PIPELINES: HOW CMS GOT ITS PLUMBING FIXED

JOHN P. HARTMANN
IBM

How CMS Got Its Plumbing Fixed

John P. Hartmann IBM FSC, Nymøllevej 85, DK-2800 Lyngby, Denmark

Abstract

An overview of pipeline concepts is followed by a description of how these concepts were adapted to single-tasking CMS, and how the implementation evolved.

The data flow model of programming is well suited for many (but not all) programming problems. *CMS Pipelines* is a simple, robust, and efficient tool to use data flow techniques in VM/CMS.

Programs running in a pipeline read and write records on a symmetrical device-independent interface. A non-trivial problem is often solved by running a number of simple programs, each doing a little bit of the big problem; the pipeline combines programs, often to accomplish tasks that are not imagined when a particular program is written. Pipelines are entered from the terminal or issued as commands in REXX programs.

CMS Pipelines has features not found in most other systems supporting pipelines:

- Multistream pipelines support any number of concurrent streams through a program; a simple example is the master-file-update paradigm.
- A program can temporarily replace itself with a subroutine pipeline.
- A pipeline is run only when all stages of it are specified correctly; the syntax of built-in filters is checked before the pipeline is started.

Introduction

When programs run in a pipeline, the output from one program is automatically presented as input to the next program in the pipeline. Each program reads its input and writes its output through a device-independent interface without concern for other programs in the pipeline. Thus the standard output from a program can be read by the standard input of **any** program.

Why *CMS Pipelines*?

- Things get done that might not be done otherwise. The ease with which standard programs are bolted together means that users can perform ad-hoc processing of their data in ways that would not be as economical with traditional programming.
- Users can solve problems by functional programming. By selecting appropriate filters, users can apply functions to a stream of data and not worry about how to perform a particular function to all records in a file.
- Code is re-used each time a program is run in a pipeline; unlike traditional software engineering, code re-use with *CMS Pipelines* requires no modification or compilation.
- Pipeline programs are device-independent. This takes the drudgery out of writing programs. All pipeline programs can use new host interfaces as soon as a single device driver is written to support the interface in question.
- A complex task is often broken into simpler tasks, some or all of which are performed by built-in programs. What programs remain to be written, if any, are often significantly simpler than a program to perform the original task directly.
- A simple, efficient interface supports REXX programs in a pipeline, bringing device-independent I/O to REXX.
- *CMS Pipelines* supports most CP/CMS devices and interfaces, many of which are not available to REXX programs using standard CMS interfaces.

Sample Pipeline

Though CP has a command to display the number of users logged on to a system, there is no command to display the number of disconnected users. Figure 1 shows how to obtain this information with *CMS Pipelines*.

The first word is the CMS command to run a pipeline. The rest of the line is a *pipeline specification* defining which programs to select and run. There are five programs in this example; they are separated by a solid vertical bar (|).

cp issues the query command to CP and writes the response to the pipeline, with a line for each line of CP response. Four users are shown on each line.

split splits lines at the commas that separate the four users. The comma is discarded.

locate selects lines with the string `'- DSC'`. This selects all disconnected users.

count counts the number of lines in the input stream. This count is the number of disconnected users because there is a line for each user, and connected users have been discarded from the file.

console copies its input stream to the terminal of the virtual machine. It also copies the input lines to the output, but in this example the output from *console* is not connected.

How does one count the number of disconnected users with standard CMS commands? With difficulty, it would appear, and certainly not without writing a program.

```
pipe cp query names|split ,|locate /- DSC/|count lines|console
123
Ready;
```

Figure 1. Sample Pipeline

Hartmann

What is CMS Pipelines

CMS Pipelines Structure

The user sees three parts of *CMS Pipelines*:

Command Parser: Scans the argument string to the PIPE command to build a control block structure describing the pipeline to run. It ensures that the pipeline specification is well formed, that all programs exist, and that the syntax is correct for those programs where a syntax description is available to the parser.

Library of Built-in Programs: Contains device drivers, filters, and many utility functions that can be selected by the parser.

Dispatcher: Starts programs and passes control between programs to maintain an orderly flow of data through the pipeline. Programs call the pipeline dispatcher to read and write the pipeline. The dispatcher runs programs as *co-routines*; control passes from one program to another only when a program calls the dispatcher to transport data.

REXX Programs

Though many tasks can be performed with a combination of built-in programs, there are bound to be times when *CMS Pipelines* does not provide the primitive function needed for a particular task. A program must be written to perform the missing function when *pipethink* (chipping sub-problems off a big problem) does not come up with a useful solution.

However, the program to be written needs only to solve one particular little problem; most of the task should be performed with built-in programs.

Programs to process pipeline data can be written in REXX, PL/I, IBM C/370, Assembler, and other languages that use Assembler calling conventions. REXX is used exclusively in the examples in this paper.

A REXX program processing data in the pipeline is stored as a disk file; it has file type REXX to distinguish it from EXEC procedures. The REXX program can be EXECLOADed or installed in a shared segment just like all other REXX programs.

The default command environment for REXX pipeline programs processes *pipeline commands* to move data from the pipeline into the program's variable pool, and to write output lines into the pipeline.

As an example of a function that is not readily made with built-in programs, consider how to display the number of terminals that are in the state between displaying the VM logo and having a user logged on. Local terminals in this state are shown with the a user ID comprised of LOGO followed by the four-digit device address.

NOTLOG REXX

```
/* Select LOGOxxxx userIDs */
signal on novalue
signal on error
do forever
   'readto in'
   parse var in user+8 '-' device .
   If user = 'LOGO'device
      Then 'output' in
end
error: exit RC*(RC¬=12)
```

The program in the sample above reads input lines into the variable `in` which is parsed to obtain the user ID and the device address. The input line is copied to the output with the `output` command when the line represents a terminal in limbo. Note that the two commands are not symmetrical: the name of the variable to receive the next input line is a literal; the variable is set as a side effect of the command. The line to write is the argument string to the `output` command. The loop terminates when either of the two commands gives a non-zero return code. The return code from the filter is 0 at normal end-of-file or the return code from the pipeline.

```
pipe cp q n|split ,|strip|notlog|console
LOGOL097 - L097
Ready;
```

It becomes cumbersome to write long pipelines on the terminal, especially when fine-tuning a suite of filters: put them into a REXX EXEC instead. Commands on the terminal are in *landscape* format (a single line); when writing EXECs, it is more convenient to write pipelines in *portrait form* with one line per program. This is the sample above in portrait form:

```
/* Limbo sample */
'PIPE',
   'cp q n|',
   'split ,|',
   'strip|',
   'notlog|',
   'console'
exit RC
```

CMS Pipelines supplies a sample XEDIT macro to convert from landscape to portrait form.

Using *CMS Pipelines* in REXX EXECs

It is easy to augment REXX EXECs with pipelines: use the PIPE command with device drivers to read and write REXX variables.

Sort: The example below sorts the contents of the stemmed array `unsorted`. *stem* reads and writes a stemmed array. The variable `unsorted.0` has the number of variables in the array; the first variable is `unsorted.1`, and so on. The result is stored in the array `sorted`.

```
'PIPE stem unsorted.|sort|stem sorted.'
```

Discovering Stemmed Variables: The device driver *rexxvars* writes the source string and all exposed variables in a REXX program into the pipeline. It writes the name and value of a variable on separate lines. The first column is the record type (`n` for a variable name); the source string, name, or value begins in column 3. Given this, finding all variables with a common stem is a matter of *find*. To find the names of all variables that have the stem `array`:

```
'PIPE',
   'rexxvars|',      /* Read all variables */
   'find n ARRAY.|', /* Names of array     */
   'spec 3-* 1|',    /* Discard type prefix*/
   'buffer|',        /* Ensure no interf.  */
   'stem vars.'      /* Store in stemmed   */
```

rexxvars Reads the names and values of all exposed variables in the REXX program.

find selects name lines for variables with stem `array`. Discard the lines with the values of the stemmed variables and information about other variables.

spec moves the name (from column 3 onwards) to the beginning of the record.

buffer stores all lines in a buffer before writing any to the output. This ensures that the variables to be set with the result do not interfere with the variables being queried.

stem writes the names of all variables with the stem `array` into the stemmed array `vars` where they can be accessed with a numeric index.

Transporting Variables Between REXX Programs: The device drivers supporting REXX variables can manipulate REXX environments prior to the one issuing the PIPE command. To copy the stemmed array `parms` from the caller to the current REXX program:

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

The number after the first stem indicates that the EXECCOMM before the current one is to be read.

Find the Caller: The first line of output from *rexxvars* has the letter `s` in the first column and the source string (the string parsed with `Parse Source`) from column 3 onwards. When *rexxvars* is applied to the environment before the current one, the first line is the source string for the caller. This can be parsed to determine the caller of a REXX program. *var* sets the variable to the first line on the input stream.

```
'PIPE rexxvars 1|take 1|var source'
parse var source 3 . . c_fn c_ft c_fm .
parse source . . m_fn m_ft m_fm .
say m_fn m_ft 'called from' c_fn c_ft c_fm'.'
```

Multistream Pipelines

Imagination sets the limits for multistream pipelines; here we show two simple examples without attempting to explain how multistream pipelines work in general. Refer to the tutorial or reference manuals for further information.

A program reads and writes the pipeline through a *stream*. When the program has access to several streams, they are named the *primary stream*, the *secondary stream*, and so on. A stream has an input side and an output side. The input side reads from the left-hand neighbour (what is before the previous |); the output side writes to the right-hand neighbour (what is after the next |).

CMS Pipelines has many built-in *selection* programs to select subsets of the input file that satisfy some selection criterion. (*locate* has already been

used.) Selection stages discard records that are not selected when the program is in a straight pipeline. With multistream pipelines, selection stages direct rejected records to the *alternate output stream*, if defined.

Count Connected and Disconnected Users: As an example, the pipeline in Figure 3 displays the count of connected users and the count of disconnected users. Figure 2 shows the topology of the pipeline.

There are two pipelines in this example: the left-hand one is the *primary stream* for the programs in it. The wide boxes represent programs that use two data streams: the right-hand pipeline is the *secondary stream* for these programs. (Both of the *count* and *change* filters read and write their primary streams.)

Some trickery is needed to transform this two-dimensional picture into a parameter string which must of nature be one-dimensional. An *end-character* separates pipelines in a pipeline specification. It ends one pipeline and begins the next. There is no default end-character; it must be declared in each multistream pipeline.

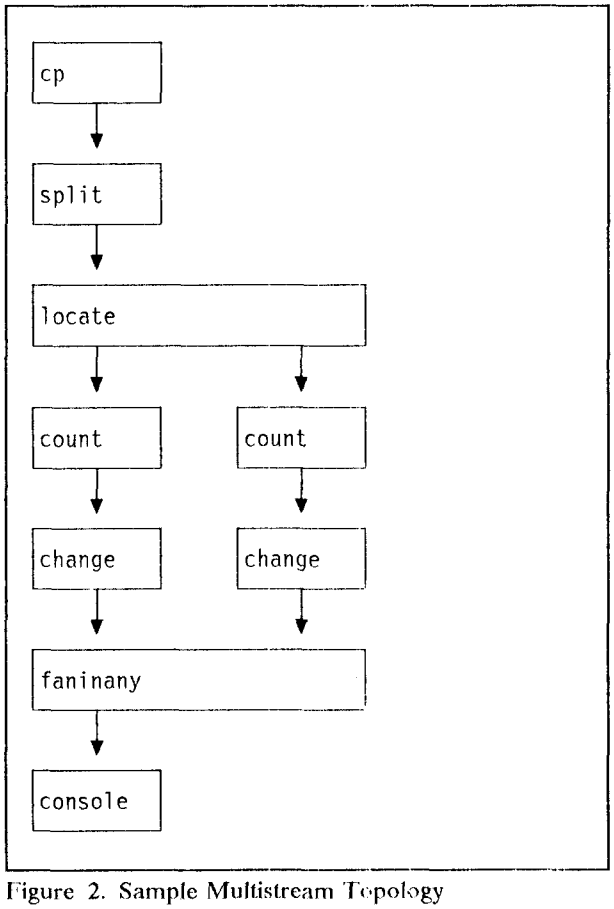


Figure 2. Sample Multistream Topology

```
/* Count logged and disconnected */
signal on novalue
address command
'PIPE (end \)',
   ' cp query names',
   '|split ,',
   '|l:locate /- DSC/',
   '|count lines',
   "|change //Disc'd: /",
   '|f:faninany',
   '|console',
   '\l:',
   '|count lines',
   '|change //Logged: /',
   '|f:'

exit RC
```
Figure 3. LND EXEC: Count Users

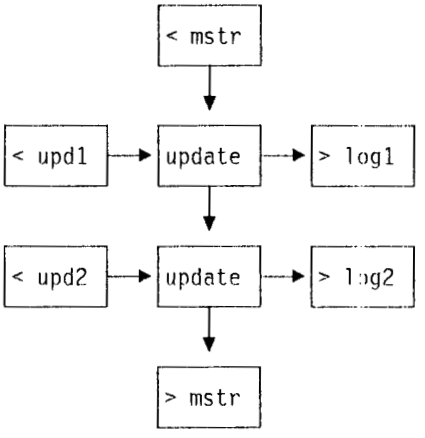
Parentheses at the beginning of the pipeline specification enclose *global options*. The end-character is one such. The backslant (\) is defined as the end-character in Figure 3.

`l:` and `f:` are *labels*. Both are used twice in this sample. The first time a label is used declares the *primary stream* for the particular invocation of the program written after the label. In the case of *locate*, it reads from the primary input stream (what is before it); it writes lines with the string to the primary output stream (what is after it). *locate* writes records without the required string to the *secondary output stream*. The secondary output stream is declared the next time the label is used (after the end-character in this sample). Whereas *locate* reads one input stream and writes to two output streams, *faninany* reads records from whichever input stream has one. *faninany* writes all records to the primary output stream. In this example it merges the lines with the count of selected and discarded records.

```
lnd
Disc'd: 130
Logged: 46
Ready;
```

Update: Many built-in programs support multi-stream pipelines. As an example, the *update* built-in program provides a subset of the function of the CMS UPDATE command. It reads the master file from the primary input stream and writes the updated file to the primary output stream. It reads the update from the *secondary input stream* and writes the update log to the *secondary output stream.*

update does not perform multilevel updates under the control of a control file. As a typical example of applied *pipethink*, *update* programs are cascaded (written one after the other) to implement multi-level updates. A controlling program reads the control file and auxiliary control file(s) to determine which updates to apply and their order.



Changing Pipeline Topology Dynamically

A pipeline program can issue pipeline commands to change the topology of its connections to other pipeline programs. The command CALLPIPE runs a *subroutine* pipeline; the program issuing the command resumes when the subroutine has completed. ADDPIPE adds a new pipeline to the set of running pipelines.

Subroutine Pipeline: Subroutine pipelines often hide the details of a task; they are the easiest way to create new pipeline filters.

In the previous examples, the sequence of *cp*, *split*, and *strip* was used over and over again. This example shows how to put these programs into a subroutine, USERS REXX, that can be called as a program.

A subroutine pipeline is likely to see more use than a cascade of filters in any one pipeline. Make sure it works in general, not just in the context where the cascade of filters comes from. In this case, the CP response is too long for the default buffer size when the system has between 400 and 500 users logged on. To ruggedise USERS REXX, a pipeline is added to query the number of users logged on, and allocate sufficient buffer space to hold the reply to the query. The second pipeline is the subroutine implementing the cascade of filters. (It also deletes lines listing virtual machines connected to the *CCS system service.)

USERS REXX

```
/* USERS REXX: Write a Line for each user */
signal on error

'callpipe',
   ' cp query users',
   '|strip',
   '|chop before 40',
   '|var users'

'callpipe',
   ' literal QUERY NAMES', /* Command    */
   '|cp' users*25+100, /* Issue CP       */
   '|nfind VSM_-', /* Ignore VTAMS       */
   '|split ,',    /* One line for each   */
   '|strip',      /* Strip leading blank */
   '|*:'          /* Pass on to next     */

error: exit RC
```

The argument to CALLPIPE is a pipeline specification like the argument to the PIPE command, with a difference. *: is a *connector* to show where to connect the input and output streams of the calling program. As used in this example, it specifies that the output stream of the subroutine pipeline is to be connected instead of the output stream of the calling program. The calling program's output stream is restored when the subroutine returns and the caller continues after the CALLPIPE command is complete.

Using USERS REXX, the combined function is performed by the command below. (It is late in the day, so the number of disconnected users has gone up since the last sample.)

```
pipe users|locate /- DSC/|count lines|console
131
Ready;
```

INCLPACK REXX

```
/* Include package files recursively */
signal on novalue

call dofile
exit

dofile: procedure
parse arg stack
do forever
   'readto in'
   If RC¬=0
      Then leave
   If left(in,7)¬==' &1 &2 '
      Then iterate  /* Comment */
   'output' in      /* Write line */
   parse var in . . fn ft fm .
   If ft¬='PACKAGE'
      Then iterate  /* Not a package */
   fid=fn'.'left(fm,1)
   If find(stack, fid)>0
      Then iterate  /* Recursion */
   'addpipe <' fn ft fm '|*.input:'
   If RC/=0
      Then exit RC
   call dofile stack fid
   'sever input'
end
If RC=12  /* EOF? */
   Then return
exit RC
```

Parallel Pipelines: The ADDPIPE pipeline command adds a pipeline specification to the current set of pipelines without suspending the program that issues the command. It can add programs, for instance, to process the input stream or divert the output stream temporarily.

As an example, INCLPACK REXX processes an input stream in the format used to describe files on the *CMS Pipelines* distribution tape (a PACKAGE file). Such a file has ' &1 &2 ' in columns 1 to 7; the file name, type, and mode are in the next 20 columns.

This program has a recursive procedure to process a file. The argument string to the procedure is the path of open package files. The loop body reads a line, checks if it identifies a file (otherwise it is assumed to be a comment that is discarded). The input line is copied to the output stream and inspected to determine if it represents a nested package file that has not already been processed in this path.

ADDPIPE puts the current input stream on a stack of dormant primary input streams for the stage and connects the primary input stream to < which reads the package file. The procedure `dofile` is called to process the package file. When done, the input stream (which is now at end-of-file) is severed. This re-instates the stream on top of the dormant stack to continue reading the file that referenced the one just done.

```
pipe < allpipe package|count lines|console
>11
>Ready;

pipe < allpipe package|inclpack|count lines|console
>126
>Ready;

pipe < allpipe package|inclpack|sort unique|count lines|console
>126
>Ready;
```

Figure 4. Processing a Package Recursively

```
pipe literal 60|dup *|delay|spec /lnd/ 1|subcom cms
```

Figure 5. Sample Event-driven Pipeline

Event-driven Pipelines

Most pipelines process lines as quickly as they are read from the host interface (for instance a tape or a CMS file). A few device drivers, however, wait for events and write a line to the pipeline when the event occurs:

- *delay* writes a line after an interval has elapsed or at a particular time-of-day.
- *immcmd* writes a line with the argument string when a particular immediate command is issued by the user at the terminal.
- *starmsg* connects to the message system service. It writes a line whenever CP presents a message or response to it.

These device drivers support pipelines in service machines to process user requests sent, for instance with SMSG, as well as authorised commands entered from the terminal when the virtual machine is connected, or sent with the SEND command from the secondary user.

The example of an event-driven pipeline in Figure 5 shows how to issue the LND command in Figure 3 on page 5 once a minute.

literal writes a literal 60 (the number of seconds to wait) into the pipeline.

dup makes an infinite number of copies of the line. (But only one at a time; this does not flood the pipeline.)

delay reads a line; the first word specifies when it must copy the line to the output. In this example it is the number of seconds to wait. The input line is copied to the output after the delay. Having written the line *delay* reads another input line and waits for 60 seconds once more. Thus, *delay* writes a line every 60 seconds.

spec is a program modelled on the COPYFILE option SPECS. As used here, it writes an output record with the literal string lnd for each input record (it does not reference fields in the input record). *spec* does not delay the record; in this pipeline it writes a record once every 60 seconds.

subcom passes input lines to the CMS subcommand environment which issues them with full command resolution. The response is written directly to the terminal by CMS.

How CMS Pipelines Works

CMS Pipelines is in two module files: PIPE MODULE is a small transient bootstrap module; the main pipeline module is PIPELINE MODULE. The main module can be disk resident or installed in a shared segment. A disk resident module is installed as a system nucleus extension; it is called from PIPE MODULE to install a PIPE user nucleus extension. The bootstrap module is not called by CMS once the main module is installed.

With this set-up, the pipeline code is protected, but CMS considers pipeline programs as user programs and recovers from an ABEND.

Filter Package: A filter package is a module file that contains filters with an entry point table defining its programs and optionally a message table for messages specific to programs in the filter package. The filter package also has a glue module that attaches it to the main pipeline module. A filter package is in a shared segment or NUCXLOADed. Once loaded, the filter package identifies itself to the pipeline module using an unpublished protocol; from then on programs in the filter package are considered an extension to the main pipeline module.

Four filter packages are installed automatically, if present, when the main pipeline module is initialised:

PIPPTFF Filters in this package replace built-in filters. This allows the replacement of some built-in programs without regenerating the main pipeline module. It also provides a convenient way to test fixes to built-in programs.

PIPSYSF System filter package. This is intended for programs to be available enterprise-wide.

PIPLOCF Local filter package. Filters available to all users in a particular system or installation.

PIPUSERF User filter package. A user can create a user filter package with private filters that are used often and thus should remain in storage.

A filter package can have any name. If a filter package is invoked as a CMS command, it installs itself as a nucleus extension (if not already one) and attaches its tables to the main pipeline module. Thus, to ensure that the contents of a filter package are available, one only has to issue the name of the package as a CMS command.

```
/* post processor */
address command
'PIPLSTPP' /* Ensure installed */
'PIPE < some listing|postproc ...'
```

Scanning a Pipeline Specification

The argument string to the PIPE command, as well as the CALLPIPE and ADDPIPE subcommands, is a pipeline specification that is processed by the parser. Having determined the over-all topology of the pipeline network, the parser resolves entry points and allocates working storage for programs that specify their requirements in a program descriptor. When the parser finds no errors in the pipeline specification, the control block structure is passed to the dispatcher for execution.

Resolve Entry Points: Entry points are resolved via *entry point tables*; each entry has the external name, flags, and a pointer.

Entry point tables are searched in this order:

1. The PIPPTFF filter package. This filter package is intended to hold replacements for built-in programs.
2. Built-in programs. These programs are in PIPELINE MODULE.
3. The PIPSYSF, PIPLOCF, and PIPUSERF function packages and other filter packages installed by the user or installation. The packages are searched in the order they are installed; by default, PIPSYSF is searched first.
4. Programs in the PIPPRV entry point table. This entry point table is intended for installation use to identify programs linked into the PIPELINE MODULE. The module shipped has no PIPPRV entry point table.

If an entry point is not resolved in any of these entry point tables, *CMS Pipelines* looks for a file with file type REXX (using EXECSTAT) and invokes the program as a REXX filter if one is found.

The entry point as resolved by look-up in an entry point table is not necessarily the first instruction of the program to run. The entry point table can specify that the entry point requires a high-level language runtime environment, or that the particular type of entry point be determined from inspection of storage at the address resolved so far.

When no high-level language is indicated, the entry can be an alternate format EXEC, an executable instruction, or a byte of binary zeros indicating an entry descriptor.

An alternate format EXEC is assumed to be a REXX filter. It is invoked with suitable parameter lists[1]. Other executable entry points are assumed to require CMS parameter lists (both extended and tokenised).

Entry Descriptor: An entry descriptor is defined by *CMS Pipelines* conventions. It has a byte of binary zero followed by three bytes of lowercase characters defining the type of descriptor:

cmd A pipeline command to be issued. The following fullword is the length of the command which follows. The command is usually CALLPIPE to invoke a subroutine pipeline to implement the function.

ept Another level of entry point table. The next word of the filter definition is looked up in the table that follows the descriptor.

---

[1] Due to the *CMS Pipelines* restriction that programs must not issue pipeline requests from commands (subroutines) that are called with CMSCALL macros, the runtime environment is called with a BALR; the runtime environment must be a nucleus extension or install itself as a nucleus extension when called (using CMSCALL) with a null parameter list.

lup A look-up routine (for instance *ldrtbls* to find an entry point in the CMS loader tables). The next word of the filter definition is passed to the look-up routine. It returns the resolved entry point address, or zero when the entry point cannot be found.

rex A REXX program that has been processed by the PIPGREXX filter to generate an in-storage program. The next word in storage is the length of the list that follows, in bytes. The program list (pairs of addresses and lengths) follows.

pip The entry address is the beginning of a program descriptor.

The entry point resolved by a second level of entry point table or by a look-up routine is inspected for an entry descriptor. These can be nested to any depth.

Program Descriptor: The program descriptor defines a built-in program to *CMS Pipelines*. It specifies attributes of the program that allow the pipeline parser to:

- Perform checks that are done by the program itself in a traditional implementation. For instance, does the program require arguments, must there not be arguments, or are arguments optional? Checking syntax before starting the pipeline means that the complete pipeline can be aborted when an error is found in the parameters to a single program.
- Allocate storage for all programs with one call to the host system storage management. The descriptor states the amount of storage to be allocated on the initial entry. Work areas for neighbouring invocations of programs are allocated adjacent; this may reduce the working set.
- Call a syntax exit, if specified, to perform further argument scan. For instance, the syntax exit can ensure that a disk file to be read does exist.
- Obtain the address of the main entry to call when no syntax check fails.

Commit Level

The commit level is an integer. A program starts on a particular commit level. The program advances its commit level to co-ordinate its progress with other programs. When a program returns on its original invocation, the return code is inspected and an aggregate return code is computed for the pipeline specification.

The programs that start at the lowest commit level are invoked first. This set of programs run until each of them returns or issues a COMMIT request to increase its commit level. The commit level is increased when there are no programs left at the original commit level. Programs on the new commit level are started only if the aggregate return code is zero at the time the commit level is reached; programs that start on a commit level are abandoned if any program has returned with a non-zero commit code at a lower level of commit. Programs that were started at a lower commit level receive the aggregate return code as the return code for the commit when the requested commit level is reached.

The convention for all *CMS Pipelines* built-in programs is that they transport data on commit level 0; most of the built-in programs start on commit level 0 as well.

The syntax exit can be considered to be commit level minus infinity.

The syntax exit must not allocate resources (for instance open files or obtain storage) because these resources are not released if some other syntax exit fails. On the other hand a program can allocate a resource on, for instance, commit level -1. It can then increase its commit level to 0. If the return code on the commit is not zero, the program can de-allocate the resource and exit; it can continue if the return code is zero.

When a subroutine pipeline commits to a level that is higher than the one of its caller, the caller commits to this higher level before the subroutine's commit completes. A subroutine pipeline can be abandoned before it commits its caller when there are errors in the subroutine; the return code can cause the caller's pipeline to be abandonned too.

Most built-in programs process records of any length. To do this, they typically process a record this way. Processing stops when a non-zero return

code is received. A positive return code indicates end-of-file; a negative one indicates a stall (deadlock).

- Preview the input record. The address and length of the record is provided. The record is not moved in storage.
- Process the record. If the output record is a subset, the address and length from the preview are modified without moving the record. A record that is modified must be loaded into a buffer in the program that processes it.
- Write the output record. An unmodified record is written from the producer's buffer; a modified record is written from the program's own buffer.
- Release the input record with a read into a buffer of length zero. This lets the producer continue.

Data Transport

CMS Pipelines transports records between pipeline programs without buffering. A record is moved in the pipeline when the left-hand side of a connection is writing and the right-hand side is reading.

The most important functions of the device-independent interface are:

- Write a line. The program provides the address and length of a buffer where the record is stored. The program is suspended until the right-hand side performs a read operation. The number of bytes read by the other side of the connection is returned.
- Read a line, moving it into a buffer or work area. The program specifies the address and length of the area into which the next input record is stored. The program is suspended until the left-hand side performs a write operation. The number of bytes stored is returned.
- Preview the next line. The address and length of the next line are returned. The program is suspended until the left-hand side performs a write operation. This function does not read a line; successive previews return the same record. The program on the left-hand side remains suspended in its write call until the record is read into a buffer or released with a read call for zero bytes.
- Select a particular stream for subsequent reads or writes, or both. The program can also select whichever input stream has a record available; in this case, it is suspended if no input stream has a record available.
- Sever a stream. The connection to the other side is broken. End-of-file is reflected on the other side.
- Short-circuit the currently selected input and output streams. The streams on the left-hand and right-hand neighbour are connected directly as if the program has never been in the pipeline. This is convenient for programs that inspect the beginning of a file to determine if any particular processing is required. Shorting the connections avoids the overhead of copying the rest of the file.

REXX Interfaces

CMS Pipelines supports REXX in two ways:

- REXX programs can process pipeline data. In this case, the program issues commands to transmit data to and from the pipeline. Such programs are started on commit level -1; they are committed to level 0 when they issue a pipeline command to transport data, or an explicit COMMIT pipeline command. Thus, if the program discovers an error in its arguments, it can return with a return code before the implied commit; this causes the pipeline to be abandoned. Likewise an error that causes a subroutine pipeline to be abandoned can be propagated to the calling pipeline which can then also be abandoned.
- Device drivers can access variables in a REXX environment that is active at the time the pipeline specification is parsed. The REXX program is passive; it performs no action to make this happen.

REXX Pipeline Commands: Because filters run as co-routines, REXX filters do not in general return to the caller in the reverse of the order they are started. REXX filters are invoked by a branch to the address in AFXEC in NUCON instead of an SVC (or CMSCALL); thus, all REXX programs in a pipeline run on the same SVC level. On MVS, REXX filters run in re-entrant environments.

This is the reason why `readto` and `peekto` (which previews the next record) are commands with side effects rather than function calls: REXX calls an external function with SVC (or CMSCALL).

Commands in the REXX filter are processed using *Non-SVC Subcommand Invocation*. REXX programs use the Address instruction to issue commands to other environments.

As REXX programs are dispatched, *CMS Pipelines* maintains the CMS subcommand stack to ensure that the topmost EXECCOMM represents the running program.

Access to REXX Variables: The address of the most current EXEC or REXX environment is obtained (using SUBCOM) when a pipeline specification is parsed. This is the base environment for all device drivers that access REXX variables. To avoid interference from REXX stages in the pipeline, device drivers branch directly to EXECCOMM using this environment (or an earlier one if requested).

Dispatcher Strategy

At the current commit level, the dispatcher maintains a stack of programs that have not started, or are ready to run. Programs that are committed to a higher level than the current one are kept on a separate list; they are moved to the dispatch stack when the dispatcher commit level is increased to the level that the program are committed to.

Initially the dispatcher stack has the rightmost program in the pipeline specification at the bottom; the leftmost program is started first.

A program runs until it calls the pipeline dispatcher to transport data or perform some other function. As an example, refer to the pipeline in Figure 1 on page 2.

cp issues the command to CP and gets the response in a buffer. It calls the pipeline dispatcher to write the first line into the pipeline. The dispatcher checks the program at the other end of the connection to see if it is ready to read the line. *split* is not waiting for input, it is ready to run and not started, so *cp* is suspended (waiting for output to be consumed) and *split* is started.

split calls the pipeline dispatcher to get the address and length of the next input line. (The line is not moved in storage.) The line is available to the dispatcher, so the information is returned and *split* is resumed. It locates the first comma in the input line and calls the dispatcher to write the part of the line up to the comma.

In the same way, *locate* is started. It inspects the line. Assuming the first line is for a connected user, *locate* calls the dispatcher to indicate that it has finished with the input line. The dispatcher makes both programs ready to run. To pump data out of the pipeline as quickly as possible, the dispatcher puts the right-hand program last on the ready stack, so *locate* is resumed once more. It calls the dispatcher to get another record and is suspended waiting for input to be made available because *split* has not yet written the next line.

split is resumed to provide the second record. This process is repeated for each record in the input file.

cp returns on the initial invocation when all lines are processed. The pipeline dispatcher severs all streams available to a program (in this case there is only the primary output stream). Severing the stream which *split* is waiting for sets return code 12 and makes the program ready to run.

split is resumed. It notes the return code meaning end-of-file and returns as well. This reflects end-of-file to *locate* which also returns. *count* gets end-of-file and writes a line with the count on its primary output stream. *console* is finally started to process the line and write the response to the terminal.

How CMS Pipelines Evolved

CMS Pipelines evolved over the 1980s. The first implementation ran on VM/System Product Release 1; the parser used the tokenised parameter list—the untokenised command string was not available to a CMS command in those days. The first built-in programs supported the console, disk files, and virtual unit record output devices. Filters were resolved from a few built-in device drivers and the CMS loader tables. A pipeline was run by calling the parser (with a BALR instruction). This implementation was convenient to write CMS user area modules.

VM/System Product Release 2 introduced NUCXLOAD to load relocatable modules from a LOADLIB into free storage as commands. A

command interface was written to support this. Because NUCXLOAD was a transient module originally, it was not practical to have a bootstrap module; an EXEC was used instead. It ensured that the pipeline module was installed in storage before invoking it.

By early 1982 it was clear to insiders that REXX would be part of VM/System Product Release 3. An interface was quickly written when it was realised that:

- The language is attractive to process data.
- The interpreter is re-entrant.
- The mechanism for Non-SVC Subcommand Invocation allows subcommands to be issued on one CMS nesting level.
- The system interfaces (after some tweaking) are suitable to maintain concurrent invocations of REXX programs.

The parser was rewritten to use the extended parameter list on VM/System Product Release 3.

There were several attempts at multistream pipelines and dynamic reconfiguration at this time. After some experimentation, the pipeline specification found its current form in the summer of 1985.

VM/System Product *CMS Pipelines* Program Offering (5785-RAC) was announced on October 6, 1986.

NUCXLOAD was made nucleus resident in VM/System Product Release 4. The PIPE bootstrap was written to avoid going through an EXEC to run a pipeline.

The program descriptor was introduced in Modification Level 2 which shipped in November 1987. XA toleration was shipped in Modification Level 3 in December 1988. Modification Level 4, shipped in October 1989, provided XA exploitation and support of PL/I and IBM C/370. Modification level 5, shipped in August 1990, provided support for commit levels and VM/ESA.

Virtual Machine CMS Pipelines RPQ P81059 was announced October 31, 1989.

Conclusion

CMS Pipelines moves CMS away from the single-task single-program model. *CMS Pipelines* is attractive because it:

- Makes the system more efficient and responsive. Passing data (in storage) between programs saves I/O operations. Running co-routines saves processor time relative to calling subroutines.
- Makes the programmer more efficient. The user and the programmer can often plug functional building blocks together without having to worry about procedural code. A solution is often expressed as a subroutine pipeline that can be called from other programs. Filters are easily added to tailor existing solutions.
- Makes programs more robust. A filter is tested out of context, and often exhaustively. It is easy to perform a regression test.
- Supports REXX as a programming language both to write command procedures that use pipelines for processing, and as programs in the pipeline processing data.
- Provides multistream pipelines. Selection filters can split a file in streams that are processed in different ways. Programs using multiple streams can be cascaded.
- Supplies a library of more than 100 built-in programs to access host interfaces and operate on data.

References

CMS Pipelines Tutorial, GG66-3158, explains *CMS Pipelines* in 15 easy chapters with many examples.

CMS Pipelines User's Guide and Filter Reference, SL26-0018, has a task-oriented guide to *CMS Pipelines* and a reference section describing built-in programs and messages.

CMS Pipelines Toolsmith's Guide and Filter Programming Reference, SL26-0020, describes multistream pipelines, the REXX interface, and the original Assembler programming interface.

CMS Pipelines Installation and Maintenance Reference, SL26-0019, describes maintenance procedures, and how to generate a filter package.