate commands.
PIPESTOP	Terminates stages waiting for an external event.
STARMONITOR	Writes monitor records from the CP *MONITOR system service.
STARMSG	Writes lines from a CP message service.
STARSYS	Writes lines from and sends replies to a CP system service.
Execs	
REXX	Invokes a REXX program as a stage.
REXXVARS	Gives information about REXX variables.
SCM	Lines up comments and completes unclosed comments in REXX and C programs.
STACK	Reads from or writes to the program stack.
STEM	Gets or sets REXX or EXEC 2 variables with the specified stem.
VAR	Gets or sets a REXX or an EXEC 2 variable.
VARLOAD	Sets a REXX or an EXEC 2 variable.
File Input/Output	
<	Reads the contents of a CMS file.
>	Creates or replaces a CMS file.
>>	Creates or appends to a CMS file.
AFTFST	Provides information about open files.
FILEBACK	Reads the contents of a CMS file backward.
FILEFAST	Reads the contents of a CMS file or creates or appends to a CMS file.
FILERAND	Reads specific records or ranges of records from a CMS file.

CMS Pipelines Summary

Table 3. What Each Stage Does (continued)

Stage	Task Performed
FILESLOW	Reads the contents of a CMS file or creates or appends to a CMS file beginning at a specified record number within the file.
FILEUPDATE	Replaces records in a CMS file.
FMTFST	Formats a file status table (FST) entry.
GETFILES	Reads a list of CMS files.
MDISKBLK	Reads blocks of data from an accessed CMS minidisk.
STATE	Determines whether the specified file or files exist.
STATEW	Determines whether the specified writable file or files exist.
Miscellaneous	
?	Displays a message that describes how to obtain CMS HELP information for CMS Pipelines.
AHELP	Provides the author's help information on CMS Pipelines messages, stages, pipeline subcommands, related host commands, syntax variables, tutorials, and miscellaneous topics.
BUFFER	Accumulates all records in a single stage not passing any on until all have been received.
CASEI	Selects records relative to a target character string regardless of the case representation of the character string.
CONFIGURE	Tailors CMS Pipelines in your own z/VM logon session using pipeline configuration variables.
COPY	Delays by one record the passing of records from the input stream to the output stream to prevent a pipeline stall.
COUNT	Counts bytes, blank-delimited character strings, or records.
CRC	Computes a checksum on its input stream records using a CRC algorithm.
DUPLICATE	Writes each input record in addition to the specified number of copies of each input record.
ELASTIC	Puts a sufficient number of input records into a buffer to prevent a pipeline stall.
ESCAPE	Inserts escape characters in records.
GATE	Causes portions of a pipeline to end.
HELP	Displays help information on CMS Pipelines messages, stages, and pipeline subcommands.
INSTORE	Reads records from its input stream into storage and writes a single record containing only the pointers to the records in storage.
LDRTBLS	Runs a compiled user-written stage that has been loaded with the CMS LOAD command.
MACLIB	Generates a macro library from COPY file members.
NUCEXT	Resolves an entry point from a nucleus extension to find a compiled user-written stage to run.
OUTSTORE	Unloads a file loaded into storage by INSTORE.
PAUSE	Sends a signal from the pipeline containing the PAUSE stage to the pipeline containing the RUNPIPE stage to receive a type X'11' PAUSE event record.
PIPCMD	Issues primary input stream records as pipeline subcommands.
PREDSELECT	Copies a record from its primary input stream to either its primary or secondary output stream depending on the order of arrival of input records on its other input streams.
QSAM	Gets records from or puts records to a physical sequential data set using queued sequential processing.
QUERY	Displays one of the following: the version of CMS Pipelines, the message level, the list of messages that have been issued, or the level of CMS Pipelines.
RUNPIPE	Issues input stream records as pipelines.
SORT	Arranges records in ascending or descending order.
TIMESTAMP	Determines when a record was read.

Table 3. What Each Stage Does (continued)

Stage	Task Performed
UDP	Allows access to a TCP/IP port.
ZONE	Defines locations of the input data in records from which records are selected when using a specified stage.
Multiple Streams	
COLLATE	Matches records from two input streams and writes matched and unmatched records to different output streams.
DEAL	Writes a primary input stream record to one of its connected output streams in either sequential order starting with the primary output stream, or some other order specified on the secondary input stream.
FANIN	Combines multiple input streams into a single stream in a specified order.
FANINANY	Combines multiple input streams into a single stream. FANINANY reads an input record from any input stream that has a record available.
FANOUT	Copies primary input stream records to multiple output streams.
GATHER	Reads records from its connected input streams in either sequential or some other specified order, and writes them to its primary output stream.
JUXTAPOSE	Prefaces records in the secondary input stream with records from the primary input stream.
LOOKUP	Finds records in a reference.
MERGE	Combines records from all input streams in ascending or descending order.
NOT	Reverses the primary and secondary output streams of a specified stage.
OVERLAY	Reads a record from each input stream and merges the records read into a single record.
SYNCHRONIZE	Reads records from each of its input streams while each stream has a record available.
UPDATE	Modifies the primary input stream based on the contents of the secondary input stream.
Selecting Records	
ALL	Selects records containing a specified string or specified strings defined by an expression comprising of character strings and logical operators.
BETWEEN	Selects records between two specified targets including the records containing the target. The specified targets must begin in the first column of a record.
DROP	Discards one or more records.
FIND	Selects records that begin with a specified text.
FRLABEL	Selects records that follow a specified target including the target record. The specified target must begin in the first column of a record.
FRTARGET	Selects all records starting with the first record selected by a specified stage.
HOLE	Discards records.
INSIDE	Selects records between two specified targets not including the records containing the target. The specified targets must begin in the first column of a record.
LOCATE	Selects records that contain a specified string of characters. The characters can appear at any position within the record.
NFIND	Selects records that do not begin with a specified text.
NINSIDE	Selects records not located between two specified targets. The records containing the targets are also selected. The specified targets must begin in the first column of a record.
NLOCATE	Selects records that do not contain a specified string of characters.
PICK	Compares a field in the primary input stream record to a specified string or a second field in the record, and selects the record if the comparison satisfies the specified relation.

CMS Pipelines Summary

Table 3. What Each Stage Does (continued)

Stage	Task Performed
OUTSIDE	Selects records not located between two specified targets. The records containing the targets are not selected. The specified targets must begin in the first column of a record.
STRFIND	Selects records that begin with a specified string of characters.
STRFRLABEL	Selects records that follow a specified target including the target record. The specified string must begin in the first column of a record.
STRNFIND	Selects records that do not begin with a specified string of characters.
STRTOLABEL	Selects records that precede a specified target, not including the target record. The specified target must begin in the first column of a record.
STRWHILELABEL	Selects consecutive records that begin with a specified string. The records must be at the beginning of the input stream. The specified string must begin in the first column of a record.
TAKE	Selects one or more records from the beginning or end of the primary input stream.
TOLABEL	Selects records that precede a specified target not including the target record. The specified target must begin in the first column of a record.
TOTARGET	Selects all records up to but not including the first record selected by a specified stage.
UNIQUE	Compares the contents of adjacent records and discards or retains the duplicate records.
WHILELABEL	Selects consecutive records that begin with a specified string. The records must be at the beginning of the input stream. The specified target must begin in the first column of a record.
TCP/IP	
HOSTBYADDR	Resolves IP (internet protocol) addresses into a domain or host name.
HOSTBYNAME	Resolves a domain or host name into an IP (internet protocol) address.
HOSTID	Writes a single output record containing the default IP (internet protocol) address of the TCP/IP system in dotted-decimal notation.
HOSTNAME	Writes a single output record containing the host name of the TCP/IP system.
IP2SOCKA	Converts a human readable port number and IP (internet protocol) address to a special sixteen-byte hexadecimal record, which is used by the UDP stage to create datagrams.
SOCKA2IP	Converts a special sixteen-byte hexadecimal input record, which is used by the UDP stage to create datagrams, to a human readable port number and IP (internet protocol) address; it converts a four-byte input record to a readable IP address.
TCPCLIENT	Connects to a TCP/IP server, transmits its input records to the server, and writes data it receives from the server onto its output stream.
TCPDATA	Receives data from and transmits data to the client from the server.
TCPLISTEN	Listens for and accepts connection requests on a TCP/IP port using the socket interface to TCP/IP.
XEDIT	
XEDIT	Reads from or writes to a file that is in the ring of files currently being edited.
XMSG	Issues XEDIT messages during an XEDIT session.

PI

Table 4. What Each Pipeline Subcommand Does

Subcommand	Task Performed
ADDPIPE	Adds one or more pipelines to the set of running pipelines.
ADDSTREAM	Defines an unconnected input or output stream.

Table 4. What Each Pipeline Subcommand Does (continued)

Subcommand	Task Performed
BEGOUTPUT	Enters an implied output mode where anything directed to the subcommand environment is written to the currently selected output stream.
CALLPIPE	Invokes a subroutine pipeline.
COMMIT	Commits a stage to a different commit level.
GETRANGE	Extracts part of a record to be processed in the same way CMS Pipelines built-in programs select a substring of the input record.
MAXSTREAM	Gets the number of the highest numbered input or output stream.
MESSAGE	Displays a message at the terminal.
NOCOMMIT	Disables automatic commits performed by pipeline subcommands.
OUTPUT	Writes a record to the currently selected output stream.
PEEKTO	Reads a record from the currently selected input stream without removing the record from the stream.
READTO	Reads a record from the currently selected input stream.
RESOLVE	Determines if a stage is contained within an attached filter package.
REXX	Invokes a REXX program as a stage.
SCANRANGE	Establishes a token that is used by GETRANGE to parse an argument string containing an inputRange specification.
SCANSTRING	Parses an argument string containing a delimitedString specification in the same way that CMS Pipelines built-in programs scan their arguments when a delimitedString is specified.
SELECT	Selects a stream.
SETRC	Sets a return code.
SEVER	Disconnects from the currently selected stream and restores the previous connection, if any.
SHORT	Connects the currently selected input stream to the currently selected output stream.
STAGENUM	Gets the relative position of a specified stage in a pipeline of the primary stream.
STREAMNUM	Gets the stream number of a specified stream.
STREAMSTATE	Gets the state of a specified stream.

Table 5. What Each Assembler Macro Does

Assembler Macro	Task Performed
PIPCMD	Issues primary input stream records as built-in stages or pipeline subcommands.
PIPCOMMT	Increase a stage's commit level or receive the current aggregate return code.
PIPDESC	Set up a static area containing the constants defining such characteristics as the label referred to in the entry point table, the entry point of the program, the size of the work area and a program identifier to be used in the CMS Pipelines error messages.
PIPEPVR	Declare the address of a table of addresses required by pipeline assembler macros.
PIPINPUT	Read and consume the next record from the currently selected input stream.
PIPLOCAT	Obtain the address and length of the next record in the currently selected input stream without consuming the record.
PIPOUTP	Write a record to the currently selected output stream from a buffer.
PIPSEL	Select a stream for subsequent use by assembler macros that reference streams, such as PIPINPUT, PIPLOCAT, PIPSEVER, or PIPSHORT. The stream specified becomes the currently selected stream.

CMS Pipelines Summary

Table 5. What Each Assembler Macro Does (continued)

Assembler Macro	Task Performed
PIPSEVER	Detach the connected, currently selected stream from the stage in the pipeline that issued the PIPSEVER assembler macro.
PIPSHORT	Connect the currently selected input stream directly to the currently selected output stream.
PIPSTRNO	Specify a stream by number or name and have the stream number returned.
PIPSTRST	Determine from the status of the specified stream whether or not there is input or output data.

`PI` `end`

Note: Additional stages exist that are not documented in this book. These can be found in the *CMS/TSO Pipelines: Author's Edition*, SL26-0018.

Appendix C. Migrating to CMS Pipelines

CMS Pipelines in z/VM version 6 release 1 is based on a program that can be ordered separately. In Europe, the Middle East, and Asia, the program can be ordered as *CMS Pipelines 1.1.6 5785-RAC Program Offering*. In the USA, it can be ordered as *RPQ P81059 5799-DKF 1.1.1*. Both of these programs provide a level of function that we refer to as *CMS Pipelines 1.1.6*. For brevity, we refer to the CMS Pipelines support included in z/VM V6.1 as *z/VM CMS Pipelines*.

This appendix describes some differences between z/VM CMS Pipelines and CMS Pipelines 1.1.6. If you haven't used CMS Pipelines 1.1.6, skip this appendix.

Even though you are using z/VM V6.1, your installation may also have CMS Pipelines 1.1.6 installed. To use z/VM CMS Pipelines, make sure the disk containing CMS Pipelines 1.1.6 is *not* accessed before the S disk in the CMS search order. (Enter `listfile pipe module *` to find the disks containing CMS Pipelines.) If you want to use CMS Pipelines 1.1.6 instead of z/VM CMS Pipelines, access the disk containing CMS Pipelines 1.1.6 ahead of the S-disk.

To verify that you are using the desired version of pipelines enter:

```
pipe query version
```

The response indicates which version you are using. Another indicator of the version being used is the message prefix. Messages produced by z/VM CMS Pipelines begin with `FPL`.

Note: IBM will accept documentation indicating that a defect is causing a problem in what IBM considers an unsupported environment for z/VM CMS Pipelines. IBM does not guarantee service results or represent or warrant that any errors will be corrected.

Terminology Differences

The following list summarizes the major terminology differences between z/VM CMS Pipelines and CMS/TSO Pipelines 1.1.6:

- In CMS/TSO Pipelines 1.1.6, filters, host command interfaces, and device drivers were collectively referred to as *built-in programs*. In z/VM CMS Pipelines we use the term *stage* instead. We also distinguish between stages that are included with z/VM and those that users can write. Those included with z/VM are referred to as *built-in stages*, while those users write are known as *user-written stages*.
- In CMS/TSO Pipelines 1.1.6., programs you wrote for use in pipelines were referred to as *REXX programs* or *Assembler programs*, or *REXX filters* or *Assembler filters*. In z/VM CMS Pipelines, we refer to these programs as *user-written stages*.
- In CMS/TSO Pipelines 1.1.6, functions like READTO and OUTPUT that are used in REXX filters are known as *pipeline commands*. In z/VM CMS Pipelines, we refer to these as *pipeline subcommands*, because the programs execute in a pipeline environment (similar to the XEDIT environment with its XEDIT subcommands).
- In z/VM CMS Pipelines, we use the terms *filter*, *host command interface*, and *device driver* to categorize stages. To the PIPE command, however, they are all just stages that happen to do different kinds of functions.

Writing Stages

In CMS/TSO Pipelines 1.1.6, you can write filters (stages) in several high-level languages and in Assembler language. In z/VM CMS Pipelines, REXX and Assembler are the only supported languages for writing stages. Both interpreted and compiled REXX are supported, as well as Assembler.

Existing filters that are written in C/370™, PL/I, or Assembler language may continue to run on z/VM, but these are unsupported environments.

Differences in DB2 Server for VM Support

z/VM CMS Pipelines includes the required linkages to the DB2 Server for VM front-end module ARIRVSTC. See *z/VM: CMS Planning and Administration* for instructions on preparing CMS Pipelines for use with DB2 Server for VM.

Differences in the QUERY Stage

When the LEVEL operand is specified and the primary output stream is not connected, QUERY displays a message similar to the following:

```
FPLINX086I CMS/TSO Pipelines, pppp-ppp level
```

Where:

> *pppp-ppp/pppp-ppp* is the product number
>
> *level* is the level of Pipelines code

If the primary output stream is connected, a record containing the hexadecimal value of the Pipelines level is written to the output stream.

If the VERSION operand is specified and the primary output stream is not connected, QUERY displays the following message:

```
FPLINX086I CMS/TSO Pipelines, pppp-ppp version
(Version.Release/Mod) - Generated dd mmm yyyy at hh:mm:ss
```

Where:

> *pppp-ppp/pppp-ppp* is the product number
>
> *version* is the version, release, and modification
>
> *dd mmm yyyy* at *hh:mm:ss* is the date and time the code was generated

These messages let you use QUERY to distinguish between various levels of z/VM CMS Pipelines code and CMS/TSO Pipelines code.

Changed Filter Package Execs

The PIPGFTXT EXEC and the PIPGFMOD EXEC are now to be used to build a filter package containing REXX or Assembler stages. These execs replace the PIPGREXX EXEC and the PIPLNKRX EXEC which built filter packages of only REXX stages.
- PIPGFTXT EXEC replaces PIPGREXX EXEC
- PIPGFMOD EXEC replaces PIPLNKRX EXEC

Filter packages created with the PIPGREXX and PIPLNKRX EXECS can be recreated with the PIPGFTXT and PIPGFMOD execs.

Changed Commands

The names of some stages have changed (see Table 6). However, synonyms have been defined in z/VM CMS Pipelines so that pipelines using the CMS Pipelines 1.1.6 names will continue to work.

Table 6. Name Changes

CMS Pipelines 1.1.6 Name	z/VM CMS Pipelines Name
DISKBACK	FILEBACK
DISKFAST	FILEFAST
DISKRAND	FILERAND
DISKSLOW	FILESLOW
DISKUPDATE	FILEUPDATE
SYNCHRONISE	SYNCHRONIZE
TOKENISE	TOKENIZE
TRANSLATE	XLATE

For the commands that are not part of the z/VM CMS Pipelines support and are reserved for IBM use, refer to the chapter discussing restrictions in the *z/VM: CMS Pipelines Reference*.

Note that the ISSUEMSG pipeline subcommand is reserved for IBM use. The message numbering scheme does not correspond with the z/VM CMS Pipelines messages.

Some commands and operands were announced as obsolete in CMS Pipelines 1.1.6 and are not supported by z/VM CMS Pipelines, as follows:

*   COUNTLNS is not supported.

    Use COUNT LINES instead.
*   CPASIS is not supported.

    Use CP instead.
*   NULLS and NONULLS options are not supported

    z/VM CMS Pipelines behaves as though the NULLS option is specified. In z/VM CMS Pipelines, NULLS and NONULLS are invalid options.
*   SELECT INPUT ANY is not supported.

    Use SELECT ANYINPUT instead. In z/VM CMS Pipelines, SELECT INPUT ANY selects a stream named ANY.
*   The COUNT stage does not support the STACK, LIFO, and FIFO operands.

    Connect COUNT's secondary output stream to STACK if you want the count placed on the stack for some other program to process. It is likely that you will wish to store the count in a REXX variable; use VAR or STEM to do so.
*   The PRINTMC, PUNCH, and URO stages do not support the STOP operand. (This operand is reserved for IBM use only.)
*   XTRACT is not supported.

    Use MEMBERS instead. Specify the file type as TXTLIB and asterisk as the file mode.
*   The PIPE command, ADDPIPE subcommand, and CALLPIPE subcommand do not support the STOP option. STOP is reserved for IBM use only.

The SQL stage has a default operand of COMMIT. The default when using
CMS/TSO Pipelines 1.1.6 is RELEASE.

Changed Sample Programs

The SC XEDIT macro, which aligns comments in REXX execs, is provided with
CMS/TSO Pipelines 1.1.6. It is also provided with z/VM CMS Pipelines as a sample
program, but is renamed to SCM XEDIT.

Changed Messages and Return Codes

The message numbers and message texts in z/VM CMS Pipelines are the same as
those in CMS/TSO Pipelines 1.1.6.

Operating Environments Supported by z/VM CMS Pipelines

z/VM CMS Pipelines is supported only in the CMS environment on z/VM V6.1. It is
not supported on GCS, MVS, or any other operating environments.

Appendix D. ECHONET C Source Code

Figure 318 is the C source code for a user-written program titled ECHONET. This program is a network server that echoes data until it receives an end-of-message token or a timeout occurs. Immediately after a client connection request is accepted, the greeting record (EOD) is sent to the server as an end-of-message token. The messages received by the server can be taken and echoed "as is" or can be considered a sequence of blocks, each preceded by a block length. In this case, blocks are echoed. The ECHONET C program is compiled with the DEFINE option to specify either the VM or AIX platform:

```
-DAIX on AIX platforms
DEF(VM) on VM
```

Note: If you include the value VM with the compiler option DEFINE, for example, `DEF(VM,`*other values*`)`, it compiles on your z/VM system or produces a program to run on your z/VM system. If you wish to perform a compile on an AIX system, use the option `DAIX`.

The following operands are used:

portnumber   Provides the port number used by the server to listen to clients' requests. Unless provided, the default port number 45678 will be used. If used, this option must be first.

NOGreeting   Prevents the greeting token from being sent.

TImeout *seconds*
Sets the number of seconds to wait for the next client message and then closes the connection if a message is not received.

BUFfer *size*   Sets the buffer size for messages

CLients *max_number*
Sets the maximal number of consecutive client connections before the server stops.

SF or SF4   Specifies that messages arrive in blocks. Each block is preceded with the block size in two (SF) or in four (SF4) network bytes. This prevents partial messages from being echoed because the block of messages is not echoed until all messages are collected. Echoed messages are also blocked in the same way they arrived.

```
/* ECHONET */
#define  SERV_PORT    45678                /* Set default values */
#define  TIMEOUT      3
#define  MAX_CLIENTS  12
#define  UNBLOCKED    0
#ifdef AIX                                  /* AIX header files */
  #include <sys/types.h>
  #include <sys/select.h>
  #include <sys/socket.h>
  #include <netinet/in.h>
  #include <time.h>
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 1 of 13)

```
  typedef int boolean_t;     /* Define boolean type for AIX */
#endif

#ifdef VM                    /* VM header files */
  #include <manifest.h>
  #include <bsdtypes.h>
  #include <socket.h>
  #include <in.h>
  #include <bsdtime.h>
  #include <tcperrno.h>
                             /* Define boolean type for VM */
  typedef enum {FALSE, TRUE} boolean_t;
  #define  perror(s) tcperror(s)  /* Define error function for VM */
#endif /*ifdef VM*/

/* ---------- No platform dependent code beyond this line --------- */

#include <stdio.h>      /* Header files for both AIX and VM */
#include <errno.h>
#include <netdb.h>
#include <stdlib.h>

#define BUF_LEN          512     /* Definitions */
#define SF               2
#define SF4              4
#define NODE_ADDR_LEN    16
#define NODE_NAME_LEN    64
#define MAX_OPT_LEN      64
#define HOSTENT_LEN      sizeof(struct hostent)
#define SOCKADDR_LEN     sizeof(struct sockaddr_in)
#define EOD             "EOD"

                              /* Internal function prototypes */
 int  deblock_SF( const char*, int, int, int*, char*, int*, int);
 int  deblock_size( const char*, int, int);
 int  block_SF( const char*,const int, char*, int);
 int  strnupper( char *, const size_t);
 int  abbrev( const char*, const char *, int);
 int  isConnected( void);

main (ac, av)              /* Start C program */
int ac;
char *av[];
{
  unsigned short given_port=0;
  int    saddr_len=saddr_len = SOCKADDR_LEN;
                           /* Length of the sockaddr_in structure */
  int    i, j;              /* Indices */
  int    jcl;               /* Current client index */
  int    jbuff;             /* Displacement in inbuff */
  int    msg_len;           /* Incoming message length */
  int    leftover =0;       /* Bytes in buffer not used in record */
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 2 of 13)

```
int   rec_size;            /* Outgoing block size */
int   rec_len;             /* Accrued record length */
int   blk_len;             /* Length of block ready to echo */
int   blk_type =UNBLOCKED;/* Block type unblocked */
int   try, tries =TIMEOUT;/* Timeout variable */
int   max_clients =MAX_CLIENTS;/* MAX_CLIENT variable */
int   first_msg;       /* Distinguish first message from others */
int   s, ns;               /* Socket descriptors */
                           /* Statistics variable */
int   tot_msgs, rec_bytes, blk_bytes, longest, shortest;
int   buf_size = BUF_LEN; /* Buffer length variable */
int   rc;                  /* Return code variable */

unsigned long *a;          /* Used for tracing*/
boolean_t   greeting=TRUE; /* Initial values for boolean variables */
boolean_t   connected=FALSE;
boolean_t   keep_reading;
boolean_t   force_client;
boolean_t   trace=FALSE;
char  opt[MAX_OPT_LEN]; /* Current option */
char  eod[4]    = EOD;    /* End of connection token, full word */
char  *inbuff;     /* All buffer sizes defined dynamically */
char  *record;
char  *outbuf;
char  *buffer;
                  /* Host variables */
char  srvr_nodeaddr[NODE_ADDR_LEN];
char  clnt_nodeaddr[NODE_ADDR_LEN];
char  srvr_nodename[NODE_NAME_LEN];
char  clnt_nodename[NODE_NAME_LEN];

fd_set  except_set;        /* Exceptions set */
fd_set  readfds;

struct timeval      try_period;
struct hostent      *host;  /* Structure type variables */
struct sockaddr_in  srvr_addr;
struct sockaddr_in  clnt_addr;

if (ac >= 2)                /* Ensure operands are provided */
  given_port = (unsigned short) atoi(av[1]);
else {
  printf("Usage: %s port\n", av[0]);
  printf("Otherwise port %d to listen\n", SERV_PORT);
};  /* End operand check */
                              /* Use default port if not provided */
if ( given_port == 0 )  given_port = SERV_PORT;
  printf("Port assigned to listen is %d\n", given_port);

if ( ac > 1 )              /* Set options; process operands */ {
  for (j=1; j<ac; j++)                           {
    bcopy(av[j], opt, MAX_OPT_LEN);
    strnupper(opt, MAX_OPT_LEN);
    greeting = (   abbrev("NOGREETING", opt, 3) )
             ? FALSE : greeting;
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 3 of 13)

```
          if (   abbrev("TIMEOUT", opt, 2) )
            if ((i=atoi(av[j+1]))!=0)
              tries = i;
          if (   abbrev("CLIENTS", opt, 2) )
            if ((i=atoi(av[j+1]))!=0)
              max_clients = i;
          if (   abbrev("BUFFER", opt, 3) )
            if ((i=atoi(av[j+1]))!=0 && i >= SF4+strlen(eod))
              buf_size = i;
          if (   abbrev("SF4", opt, 3) )
              /* Record must have 4 spec bytes and at least one byte */
              blk_type  = SF4;
          if (   abbrev("SF", opt, 2) )
              /* Record must have 2 spec bytes and at least one byte */
              blk_type  = SF;
          if (   abbrev("TRACE", opt, 2) )
              /* Record must have 2 spec bytes and at least one byte */
              trace  = TRUE;
        };     /* End for j=... */
      };       /* End setting options */
      if (trace) {     /* If tracing active, then print */
        printf("%s: greeting = %s\n", av[0], greeting?"True":"False");
        printf("%s: timeout  = %d\n", av[0], tries);
        printf("%s: up to %d consecutive clients\n", av[0], max_clients);
        printf("%s: buffer size used = %d\n", av[0], buf_size);
        printf("%s: blk_type = %d or <%s>\n", av[0],
                          blk_type, (blk_type==SF) ? "SF"
                                          : (blk_type==SF4) ? "SF4"
                                              : "UNBLOCKED"        );
        printf("%s: tracing is %s\n",av[0], trace?"on":"supressed");
      };      /* End print of tracing */

      /* Get local host name */

      gethostname( srvr_nodename, sizeof( srvr_nodename) );

      /* Check host name with gethostbyname */
      if ( (host = gethostbyname (srvr_nodename))  == NULL ) {
          fprintf(stderr,"%s: ", av[0]);
          perror("Unsuccessful host name resolution\n");
          exit (9);
      };   /* End checking host name */
      printf ("%s started on <%s>", av[0], host->h_name);
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 4 of 13)

```
/* Fill in srvr_addr structure to bind it to socket later */

bzero( (char *) &srvr_addr, SOCKADDR_LEN);
srvr_addr.sin_family = AF_INET;
srvr_addr.sin_addr.s_addr = htonl(INADDR_ANY);
srvr_addr.sin_port = htons (given_port);
bcopy ( host->h_addr, &srvr_addr.sin_addr, host->h_length);

/* Determine dotted decimal address of server */
bcopy( inet_ntoa(srvr_addr.sin_addr.s_addr)
      ,srvr_nodeaddr
      ,sizeof(srvr_nodeaddr)                );
printf(": %s\n", srvr_nodeaddr);
/* Create an Internet stream TCP socket */
if ( (s = socket( AF_INET, SOCK_STREAM, 0) ) == -1 ) {
  fprintf(stderr,"%s: ", av[0]);
  perror ("Cannot get a socket: ");
  exit (9);
};  /* End Internet stream TCP socket creation */
/* Bind socket to port */
if ( bind (s, &srvr_addr, sizeof(srvr_addr)) == -1 ) {
  fprintf(stderr,"%s: ", av[0]);
  perror ("Cannot bind a socket");
  exit (8);
};   /* End binding socket to port */
/* Set client limitations */
if ( ( listen (s, MAX_CLIENTS) )  == -1 ) {
  fprintf(stderr,"%s: ", av[0]);
  perror("Failed while listening");
  exit (7);
};   /* End setting client limitations */
/* Allocate memory for buffers */
if ((inbuff = (char*)calloc(3,buf_size)) == NULL) {
  fprintf(stderr,"%s: %s\n", av[0], strerror(errno));
  exit(6);
};  /* End buffer memory allocation */
record=inbuff+buf_size;
outbuf=record+buf_size;
/* Serve maximum # of clients connected at once */
for (jcl=0; jcl < max_clients; ++jcl)
{
  tot_msgs=0, rec_bytes=0, blk_bytes=0;
  longest=0; shortest=buf_size;
      /* Use 'ns' for this client */
  if ( (ns = accept (s, &clnt_addr, &saddr_len) ) == -1) {
    fprintf(stderr,"%s: ", av[0]);
    perror ("Cannot accept");
    exit (5);
  };  /* End maximum client connection routine */
  connected=TRUE;
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 5 of 13)

```
/* Determine the client name from the address accepted in clnt_addr */
    host = gethostbyaddr( &clnt_addr.sin_addr.s_addr
                         ,sizeof(struct in_addr)
                         ,AF_INET);
    if ( host == NULL )
      printf("unsuccessful client host by addr resolution\n");
    else
      printf ("\n Client's <%s> connection is accepted",
                   host->h_name);
    bcopy( inet_ntoa(clnt_addr.sin_addr.s_addr)
         ,clnt_nodeaddr
         ,sizeof(clnt_nodeaddr) );
    printf(": %s\n", clnt_nodeaddr); /* Print client IP address */
                                     /* Print port number */
    printf("   client port to strike back: %d\n", clnt_addr.sin_port);
  /* ECHONET greeting: send EOD code to hold for 'last' record */
    if ( greeting ) /* Prepare greeting phrase first if unblocked */ {
      bzero( outbuf, buf_size);
      if ( blk_type == UNBLOCKED) {
        blk_len =strlen(eod);
        bcopy(eod, outbuf, blk_len);
      } /* End preparing unblocked greeting*/
      else {          /* Prepare greeting phrase first if blocked */
        if ((blk_len=block_SF( eod, strlen(eod), outbuf, blk_type))
              != strlen(eod)+blk_type ) {
          fprintf(stderr,"%s: Cannot block greeting\n", av[0]);
          exit (4);
        };
      };    /* End preparing blocked greeting */
      if ( send( ns    /* Send greeting */
               ,outbuf
               ,blk_len           /* Just EOD, blocked or not */
               ,0          ) < 0) {
        if ( !(connected=isConnected()) ) { /* Check connection */
          close(ns);
          continue;       /* If no connection, go to next client */
                          /* Otherwise, continue */
        };
                          /* Exit if errors occurred */
        fprintf(stderr, "%s: ", av[0]);
        perror ("Greeting --> client failed");
        exit(3);
      };
      ++tot_msgs;          /* Gather statistics */
      blk_bytes += blk_len;
    };          /* End 'if' greeting*/

    /* Read messages, no longer then buf_size, into 'in' buffer and
       echo them until timeout or specific EOD message received    */

    first_msg = TRUE;  /* Assume this is first message */
    buffer =inbuff;    /* Nothing in a buffer yet */
    rec_size=0;        /* Record size is not known */
    rec_len =0;        /* Record is not collected yet */
    jbuff =0;
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 6 of 13)

```
   /* Start 'while' loop to read requests from client */
  bzero( inbuff, buf_size);
  while(TRUE)    /* Get lines from client until EOD or timeout */ {
     for ( try=0; try<tries; ++try) {    /* Set timeout loop */
        try_period.tv_sec=1; try_period.tv_usec=0;
        FD_ZERO(&readfds); FD_SET(ns, &readfds);
        if ( select( ns+1                /* Check for exceptions */
                    ,&readfds            /* Check for read request */
                    ,NULL, NULL
                    ,&try_period)  < 0 ) {
           fprintf(stderr, "%s: ", av[0]); /* Verify select */
           perror("Call to select failed");
           exit (2);
        };

        rc = FD_ISSET(ns, &readfds);
        if (rc > 0 )               /* Get data from socket */
           break;                  /* Break timeout loop */
     };

     if ( try >= tries ) {         /* Check if timeout occurred */
        printf("%s: timeout reached for the client"
               "       or client disconnected\n",av[0]);
/* If timeout/disconnect occurred, end 'while' loop and
   disconnect from client */
        break;
     };
  /* Read socket */
     if (( msg_len = read( ns
                          ,buffer+jbuff
                          ,buf_size-(buffer-inbuff)-jbuff )
                                       ) < 0 ) {
        if ( !(connected=isConnected()) ) /* Check connection */
           break;       /* 'while' loop and wait for a new client */
        fprintf( stderr, "%s: ",av[0]);
        perror( "cannot read");
/* If disconnected, end 'while' loop and wait for new client */
        break;
     };
     msg_len += jbuff;  /* Retain message lengths */
     jbuff =0;

     keep_reading = FALSE;
     force_client = FALSE;
/* Determine whether entire block arrived and put in 'inbuf' */
     do {    /* Do 'until' all blocks extracted */
        if (trace) {    /* Print trace */
           a = (unsigned long *) inbuff;
           printf("%s: got in inbuff: %08X %08X  %08X %08X\n",
                  av[0],         *a++,*a++, *a++,*a  );
        };
     /* If data is unblocked ... */
        if (blk_type == UNBLOCKED) {
           bzero( outbuf, buf_size);
           bcopy(inbuff, outbuf, msg_len);
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 7 of 13)

```
                blk_len =rec_size =rec_len =msg_len;
            } /*End unblocked data handling */
        /* If data is blocked ... */
         else /*blk_type != UNBLOCKED*/ {
           if ( rec_size == 0 )   /* Is record size known? */ {
             bzero(record,buf_size);  /* Determine record size */
             if (msg_len < blk_type ) {
               jbuff=msg_len;
                    /* Keep reading if prefix not completely read */
               keep_reading=TRUE;
               break;   /* End 'until' to resume reading from socket */
             };         /* End msg_len < blk_type */
 /* If message length is less than block type, determine record size */
             rec_size=deblock_size( buffer
                                   ,msg_len
                                   ,blk_type   );
             buffer += blk_type;
             msg_len -= blk_type;
                              /* Error checking */
             if (rec_size < 0 || rec_size > buf_size-blk_type) {
               if (trace ) {
                 fprintf( stderr, "%s: ",av[0]);
                 fprintf( stderr, "record size %d exceeds "
                           "limitation %d\n",rec_size,
                             buf_size-blk_type);
               };         /* End of tracing message */
               rec_len +=msg_len;
               bcopy(buffer, record, rec_len);
               force_client=TRUE;
               break;   /* 'until', then 'while' and go to next client */
             };         /* Insufficient rec_size */
           };/*rec_size == 0*/
     /* Deblock next buffer record */
           rec_len=deblock_SF( buffer     /* New rec_len set */
                              ,msg_len
                              ,rec_len     /* Current length */
                              ,&rec_size   /* in/out <===    */
                              ,record      /* out     <===    */
                              ,&leftover   /* out     <===    */
                              ,blk_type    /* SF or SF4 */
                            );

           if (rec_len < 0) {       /* Error checking */
             force_client=TRUE;
             break;      /* 'until', then 'while' and go to next client */
           };            /* rec_len < 0 */
 /* If incomplete record is left over, move to beginning of buffer */
           if (leftover)                               {
             bcopy(buffer+(msg_len-leftover), inbuff, msg_len);
             msg_len = leftover;
           };
           buffer = inbuff;       /* Reset the buffer */
         };                       /* blk_type != UNBLOCKED */
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 8 of 13)

```
        bzero(inbuff+leftover, buf_size-leftover);
        if (rec_len < rec_size) /* If record incomplete, keep reading */
          keep_reading=TRUE;
        else                    /* Rec ready, put it out */ {
          rec_size =0;          /* Be ready for next record */
          rec_bytes += rec_len+blk_type;
          ++tot_msgs;
          if ( rec_len > longest ) longest=rec_len;
          if ( rec_len < shortest ) shortest=rec_len;
 /* Record is ready.  Now echo it; block record before sending */
        if ( blk_type != UNBLOCKED ) {
          bzero( outbuf, buf_size);
          if ((blk_len=block_SF( record
                                ,rec_len
                                ,outbuf
                                ,blk_type)) < 0) {
                          /* Error checking */
            fprintf( stderr, "%s: ",av[0]);
            perror("unable to block a record");
            exit(1);
          };                    /* Block_SF */
          rec_len =0;
        };                      /* blk_type != UNBLOCKED */
                                /* outbuf is ready to return */
 /* If trace was requested, print 'outbuf' */
        if (trace) {
          a = (unsigned long *) outbuf;
          printf("%s: ready to echo: %08X %08X  %08X %08X\n",
                av[0],        *a++,*a++, *a++,*a  );
        };
        rc=send( ns       /* Send echo to client */
                ,outbuf
                ,blk_len
                ,0           );
        if (rc < 0) {
          force_client=TRUE;
          if ( !(connected=isConnected()) ) /* Check connection */
            break; /* End 'until', then 'while' reading messages */
          fprintf(stderr,"%s: ", av[0]); /* Error check */
          perror("Echoing failed");
          break;   /* End 'until' unblocking */
        };           /* End sending 'outbuf' */
        blk_bytes += blk_len;
      };                             /* rec_len == rec_size */
    } while (leftover!=0);           /* End of 'until' loop */
 /* If still reading, continue */
    if ( keep_reading )
      continue;                      /* 'while' so next read happens */

    if (force_client)    /* Disconnect from client if error */
      break;                  /* 'while' and go for a next client */
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 9 of 13)

```
      /* Verify the EOD */

        if (   bcmp(outbuf+blk_type, eod, strlen(eod))==0
             && blk_len == strlen(eod)+blk_type           ) {
          if (trace)  /* If trace requested, print it */
            printf("%s: end-of-data token received\n",av[0]);
          break;                              /* While to disconnect */
        };          /* Exact notice that no more messages available */
      };            /* While(TRUE) to read messages from current client */
  /* If last record size is greater than record length, report error */
    if (rec_size > rec_len)         /* Last record is incomplete */ {
      /* This is possible only for blocked records */
      fprintf(stderr,"%s: Last message is incomplete\n", av[0]);


                        /* Flush outbuf if blocking and incomplete */
      if (trace) {  /* If trace requested, print it */
        printf("%s: Incomplete %d bytes of record will flush\n",
               av[0],                    rec_len);
        printf("%s: record: >%s<\n", av[0], record);
      };
    /* If still connected, echo incomplete record */
      if ( connected )              /* Then flush incomplete message */ {
        blk_len=block_SF(record, rec_len, outbuf, blk_type);
        if ( send( ns
                 ,outbuf
                 ,blk_len        /* Send incomplete */
                 ,0           ) < 0)          {
          if ( (connected=isConnected()) ) {
            fprintf(stderr,"%s: ", av[0]);
            perror("Flushing failed");
          };
        }                         /* Send */
        else {
          blk_bytes += blk_len;
          if ( rec_len > longest ) longest=rec_len;
        };                        /* Send */
        if (trace)
          printf("%s: Last incomplete record is flushed\n",av[0]);
      };                          /* Connected */
    };                            /* Flushing */
  /* Close client connection */
    close(ns);
    printf("\n%s finished with the client #%d\n", av[0], jcl+1);
        /* Report statistics */
    printf("   Total messages: %d, bytes received: %d, sent: %d\n",
                        tot_msgs,        rec_bytes, blk_bytes);
    printf("   Shortest message received: %d, longest: %d\n",
                                    shortest,    longest);
  };                    /* All clients are done */
 /* Close server socket */
  close(s);
  printf("%s ended\n", av[0]);
  exit(0);
};        /* ECHONET ended */
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 10 of 13)

```
/* FUNCTION CALLS */
        /* ---------------- Is Client Connected --------------- */

 int
 isConnected(void)
 {
   if (  errno==ECONNRESET       /* Connection reset by peer */
      ||errno==EPIPE       ) {   /* Broken pipe */
     perror ("");                 /* Print error message */
     return FALSE;   /* End 'while' loop and wait for a new client */
   };
   if (errno != 0 )  /* If another error, print it */
     fprintf(stderr,"%s\n", strerror(errno));
   return TRUE;
 };              /* End client connection verification */

 /* -- Check for correct abbreviation operands -- */


 int
 abbrev ( const char * sample
        ,const char * var
        ,const int    up_to  )
 {
   register int j;
   int cmp_max;

   if (     (cmp_max=strlen(var)) > strlen(sample)
        || cmp_max < up_to
      ) return (0);
   for (j=up_to; j <= cmp_max; j++ )
     if ( strncmp(sample, var, j) )
       return (0);
   return (1);
 };       /* End abbreviation checking */
 /* -- Convert String to Uppercase -- */

#include <ctype.h>
 int
 strnupper( char          *s
          ,const size_t  n  )
 {
   register int j;
   int c;

   for (j=0; j<n && s[j]!='\0';++j)
     s[j]=toupper(s[j]);
   return j;
 };                          /*strnupper*/

 /* ---------------- deblock_size -------------- */

#define SF        2
#define SF4       4
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 11 of 13)

```
/* Return size of unblocking record or return negative value
   if problem exists */
int
deblock_size( const char * sample   /* Sample to cut a block from */
            ,int          sam_len /* Its length                 */
            ,int          SF_type /* Type SF or SF4 only        */
            )
{
  unsigned short len2;
  unsigned long  len;

  if (SF_type != SF &&  SF_type != SF4) return -1;

  if (sam_len < SF_type) return -1;
  switch(SF_type) {
    case SF:  bcopy(sample, (char *) &len2, SF);
              len = len2-SF;
    break;
    case SF4: bcopy(sample, (char *) &len, SF4);
              len -= SF4;
    break;
  };                         /* Switch */
  return (int) len;
};                          /* End deblock_size */
/* ----------------- deblock_SF ---------------- */
/* Return length of accrued record in process of unblocking */
int
deblock_SF( const char * sample    /* Sample to cut a block from */
          ,int          sam_len    /* Its length                 */
          ,int          rec_len    /* Accrued record length      */
          ,int *        pRec_size  /* Block size                 */
          ,char *       record     /* Where record which length  */
                             /* Must be sufficiently collected */
          ,int *        pLeftover  /* Bytes left beyond the record */
          ,int          SF_type    /* Type SF or SF4 only        */
          )
{ /* If rec_size =0, start (otherwise continue) collecting a
     record.  After adding to a record, some bytes may be left over.
     If rec_size !=0, then rec_len bytes are already collected.   */

  unsigned short len2;
  unsigned long  len;
  int            copy_len;
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 12 of 13)

```
  /* Verify SF type */
   if (SF_type != SF &&  SF_type != SF4) return -1;
                    /* Switch between SF and SF4 */
   if (*pRec_size == 0 ) {
     if (sam_len < SF_type || rec_len != 0) return -1;
     switch(SF_type) {
       case SF:  bcopy(sample, (char *) &len2, SF);
                 len = len2-SF;
       break;
       case SF4: bcopy(sample, (char *) &len, SF4);
                 len -= SF4;
       break;
     };      /* End of Switch */
     if ((*pRec_size=len)<1) return-1;
     sample += SF_type;
     sam_len -= SF_type;
   };        /* End rec_size==0 */
   if ((copy_len=*pRec_size-rec_len)<=0) return -1;
   copy_len= (copy_len < sam_len) ? copy_len : sam_len;
   bcopy(sample, &record[rec_len], copy_len);
   *pLeftover = sam_len-copy_len;
   return rec_len+copy_len;
};         /* End deblock_SF */
/* ---------------- block_SF ------------------ */

int block_SF( const char *record  /* Blocks record */
            ,const int  rec_size /* Return length */
            ,char       *buffer
            ,int        SF_type
           )
{
  unsigned short len2;
  unsigned long  len;
  if (SF_type != SF &&  SF_type != SF4) return -1; /* Verify SF type */

  if (rec_size < 1) return -1;
  len = rec_size + SF_type;
  switch(SF_type){                    /* Switch between SF and SF4 */
    case SF: if ( len > 65535) return -1;
             len2 = (unsigned short)len;
             bcopy(&len2, buffer, SF);
    break;
    case SF4: bcopy(&len, buffer, SF4);
    break;
  };           /*End of switch*/
  bcopy(record, &buffer[SF_type], rec_size);
  return (int)len;
};       /* End of block_SF */
```

Figure 318. ECHONET C Source Code for the ECHO Server (Part 13 of 13)

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, New York 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs

285

and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, New York 12601-5400
U.S.A.
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Programming Interface Information

This book primarily documents information that is NOT intended to be used as Programming Interfaces of z/VM.

This book also documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/VM. This information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

`PI`

<...Programming Interface information...>

`PI` `end`

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml

Adobe is either a registered trademark or trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Glossary

For a list of z/VM terms and their definitions, see *z/VM: Glossary*.

The z/VM glossary is also available through the online z/VM HELP Facility. For example, to display the definition of the term "dedicated device", issue the following HELP command:

```
help glossary dedicated device
```

While you are in the glossary help file, you can do additional searches:

- To display the definition of a new term, type a new HELP command on the command line:

  ```
  help glossary newterm
  ```

  This command opens a new help file inside the previous help file. You can repeat this process many times. The status area in the lower right corner of the screen shows how many help files you have open. To close the current file, press the Quit key (PF3/F3). To exit from the HELP Facility, press the Return key (PF4/F4).

- To search for a word, phrase, or character string, type it on the command line and press the Clocate key (PF5/F5). To find other occurrences, press the key multiple times.

  The Clocate function searches from the current location to the end of the file. It does not wrap. To search the whole file, press the Top key (PF2/F2) to go to the top of the file before using Clocate.

289

Bibliography

See the following publications for additional information about z/VM. For abstracts of the z/VM publications, see *z/VM: General Information*.

Where to Get z/VM Information

z/VM product information is available from the following sources:

- z/VM Information Center at publib.boulder.ibm.com/infocenter/zvm/v6r1/index.jsp
- z/VM Internet Library at www.ibm.com/eserver/zseries/zvm/library/
- IBM Publications Center at www.elink.ibmlink.ibm.com/publications/servlet/pbi.wss
- *IBM Online Library: z/VM Collection on DVD*, SK5T-7054

z/VM Base Library

Overview

- *z/VM: General Information*, GC24-6193
- *z/VM: Glossary*, GC24-6195
- *z/VM: License Information*, GC24-6200

Installation, Migration, and Service

- *z/VM: Guide for Automated Installation and Service*, GC24-6197
- *z/VM: Migration Guide*, GC24-6201
- *z/VM: Service Guide*, GC24-6232
- *z/VM: VMSES/E Introduction and Reference*, GC24-6243

Planning and Administration

- *z/VM: CMS File Pool Planning, Administration, and Operation*, SC24-6167
- *z/VM: CMS Planning and Administration*, SC24-6171
- *z/VM: Connectivity*, SC24-6174
- *z/VM: CP Planning and Administration*, SC24-6178
- *z/VM: Getting Started with Linux on System z*, SC24-6194
- *z/VM: Group Control System*, SC24-6196

- *z/VM: I/O Configuration*, SC24-6198
- *z/VM: Running Guest Operating Systems*, SC24-6228
- *z/VM: Saved Segments Planning and Administration*, SC24-6229
- *z/VM: Secure Configuration Guide*, SC24-6230
- *z/VM: TCP/IP LDAP Administration Guide*, SC24-6236
- *z/VM: TCP/IP Planning and Customization*, SC24-6238
- *z/OS and z/VM: Hardware Configuration Manager User's Guide*, SC33-7989

Customization and Tuning

- *z/VM: CP Exit Customization*, SC24-6176
- *z/VM: Performance*, SC24-6208

Operation and Use

- *z/VM: CMS Commands and Utilities Reference*, SC24-6166
- *z/VM: CMS Pipelines Reference*, SC24-6169
- *z/VM: CMS Pipelines User's Guide*, SC24-6170
- *z/VM: CMS Primer*, SC24-6172
- *z/VM: CMS User's Guide*, SC24-6173
- *z/VM: CP Commands and Utilities Reference*, SC24-6175
- *z/VM: System Operation*, SC24-6233
- *z/VM: TCP/IP User's Guide*, SC24-6240
- *z/VM: Virtual Machine Operation*, SC24-6241
- *z/VM: XEDIT Commands and Macros Reference*, SC24-6244
- *z/VM: XEDIT User's Guide*, SC24-6245
- *CMS/TSO Pipelines: Author's Edition*, SL26-0018

Application Programming

- *z/VM: CMS Application Development Guide*, SC24-6162
- *z/VM: CMS Application Development Guide for Assembler*, SC24-6163
- *z/VM: CMS Application Multitasking*, SC24-6164
- *z/VM: CMS Callable Services Reference*, SC24-6165
- *z/VM: CMS Macros and Functions Reference*, SC24-6168

- *z/VM: CP Programming Services*, SC24-6179
- *z/VM: CPI Communications User's Guide*, SC24-6180
- *z/VM: Enterprise Systems Architecture/ Extended Configuration Principles of Operation*, SC24-6192
- *z/VM: Language Environment User's Guide*, SC24-6199
- *z/VM: OpenExtensions Advanced Application Programming Tools*, SC24-6202
- *z/VM: OpenExtensions Callable Services Reference*, SC24-6203
- *z/VM: OpenExtensions Commands Reference*, SC24-6204
- *z/VM: OpenExtensions POSIX Conformance Document*, GC24-6205
- *z/VM: OpenExtensions User's Guide*, SC24-6206
- *z/VM: Program Management Binder for CMS*, SC24-6211
- *z/VM: Reusable Server Kernel Programmer's Guide and Reference*, SC24-6220
- *z/VM: REXX/VM Reference*, SC24-6221
- *z/VM: REXX/VM User's Guide*, SC24-6222
- *z/VM: Systems Management Application Programming*, SC24-6234
- *z/VM: TCP/IP Programmer's Reference*, SC24-6239
- *Common Programming Interface Communications Reference*, SC26-4399
- *Common Programming Interface Resource Recovery Reference*, SC31-6821
- *z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS*, SA76-0148
- *z/OS: Language Environment Concepts Guide*, SA22-7567
- *z/OS: Language Environment Debugging Guide*, GA22-7560
- *z/OS: Language Environment Programming Guide*, SA22-7561
- *z/OS: Language Environment Programming Reference*, SA22-7562
- *z/OS: Language Environment Run-Time Messages*, SA22-7566
- *z/OS: Language Environment Writing ILC Applications*, SA22-7563
- *z/OS MVS Program Management: Advanced Facilities*, SA22-7644
- *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643

Diagnosis

- *z/VM: CMS and REXX/VM Messages and Codes*, GC24-6161
- *z/VM: CP Messages and Codes*, GC24-6177
- *z/VM: Diagnosis Guide*, GC24-6187
- *z/VM: Dump Viewing Facility*, GC24-6191
- *z/VM: Other Components Messages and Codes*, GC24-6207
- *z/VM: TCP/IP Diagnosis Guide*, GC24-6235
- *z/VM: TCP/IP Messages and Codes*, GC24-6237
- *z/VM: VM Dump Tool*, GC24-6242
- *z/OS and z/VM: Hardware Configuration Definition Messages*, SC33-7986

z/VM Facilities and Features

Data Facility Storage Management Subsystem for VM

- *z/VM: DFSMS/VM Customization*, SC24-6181
- *z/VM: DFSMS/VM Diagnosis Guide*, GC24-6182
- *z/VM: DFSMS/VM Messages and Codes*, GC24-6183
- *z/VM: DFSMS/VM Planning Guide*, SC24-6184
- *z/VM: DFSMS/VM Removable Media Services*, SC24-6185
- *z/VM: DFSMS/VM Storage Administration*, SC24-6186

Directory Maintenance Facility for z/VM

- *z/VM: Directory Maintenance Facility Commands Reference*, SC24-6188
- *z/VM: Directory Maintenance Facility Messages*, GC24-6189
- *z/VM: Directory Maintenance Facility Tailoring and Administration Guide*, SC24-6190

Open Systems Adapter/Support Facility

- *System z10, System z9 and eServer zSeries: Open Systems Adapter-Express Customer's Guide and Reference*, SA22-7935
- *System z9 and eServer zSeries 890 and 990: Open Systems Adapter-Express Integrated Console Controller User's Guide*, SA22-7990

- *System z: Open Systems Adapter-Express Integrated Console Controller 3215 Support*, SA23-2247

Performance Toolkit for VM™

- *z/VM: Performance Toolkit Guide*, SC24-6209
- *z/VM: Performance Toolkit Reference*, SC24-6210

RACF® Security Server for z/VM

- *z/VM: RACF Security Server Auditor's Guide*, SC24-6212
- *z/VM: RACF Security Server Command Language Reference*, SC24-6213
- *z/VM: RACF Security Server Diagnosis Guide*, GC24-6214
- *z/VM: RACF Security Server General User's Guide*, SC24-6215
- *z/VM: RACF Security Server Macros and Interfaces*, SC24-6216
- *z/VM: RACF Security Server Messages and Codes*, GC24-6217
- *z/VM: RACF Security Server Security Administrator's Guide*, SC24-6218
- *z/VM: RACF Security Server System Programmer's Guide*, SC24-6219
- *z/VM: Security Server RACROUTE Macro Reference*, SC24-6231

Remote Spooling Communications Subsystem Networking for z/VM

- *z/VM: RSCS Networking Diagnosis*, GC24-6223
- *z/VM: RSCS Networking Exit Customization*, SC24-6224
- *z/VM: RSCS Networking Messages and Codes*, GC24-6225
- *z/VM: RSCS Networking Operation and Use*, SC24-6226
- *z/VM: RSCS Networking Planning and Configuration*, SC24-6227
- *Network Job Entry: Formats and Protocols*, SA22-7539

Prerequisite Products

Device Support Facilities

- *Device Support Facilities: User's Guide and Reference*, GC35-0033

Environmental Record Editing and Printing Program

- *Environmental Record Editing and Printing Program (EREP): Reference*, GC35-0152
- *Environmental Record Editing and Printing Program (EREP): User's Guide*, GC35-0151

Additional Publications

IBM ESA/370 Reference Summary, GX20-0406

Index

Special characters

A

B

C

I

identifying streams   118
IEBCOPY unloaded data set   188
ignoring characters in searches   21
IMMCMD stage
    description   160
immediate command
        setting up an immediate command handler from a
        pipeline   160
        writing immediate command arguments to a
        pipeline   161
improving performance   105, 225
indicator word   195
input connector   94, 97, 144, 145
input operand of SPECS   36
input range
    for the LOCATE stage   19
input stream
    description   4
    reading records from   88
input stream, primary   107
input stream, secondary   107
inserting data into an DB2 Server for VM table   194
inserting stages in pipelines   95, 151
interacting with CMS Pipelines from a stage   76
interactive dialogs, reading   12
interval, specifying with DELAY stage   157
ISSUEMSG stage   269
issuing CMS commands from a pipeline   53
issuing CP commands from a pipeline   53
issuing XEDIT messages during an XEDIT session   69

J

JOIN stage
    description   30
joining records   30
joining streams   117, 120

L

label
    definition   110
label definition
    description   110
label reference
    description   110
LAST operand
    of DROP   26
    of TAKE   26
LATER example exec for DELAY   156
LATER2 example exec for DELAY   157
layout of output records, defining   35
leading character, removing   33
LEFT operand
    SPECS stage   42
length, selecting records by   20, 21
LEVEL operand
    QUERY stage   268
LFD example exec for FANINANY   118

LIFO operand on the COUNT stage
    COUNT stage   269
limiting the range of CHANGE   34
limiting the range of LOCATE   19
limiting the range of XLATE   28
line
    See record
line-end character
    example   185
    specifying for BLOCK   185
    specifying for DEBLOCK   186
    using   182
line-end character format, blocking and deblocking   185
LINEND operand
    BLOCK stage   185
    DEBLOCK stage   186
LINES operand
    COUNT stage   48
linking records   30
LISTERR option
    PIPE command   238
LISTMRG example exec for MERGE   127
LITERAL stage
    description   18, 60
literals, writing to a pipeline   18, 60
literals, writing with SPECS   39
load module for a filter package, creating   227
loading data into an DB2 Server for VM table   194
loading filter packages   225
LOCATE stage
    description   18
    OR operation   117
LOCDEPT REXX example user-written stage   144
LOGIT REXX example (shows CALLPIPE)   97
long-running PIPE command   155, 172
looking for strings in records   17, 25, 129
LOOKUP stage
    description   127
lowercase characters, translating records to   27

M

machine carriage control
    converting from   174
    converting to   174
machine, service   155
macro
    assembler
        summary of   265
macro, XEDIT example
    SC XEDIT macro for aligning comments   270
    SCM XEDIT macro for aligning comments   270
    SNIP XEDIT macro   71
    STATE XEDIT (shows SUBCOM stage)   56
    TRAILER XEDIT macro   71
    WORDLIST XEDIT (shows XEDIT stage)   248
mail, electronic, spool file format for   176
maintaining relative order of records   134
manipulating output records   35
manual, reference   16
map of pipeline, redrawing   146

IBM ®

Program Number: 5741-A07

Printed in USA