Vertical Pipelines

21st February, 1987

Brent Longborough

Travel & Transportation ISU
IBM Brazil
(55)-21-271-4172
bombeiro@vnet.ibm.com

In memory of JCMN
5th December 1952
24th November 1986

Acknowledgment

None of this would have been possible (or even useful!) without *CMS Pipelines*, Program Number 5785-RAC, brainchild of John P. Hartmann, of IBM Denmark, to whom most of the credit is due.

The deficiencies, of course, are entirely my responsibility.

| **Second Edition (February 1987)**

| This edition of this manual supports Version 2.00 of VPIPE PACKAGE.

Preface

This manual, which describes the VPipe Package available on VMTOOLS, consists of three main chapters and an Appendix.

- "Building a Vertical Pipeline" will teach you how to build and run a Vertical Pipeline, or VPipe. This is the principal chapter in this manual.
- "Vertical Pipelines with Parameters" will then show you how you can parameterise your VPipe.

These are the principal chapters in this manual.

- Should the need arise, "Interfacing with Conventional Pipelines" will show you how to integrate your VPipes into conventional Pipeline definitions.
- "Appendix A, VPipe Processing" explains how the VPipe definition is processed and run, and defines, more or less formally, the contents of a syntactically correct VPipe.

Summary of Changes

Version 2.00

VPipe Version 2.00 introduces the possibility of generating parameterised Pipelines, via Rexx expressions.

- A new chapter, “Vertical Pipelines with Parameters” , has been added.
- The description in “Appendix A, VPipe Processing” has been updated to reflect parameterised Pipeline generation.

Contents

Figures

Introduction

VPipe, or Vertical Pipelines, is a package which allows the user of *CMS Pipelines* to simplify the creation, maintenance, testing, and production use of complete and partial Pipeline definitions.

Among the advantages that VPipe brings to the user are:

- Pipeline definitions are kept in CMS files, and may include commentary;
- Pipeline stages or groups of stages may be generated by Rexx statements and expressions;
- A Pipeline definition may include other Pipeline definitions, to any depth and complexity that virtual storage and CMS SVC limitations permit;
- Pipeline definitions may be executed directly from the definition file, or incorporated into a conventional Pipeline as if they were stages;
- The user need not be concerned with a choice of end, escape, or stage separator characters.

It is assumed that the reader is familiar with the concepts described in *CMS Pipelines: User's Guide and Reference*. The use of intersecting Pipelines may also require study of *CMS Pipelines: Toolsmith's Guide and Filter Programming Reference*. Both manuals accompany the *CMS Pipelines* package.

Building a Vertical Pipeline

This chapter should show you enough to enable you to take immediate advantage of the facilities offered by VPipe.

A conventional Pipeline definition is normally built either at the console by direct command, or as a series of Rexx statements in either an Exec or a Rexx filter.

The basic idea behind VPipe is simple: instead of entering your Pipeline definition in this manner, you create a file, with filetype "VPIPE", using your favourite editor, and define the Pipeline as a series of records. Once this is done, you can execute the Pipeline definition using the CMS command "VPipeRun".

A Simple VPipe

Let's take a fresh look at the historic "Hello, World" Pipeline:

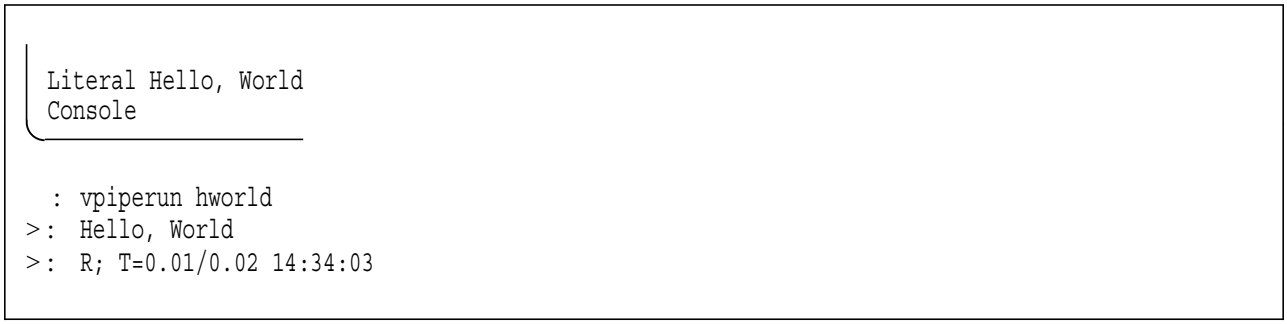

Figure 1. The Simplest Form of the Hello, World VPipe

The file, HWORLD VPIPE, consists of just two records, each of which defines a stage of the Vertical Pipeline. Both records start in column 1. The Pipeline Stage Separator does not appear in the file.

The sample console session is shown immediately below the file. Commands entered by the user are prefixed with ":", and system responses with ">:".

The results are, of course, very similar to the original "Hello, World".

Continuation Records

It isn't always convenient to squeeze a stage into a single record, especially if it needs a lot of defining (a complex Pattern stage, for example). For this reason, most stages may be broken up into more than one record. The first record of each stage must start (non-blank) in column 1. Subsequent records must have at least one leading blank, and at least one significant non-blank character. When the Pipeline is run, the stage is built from its several records by removing leading and trailing blanks and concatenating the records with a single blank between each record and the next.[1]

So we could have written "Hello, World" like this:

[1] Users of Find and other filters where trailing blanks are significant, be patient; you have not been forgotten.

```
Literal
 Hello, World
Console
```

Figure 2. The Hello, World VPipe with Continuation

or even like this:

```
Literal
         Hello,
          World
Console
```

Figure 3. The Hello, World VPipe with Silly, but Correct, Continuation

and the result would be the same as before.

Comments

I don't know about you, but I have a terrible time trying to remember what I was doing five minutes ago. The classical solution to this problem is the Comment.

Suppose, for the sake of argument, I had to provide National Language Support for the “Hello, World” Pipeline (this is not intended as a realistic National Language Support scenario). I might create BMONDE VPIPE, with comments, something like this:

```
/* This is "Hello, World" in French
/* Use in Paris at your own risk
Literal                 /* Got to be French:
 Bonjour, Monde         /* "Hello, World" in bad French
/* Output to the console
Console                 /* Write it out
```

```
  : vpiperun bmonde
>:  Bonjour, Monde
>:  R; T=0.01/0.02 16:29:22
```

Figure 4. The Bonjour, Monde VPipe, in Bad French, with Commentary

The file shown, BMONDE VPIPE, contains two sorts of commentary, flagged with the Comment Separator, “/*”, which defines the boundary between Plumbing (to the left) and Comments (to the right). For those accustomed to languages with balanced pairs of comment delimiters, it's worth mentioning that comments in VPipe extend to the end of the current record, and do not have an ending delimiter.

Records which start with /* in the first two columns are completely ignored, and do not form part of the Pipeline.

Records which define stages can also contain comments. In this case, the **last** /* in the record, and everything that follows it, is ignored. Of course, this means that you cannot use /* as part of the content of a comment; but, on the other hand, you **can** use it, without problems, as part of your Pipeline definition.

Note that all contiguous blanks in the record, immediately before the Comment Flag, are also stripped off; I'll show you how to get them back again a bit later.

Here's another example of commentary, a bit more complex. The file is XCOM VPIPE, and it operates on BMONDE VPIPE (shown above):

```
/* This VPipe removes /* comments
/* from the Bonjour, Monde VPipe
< BMONDE VPIPE *        /* Read in the file
NFind /*                /* Remove Comment Lines
Take 2                  /* Keep it short
Console

  : vpiperun xcom
>:  Literal               /* Got to be French:
>:   Bonjour, Monde       /* "Hello, World" in bad French
>:  R; T=0.01/0.02 16:29:22
```

Figure 5. Another Example with Comments

In this example, the first and fourth lines are the interesting ones.

Line 1 is a comment line, and is completely ignored, since it has /* in the first two columns.

Line 4 contains /* as part of the definition of the stage. The second (and, more important, last) /* is the comment delimiter. In this case, the second /* is not optional; the Pipeline wouldn't work correctly without it. The words after /* are useful, but optional.

Trailing blanks

This is all well and good, but, so far, we have no way of specifying a stage whose definition requires trailing blanks, since trailing blanks are stripped from the VPipe records together with trailing commentary.

To accommodate this important requirement, the Pipeline Stage Separator character, "|", is used.[2] If, after removing trailing blanks, the last character in the record is "|", it, too, is removed, and what remains, with trailing blanks included, is taken as part of the Pipeline definition.

[2] This character, vertical bar (X'4F'), may display as an exclamation mark on terminals configured for certain National Languages.

We could, for example, revise XCOM VPIPE to remove only those comment records which have at least three characters. The filter specification required is “NFind /* ” (note the single trailing blank), and the VPipe now looks like this:

```
/* This VPipe removes /* comments
/* from the Bonjour, Monde VPipe
< BMONDE VPIPE *        /* Read in the file
NFind /* |              /* Remove Comment Lines
Take 2                  /* Keep it short
Console
```

Figure 6. Trailing Blanks

Special characters

CMS Pipelines, when run without VPipe, can use up to three characters which have special effects on the Pipeline definition.

- The “Stage Separator” is used to mark the boundary between the definitions of each stage in the Pipeline. In VPipe, “|” (X'4F') is **always** used internally as the Stage Separator.
- The “End Character” is used to mark the boundary between the definitions of each Pipeline in an Intersecting Pipeline environment. In VPipe, “]” (X'6A') is **always** used internally as the End Character.
- The “Escape Character” is used to signify that the character immediately following is to keep its normal meaning, and is not to be interpreted as a Stage Separator, End Character, or Escape Character. In VPipe, “\” (X'E0') is **always** used internally as the End Character.

Within VPipe, the following considerations apply:

- When you define a VPipe, you define the structure and topology of your Pipeline without using these special characters.
- “]” and “\” may be used freely, **without additional Escape Characters**, in VPipe definitions; VPipe provides the required escape characters automatically.
- “|” is used to delimit significant trailing blanks in a VPipe record. It may also be used freely within the record, except that, where the “|” is the **last significant character** in the record, a second “|” must be added to “protect” the real one.

In consequence, although “]” and “\” are National Usage characters, and are used by VPipe itself as special characters, there are no restrictions on their usage in VPipe definitions. Nor are there any restrictions on the use of “|” (although at most one additional character may be needed for certain definitions).

There are currently no plans to allow the user to redefine these characters.

That was all rather complex - an example will, I hope, help. Here is SPECIAL VPIPE:

```
Literal 001 etc       /* Nothing special
Literal 002 etc    |  /* Trailing blank
Literal 003 \]| etc   /* Embedded specials
Literal 004 \]| etc|  /* Same again
Literal 005 ||||      /* THREE separators
Sort 1-3
Specs /"/ 1 1-* Next /"/ Next
Console
```

```
  : vpiperun special
>:  "001 etc"
>:  "002 etc    "
>:  "003 \]| etc"
>:  "004 \]| etc"
>:  "005 |||"
>:  R; T=0.02/0.03 09:14:21
```

Figure 7. Special Characters in a VPipe

Imbedded VPipe Processing

It is frequently useful to build things in small lumps. Indeed, *CMS Pipelines* is a prime example of this approach. Well, VPipes can be modular, too. If the first blank-delimited word in a record is "%", the second word is taken as the filename of an imbedded VPipe, and the corresponding Pipeline definition is included as part of the current definition. The remainder of the record is ignored.

Here are three VPipes to bring the message home:

```
/* This is CPQN VPIPE
CP QUERY NAMES   /* Get the names
Split /,/        /* Break into individual names
Strip leading    /* Take blanks off the front
```

```
/* This is CHOOSE VPIPE
Locate /BR/
```

```
/* This is IMBEX VPIPE
% CPQN       /* Query Names
Sort 1-8     /* Sort the list
% Choose     /* Only interesting ones
Console      /* Lets see them
```

```
  : vpiperun imbex
>:  BRENT   -HC09V018
>:  R; T=0.02/0.03 09:14:21
```

Figure 8. Imbedded VPipes

In this example, we run IMBEX VPIPE, which calls on CPQN and CHOOSE as imbedded VPipes. The Pipeline definition built from all this has a total of six stages.

Intersecting Pipelines

CMS Pipelines allows the user to define and run multiple Pipeline definitions which may or may not intersect. VPipes can be used, too, to define multiple Pipelines.

If, after removing commentary and a trailing "|", if any, the remainder of the record contains only blanks, the record is treated as the end of one Pipeline and the start of the next.

In a VPipe, as in a conventional Pipeline definition, intersections are defined by the use of labels. A VPipe with intersections will, then, normally contain one or more labels, with or without a stage definition in the same VPipe record.

Here, as an example, is a Pipeline definition adapted from a "Not So Simplistic Append" (in *CMS Pipelines: User's Guide and Reference*):

```
Disk FIRST FILE A   /* Read the first file
A: Fanin            /* It joins up here...
Disk LAST FILE A    /* Write the output
                    /* Second piece
Disk SECOND FILE A  /* Read the second file
A:                  /* and join to the main pipe
```

Figure 9. An Intersecting VPipe

which has the same effect as the CMS command

```
copyfile first file a second file a last file a (append
```

Vertical Pipelines with Parameters

That's all very well, but, for some tasks, it's a bit rigid. For some applications, a single VPipe might, for example, be able usefully to operate on several different files, but since the file name appears in the VPipe definition, we need to have a separate VPipe for each file that we might want to process.

Or, rather, we **would** need, if we didn't have the facilities described in this chapter.

Put shortly, it works like this: after VPipe has expanded the includes, removed comments, and stripped the correct number of trailing blanks, what is left is a stage definition. If this stage definition has, as its first character, either a single or a double quote, it is interpreted as a Rexx expression or statement, and the “real” stage is the result of that operation.

In this chapter, we'll show you how this works through examples.

Stages from Rexx Expressions

Let's create a variation on “Hello, World”:

```
"Literal Hello, World, from" userid()
Console
```

```
  : vpiperun hworld
>:  Hello, World, from BRENT
>:  R; T=0.01/0.02 14:34:03
```

Figure 10. The Hello, World VPipe, Personalised

As you can see, we've changed the first stage to a Rexx expression, which adds “from” and your user-id. This expression is evaluated at the time the Pipeline definition is **built**; when the Pipeline is **run**, the result of the evaluation is already in place.

The expression can be as complicated as Rexx will allow, and may, of course, run on to VPipe continuation lines. However, it is important to note that you **do not** need commas to indicate continuation, because the continuation is handled by VPipe, and Rexx only ever gets passed a single line.

To illustrate, here's an example of a complicated “Hello, World”:

```
"Literal Hello, World,
         from" userid() "at"
         time()
         'on' date()"."
Console
```

```
  : vpiperun hworld
>:  Hello, World, from BRENT at 16:19:31 on 21 Feb 1987.
>:  R; T=0.01/0.02 14:34:03
```

Figure 11. The Hello, World VPipe, Personalised and Complicated

Passing Parameters from VPipeRun

Your Pipeline definitions can now be made sensitive to parameters which you can pass on the “VPipeRun” command. Let's take yet another swipe at poor old “Hello, World”:

```
"Literal Hello, World," arg(1)
Console
```

```
  : vpiperun hworld in sunny Rio
>:  Hello, World, in sunny Rio
>:  R; T=0.01/0.02 14:34:03
```

Figure 12. The Hello, World VPipe, with Arguments from VPipeRun

The standard Rexx construction “arg(1)” has been used. The arguments in question are taken from those of VPipeRun; after the file name and one space as a minimum separator, everything left is taken as arguments for the VPipe.

Building Stages with Rexx Statements and Programs

But for some applications, even that isn't enough. It may be useful to perform calculations, or even execute CMS commands, while building the Pipeline definition. It may also be useful to use values from the generation of one stage as input to the generation of a subsequent stage, or even to generate multiple stages from a single input stage definition.

As we have seen, a VPipe stage definition is interpreted by Rexx when it starts with a single or double quote. The complete definition of a “Rexxable” stage definition is this:

> A syntactically correct Rexx expression, beginning with a single or double quote, and, optionally, followed by a semicolon and one or more Rexx statements, separated from each other by semicolons

How does it work?

1. The Rexx expression is evaluated and assigned to the Rexx variable S.1.

2. Then, the remaining Rexx statements are executed.
3. The variables S.1, S.2, and so on, are then inspected. Each variable must contain a stage definition, or spaces (considered a Pipeline separator), or be null.
4. The first null variable terminates stage generation of this piece of the Pipeline.

Let's have a look at some examples.

Suppose we need to configure a Pipeline to include the node-id on which we are running. One way might be something like this:

```
''';
  Address Command 'IDENTIFY ( LIFO';
  Pull . . Node .;
  S.1 = "Literal Hello, World, on" Node
Console
```

Which generates the Pipeline:

```
Literal Hello, World, on RIOVMBHQ
Console
```

with, of course, this result:

```
  : vpiperun hworld
>:  Hello, World, on RIOVMBHQ
>:  R; T=0.01/0.02 14:34:03
```

Figure 13. Hello, World with a Node Id

Programming Conventions within a VPipe

When writing Rexx programs used in a VPipe to generate Pipeline stages, you should obey certain programming conventions to ensure predictable results:

1. VPipe stages are passed to Rexx for interpretation one at a time.
2. If a stage does not start with a single- or double-quote, it is taken to be a literal representation, and is not interpreted by Rexx.
3. If a stage begins with a single- or double-quote, it is assumed to be part of a Rexx expression. Interpretation is carried out as follows: the stem S. is set to null, the string 'S.1 = ' is appended at the start of the stage, and the resultant string interpreted by Rexx. After the stage has been interpreted, the stage or stages to be generate must be in the variables S.1, S.2, ...

   Here are three examples:

Stage	'Literal' userid()
Expression	S.1 = 'Literal' userid()
Result	Literal BRENT

Stage	“Literal” userid(); S.2 = ‘Literal’ time()
Expression	S.1 = “Literal” userid(); S.2 = ‘Literal’ time()
Result	Literal BRENT
	Literal 14:47:26
Stage	‘’; Do I = 1 to 5;S.I = “Literal Test” I; End
Expression	S.1 = ‘’; Do I = 1 to 3;S.I = “Literal Test” I; End
Result	Literal Test 1
	Literal Test 2
	Literal Test 3

4. Rexx variables may be set and referenced during this process. Variables set during the generation of a given stage are available for the generation of subsequent stages.

5. Variables whose names begin with dollar sign ($, X'5B') are reserved for exclusive use of VPipe, and their use will produce undefined results.

   Note: On most terminal equipment configured for National Languages other than U.S. English, the dollar sign will appear as some other symbol.

6. The variables S.1, S.2, ..., S.n have a special significance. These variables are used to pass the generated stage definitions back to VPipe. All stem variables S. are set to null before interpreting each stage.

7. When VPipe interprets a given stage, stage generation ends when a null S. variable is found. If S.1 is null, no stages are generated at that point in the Pipeline definition.

8. If an S. variable is blank (but not null), the effect is to start a new Pipeline (just as in the case of an all-blank stage definition).

9. If the expression results in a Rexx syntax error, a stage is built in the following format:

   ```
   Syntax Error <error code> <statement>
   ```

   where <error code> is the Rexx return code plus 20000, and <statement> is the input stage definition causing the error. The effect of inserting a strange stage like this should normally cause an error when the Pipeline is executed. Here's an example of a badly defined VPipe called SYNT:

```
''2+A
```

Which generates the Pipeline:

```
Syntax Error 20041: ''2+A
```

with, of course, this result:

```
  : v synt
>:  PIPSCA027E Entry point SYNTAX not found.
>:  PIPSCA001I Executing "Syntax Error 20041: ""2+A".
>:      22 *-* 'PIPE' Vpipe(Pipename,Vargs)
>:         +++ RC(-27) +++
>:  R(-0027); T=0.01/0.02 14:34:03
```

Figure 14. How Syntax Errors Explode

10. If a Rexx Exit or Return statement is interpreted, the effect is undefined. In the current implementation, the effect is to terminate **all** Pipeline generation at that point, but this may change in future releases.

Interfacing with Conventional Pipelines

So far, I have shown you how to build and run a VPipe in isolation, using the command “VPipeRun”. However, under some circumstances, it may be necessary to use a VPipe as if it were a stage in a Pipeline being used under the direct control of the “Pipe” command.

To allow this, the package includes the filter “VPipeFil”, written in Rexx, which takes a VPipe, and inserts it into the Pipeline using CallPipe. The syntax is simple:

```
... | [Rexx] VPipeFil <filename> [<args>] | ...
```

causes the pipeline to run as if the given stage were the Pipeline segment defined in the file “<filename> VPIPE”.

The control characters used by VPipe are defined locally (inside “VPipeFil”), and do not spill over into the Pipeline you are defining. As a consequence, you may define your Pipeline with control characters chosen freely from the set permitted by *CMS Pipelines*.

Appendix A. VPipe Processing

The VPipe package itself uses *CMS Pipelines* to transform a VPipe definition file into a conventional Pipeline definition. This transformation is performed by the filter "VPipeCan" (part of the package). The Pipeline which does this is also provided in the form of a sample VPipe.

It may help to understand the apparently complicated syntax of a VPipe file if you understand the steps that are taken to convert it into a conventional Pipeline definition.

How a VPipe is Built

When a VPipe definition is processed by "VPipeRun" or "VPipeFil", the following sequence is followed:

1. The VPipe is processed sequentially, record by record.
2. Each record whose first blank-delimited word is "%" is expanded by including the contents of the VPipe file named by the second word in the record. This process is continued to whatever depth of nesting is necessary, unless a recursive imbed is found.
3. All records with "/*" in the first two columns are discarded.
4. If a record contains one or more copies of the string "/*", the last copy, and everything to its right, is removed. The last character in the record is now that which was immediately to the left of the deleted "/*".
5. All trailing blanks (X'40') are removed. If the record contained nothing but blanks, a single blank is left.
6. If the last character in the record is "|" (X'4F'), it is removed.
7. Each records with leading blanks has all but one removed. If the record is not completely blank, it is then concatenated to the previous record.
8. If the record begins with a single (') or double (") quote, it is evaluated as a Rexx expression or statement. The results of this evaluation, or the values assigned to the stem variable :"S." are emitted as stage records.
9. All Stage Separator, End, and Escape characters are given an Escape character as prefix.
10. Records consisting of blanks only are converted to single End characters.
11. Records with leading blanks have all but one of them removed.
12. Records without leading blanks are prefixed with a Stage Separator.
13. The records are now concatenated together as a single record.
14. The stage separator before the first stage is removed.
15. The record is broken into stages, by splitting it before each End, and after each Stage Separator character that is not escaped.
16. Null Pipelines are removed.

At this point, the pipeline definition has been reformatted so that:

- Rexx expressions and statements have been evaluated and incorporated into the Pipeline definition.
- Each record is either a complete stage, ending with a Stage Separator if required, or an End character to mark the boundary between two Pipelines.

- All End, Stage Separator, and Escape characters used in stage definitions have been escaped.
- If all the records are concatenated together, a syntactically correct Pipeline definition is the result, satisfying the element <pipelines> in the Figure “Syntax of a Pipeline Specification” in *CMS Pipelines: Toolsmith's Guide and Filter Programming Reference*.

Syntax for VPIPE files

```
<sep>     ::= The character “vertical bar” (X'4F')
<space>   ::= The “space” character (X'40')
<spaces>  ::= <space> | <space><spaces>
<quote>   ::= " | '

<rexxexp> ::= <quote> <rest of Rexx expression>
<rexxstm> ::= ; [<Rexx statement>]
<rexxbit> ::= <rexxexp> | <rexxbit> <rexxstm>

<pipesep> ::= <spaces>
<stage>   ::= <text> | <rexxbit>
<contin>  ::= <spaces> <text> | <spaces> <rexx fragments>
<comments> ::= /* <remainder of record>

<note>    ::= /* <anything>
<element> ::= [<pipesep> | <stage> | <continuation>]
          [<sep>] [<spaces>] [<comments>]
<include> ::= [<spaces>] % <spaces> <filename> <spaces> <anything>

<record>  ::= <note> | <element> | <include>
```