Writing Assembler Filters for Use with *CMS Pipelines*

John P. Hartmann

IBM Danmark A/S
Nymøllevej 85
DK-2800 Lyngby, Danmark
.jh (from VMSHARE)
john@vnet.ibm.com (from Internet)
JOHN at VNET (from BITNET)
DKIBMJPH (from IBM Mail)

SHARE 84
Sessions 2610 and 2611
Wednesday March First 1995

Abstract

CMS Pipelines implements the dataflow model of programming. It passes records through programs in a multistream topology. This allows for selective processing of records based on their contents, or based on the previous history of input data.

Though *CMS Pipelines* contains a diverse set of built-in filters from which the pipeline programmer can construct applications, custom filters must be written from time to time. The language of choice for such filters is REXX, but when large amounts of data are to flow through the filter, the programmer may be forced to look at writing the filter in Assembler. This paper explains how.

This paper contains:

- An overview of the mechanics of writing a filter package for use with *CMS Pipelines*.
- A cookbook showing five selected simple, but non-trivial, ready-to-use Assembler filters.
- Notes, gotchas, and other items of information that may be useful to the prospective Assembler filter programmer.

Overview

A *filter* is a program that processes data, which it reads from the pipeline, to produce results, which it writes back into the pipeline.

To write an Assembler filter, you must obviously write the actual code to perform the function that you wish to accomplish. It is possible to run a filter directly out of the object module that the Assembler produces, but it is more user friendly to package Assembler filters into a *filter package*. This lets you manage many filters as a single module file. Thus, to be able effectively to write *CMS Pipelines* filters in Assembler, you must also understand about filter packages.

A filter package is a special type of module, which is installed in storage as a nucleus extension and which attaches itself to the main pipeline module as a logical extension to the contents of *CMS Pipelines*.

A filter package contains:

- The programs that implement the filters.
- An *entry point table*, through which *CMS Pipelines* is able to resolve the filters in the package.
- A *glue module*, which attaches the filter package's entry point table to the *CMS Pipelines* search order.

To create an object module that can be included in a filter package, the program must first be processed in some language-dependent way:

- Assembler programs are processed by Assembler H or the High-level Assembler to generate a normal object module.
- REXX programs can be compiled by the REXX compiler using the OBJECT option to produce an object module.
- REXX programs can also be processed by a *CMS Pipelines* utility, which generates an object module that contains the program in a form that can be passed to the interpreter. Optionally, comments and redundant blanks can be removed when creating such an object module. It is, however, important to remember that even though the REXX program is stored in an object module, it is still interpreted when it is run.

Some "magic" module names are defined, for which *CMS Pipelines* automatically attaches the filter package when it initialises itself. Other modules must be attached by invoking them as a CMS command.

How to Write the Assembler Program

CMS Pipelines provides the filter with a complete operating environment where most issues have already been dealt with:

- Working storage is allocated by *CMS Pipelines*; the filter is inherently re-entrant.
- Arguments are easily scanned using the *CMS Pipelines* services for parameter scanning and verification.
- Data are easily read and written using *CMS Pipelines*'s dispatcher services.
- Buffers are managed by *CMS Pipelines*; there is usually no need to perform one's own storage management.

- Structured programming is encouraged by a set of macros that implements the standard IF-THEN-ELSE, looping, and multiway selection functions.

It is, of course, entirely your choice whether you will wish to take advantage of these macros, in particular since many of them are not formally documented by IBM. You may wish to make do without the macros that implement structured programming, particularly if the macros do not suit your tastes. But not using the automatic storage management that *CMS Pipelines* offers may mean that the mechanics of the program becomes much more difficult to handle.

The Contents of an Assembler Filter

An Assembler program that contains a *CMS Pipelines* filter consists of three parts:

- The front matter, which sets up required global symbols and identifies the source module.
- One or more procedures, which implement the logic of the filter(s).
- The back matter, which makes up the end of the file.

The front matter and the back matter require four macros in total. They are probably best thought of as black boxes, which can easily be generated by an XEDIT macro.

The Listing File—PRINT NOGEN

The listing file produced when assembling a *CMS Pipelines* program is intended to be processed by a post processor, which is shipped in source form in DMSPPP. This post processor does not support the listing format used by the High-level Assembler; you may be better off suppressing the generated instructions with PRINT NOGEN.

Example 1, Truncate after 20 Bytes

This section walks through a filter that truncates records after twenty bytes of input. Because it writes an unmodified subset of its input records, the filter does not need to buffer the record; that is, it need not read the record into a private area of storage. Thus, the example in Figure 1 on page 3 is one of the simplest possible non-trivial filters. The filter is shown in its entirety below; the following sections describe each statement.

```
*
* Front matter
*
 COPY    DMSPIPID             Get component ID
FLTPKG   MODBEG FREETYPE=NONE,CNOTE=NO
 DMSPDEFS VECTOR=R9
*
* Procedure containing filter code
*
CHOP20   PROC  DCL=NO,SAREA=STACK,ENTRY=NO
 REPEAT  ,                    Begin main loop
   PIPLOCX ,                  Peek at the next record
   C     R0,=F'20'            Too long
   COND  HIGH,LA,R0,20        Truncate if so
   PIPOUTX ,                  Write output record
   PIPINPUT (,0)              Consume input record
 UNTIL   NOTZERO
 PEXIT   RC=(R15)
&MODULE.C2 PIPDESC FP=NO,ARGS=NO,STREAMS=1,STOPABLE=YES
 PROCEND ,
*
* Back matter
*
 MODEND  ,
```

Figure 1. Truncate after 20 Characters

The Front Matter

```
 COPY    DMSPIPID             Get component ID
```

This line copies a few global variables into the open code. The variables are used by subsequent macros. They must be set for successful assembly; thus, this statement must be the first non-comment statement in a module.

```
FLTPKG   MODBEG FREETYPE=NONE,CNOTE=NO
```

This macro defines the beginning of the module proper. The label field specifies the name of the module's control section. The first keyword operand specifies that storage is managed by *CMS Pipelines*. The second operand suppresses an IBM copyright notice, which would otherwise be generated.

The macro generates the START assembler instruction as well as an eye-catcher that shows the module name and the time of assembly.

```
 DMSPDEFS VECTOR=R9
```

This macro instruction specifies that the program will keep the address of the *pipeline services transfer vector* (PSTV) in general register 9. The PSTV is provided by the pipeline dispatcher in general register 9 when the stage is started. It contains the entry point addresses for the *CMS Pipelines* service routines.

In addition DMSPDEFS generates many equates, in particular, the ones usually employed for the general registers.

The Procedure

In general, a procedure contains three parts:

- The procedure work area. An instance of the work area is allocated when the procedure is entered; it is de-allocated when control returns from the procedure. That is, each invocation of the stage has its own private procedure work area.
- The procedure body. The programmer writes the executable code for the procedure body. *CMS Pipelines* generates appropriate prologue and epilogue to save and restore registers; it also generates code in the prologue to initialise variables in the procedure work area. The procedure body is re-entrant and is shared by all invocations of the filter; it can even be in a shared segment.
- The static constants area. This area contains read-only constants and machine instructions that are the subject of the EXECUTE instruction. The most notable item in the static area is the *program descriptor*, which is a control block that describes the stage to *CMS Pipelines*. The static area also contains the constants that might be required to initialise the procedure work area; these are generated automatically by *CMS Pipelines*. There is one copy of the static area for all invocations of the stage; it is an appendage to the procedure body.

The three parts are delimited by four macros, as shown in Figure 2:

```
NAME     PROC  ,
* Procedure work area
 PBEGIN  ,
* Procedure body
 PEXIT   ,
* Static constants
 PROCEND ,
```

Figure 2. Structure of a Procedure

The procedure work area can be elided, as can the static constants area. An outer procedure can have more than one procedure body.

```
CHOP80   PROC  DCL=NO,SAREA=STACK,ENTRY=NO
```

The PROC macro declares the beginning of a procedure. The label field contains the name of the procedure.

Normally, the PROC macro would be followed by the procedure work area, but in this case, the work area is suppressed by the operand DCL=NO.

The following operand (SAREA=STACK) must be specified this way for *CMS Pipelines* filters that are defined by a program descriptor, as this filter is. It means that the address of a work area is provided in general register 1 by the caller, which is the *CMS Pipelines* dispatcher.

The last operand (ENTRY=NO) suppresses an external entry point for the procedure. No entry is generated, because the procedure is referenced from the program descriptor, which is an external entry point.

```
REPEAT  ,                       Begin main loop
```

The REPEAT macro marks the beginning of the main loop for the filter. It is paired with an UNTIL macro, which closes the loop. Control comes back here for each record to be processed.

```
  PIPLOCX ,                     Peek at the next record
```

The PIPLOCX macro calls the *CMS Pipelines* dispatcher to get a peek at the next input record without consuming it. (PIPLOCX performs the same function as the pipeline command PEEKTO in a REXX filter.) General register 1 is set to the address of the record and general register 0 is set to the length of the record.

PIPLOCX contains a PIPLOCAT macro, which has been augmented by testing of the return code. If the return code is negative, PIPLOCX exits with the return code it received from *CMS Pipelines*; this would indicate a stalled pipeline. The pipeline dispatcher sets return code 12 at end-of-file. PIPLOCX then goes directly to the procedure exit with return code 0, because no error was found.

You can do that in this case, because no additional processing is required after end-of-file is met. In particular, no files need be closed and no resources need to be deallocated.

```
  C     R0,=F'20'               Too long
```

Test if the input record is too long.

```
  COND  HIGH,LA,R0,20           Truncate if so
```

Truncate the record if it is longer than twenty bytes.

```
  PIPOUTX ,                     Write output record
```

Write the possibly truncated record to the output. (PIPOUTX performs the same function as the OUTPUT pipeline command in a REXX filter.)

When it is written without operands, PIPOUTX expects the address of the record to be in general register 1 and its length to be in general register 0. In this example the address and length are unchanged from the PIPLOCX macro that set them, unless the record was truncated.

PIPOUTX issues the PIPOUTP macro and tests the return code in the same way that PIPLOCX does. This allows end-of-file to propagate backwards.

```
  PIPINPUT (,0)              Consume input record
```

PIPINPUT reads and consumes a record from the input stream in the same way that the READTO pipeline command consumes an input record in a REXX filter.

The second sub-operand specifies that the read buffer contains zero bytes. Performing a zero-length read is the idiomatic way to consume an input record that has been peeked at. Of course, the record should not be consumed before the derivative output record has been written.

This program does not *delay the record*, because it uses a peek/output/read cycle.

```
UNTIL   NOTZERO
```

Return to the REPEAT macro unless the return code on the consuming read is non-zero. A non-zero return code on a consuming read is an “impossibility”.

```
PEXIT   RC=(R15)
```

Return to the pipeline dispatcher at end-of-file. The return code is already in general register 15.

```
&MODULE.C2 PIPDESC FP=NO,ARGS=NO,STREAM=1,STOPABLE=YES
```

This macro generates the *program descriptor* for the filter. The label is FLTPKGC2, because the &MODULE. variable is set by MODBEG.

The keyword operands specify:

- That the filter does not use the floating point registers; thus, they do not need to be saved and restored. (FP=NO) There is a significant performance gain by specifying FP=NO; so much that it is the default for the PIPDESC macro.
- That no arguments are allowed. *CMS Pipelines* infrastructure will reject an invocation that specifies a non-blank argument string. (ARGS=NO)
- That only one stream is allowed. *CMS Pipelines* infrastructure will reject an invocation that has a secondary set of streams defined. (STREAMS=1)
- That the stage is *summarily stopable*. This authorises the *CMS Pipelines* dispatcher to terminate the stage when it has determined that the input or the output is at end-of-file. (STOPABLE=YES)

The placement of the program descriptor defines the procedure body to run; put the program descriptor in the static area of the procedure that performs the filter’s function.

```
PROCEND ,
```

The end of the procedure.

The Back Matter

```
 MODEND  ,
```

The end of the module. Do not use an END card; one has already been generated.

Assembling and Testing a Filter

Use the standard CMS macro libraries when you assemble the filter:

```
global maclib dmsgpi dmsom
hasm fltpkg
load fltpkg
pipe literal a line|ldrtbls fltpkgc2|console
```

To test the filter right out of the object module without first generating a filter package, you LOAD the object module and use *ldrtbls* to resolve it via the CMS loader tables.

Example 2, Coerce to 20 Bytes

The example in Figure 1 on page 3 serves as a convenient base to develop a filter that coerces the input record into a fixed format. In this example, the record length is fixed at twenty bytes. Longer input records are truncated and shorter ones are padded with asterisks.

Only the procedure is shown, because it is inserted after the procedure in the module shown in Figure 1 on page 3.

```
COERCE20 PROC  SAREA=STACK,ENTRY=NO
COERCE_REC DS  CL20
 PBEGIN  ,
 REPEAT  ,                    Begin main loop
   PIPLOCX ,                  Peek at the next record
   C     R0,=F'20'            Too long
   IF    LOW
     LA       R14,COERCE_REC Get buffer
     LA       R15,L'COERCE_REC ... and length
     LR       R4,R1          Get record
     LR       R5,R0
     ICM      R5,8,=C'*'     Pad with stars
     MVCL     R14,R4         Load record
     LA       R1,COERCE_REC  Address output record
   FI    ,
   LA    R0,20                Load fixed length
   PIPOUTX ,                  Write output record
   PIPINPUT (,0)              Consume input record
 UNTIL   NOTZERO
 PEXIT   RC=(R15)
&MODULE.F2 PIPDESC FP=NO,ARGS=NO,STREAMS=1,STOPABLE=YES
 PROCEND ,
```

Figure 3. Coercing the Record

The differences from the previous example are:

- A procedure work area is allocated to contain the output record in the area that has the label COERCE_REC. This is needed, because the record can potentially be modified; a filter is not allowed to modify the producer's record. That is, the filter should not modify the record that it has peeked, because it is still in the producer's buffer. This cannot be enforced; nor can non-conformance be detected.
- If the record is shorter than twenty bytes, the contents of the input record are loaded into the buffer and padded with asterisks. Note that the output record is known to be shorter than 16M (it is twenty bytes); in general, you must be very careful to handle records that are longer than 16M.
- The output record is always twenty bytes.

Though the example looks obvious, you should take a while to look at this statement from the work area:

```
COERCE_REC DS  CL20
```

This reserves a twenty-byte area in the procedure work area for use as a record buffer. It looks deceptively like a Define Symbol Assembler instruction, but *CMS Pipelines* has done one of its magic and mirrors tricks here; the DS instruction has been replaced by a macro through the mechanism known as OPSYN.

Thus, *CMS Pipelines* is able to save the value of constants that are defined in the procedure work area, had there been any, in the static section so that the prologue can initialise the variables in any particular instance of the work area. This is clearly a productivity booster, because you do not have to code the constants twice. But there are a few caveats:

- If possible, use only DC and DS instructions in the work area, or macros that expand to such instructions.
- ORG and other Assembler instructions are not recommended, but if you must use them, surround them by DS instructions for zero-length fields. This defeats the optimising step that tries to initialise several DCs in one move instruction.
- Do not use machine instructions in the work area; they will not be initialised. If you need to modify a program dynamically, move a static prototype into the work area yourself.

Example 3, Truncate at Specified Length

The next example shows a variation on Figure 1 on page 3 to allow the maximum record length to be specified as an argument; a default is used when the argument string is blank.

This example is, by far, the most complicated of the examples in this paper. It shows how to specify a *syntax exit* for the filter. The syntax exit is called by the *CMS Pipelines* scanner after it has resolved the filter, but before the pipeline is started. In this example, the syntax exit stores the value of the argument into the work area, where it can be used by the main loop.

The procedure in Figure 4 on page 9 contains two procedure bodies. *CMS Pipelines* calls them with the same work area address in register 1 and it does not disturb the work area between the two calls. Thus, the syntax exit can leave a value for the main filter to use.

```
CHOPVAR  PROC  SAREA=STACK,ENTRY=NO
CHOPVAR_LENGTH DS  F          To hold the length
 PBEGIN  EP=CHOPVAR_SYNTAX,ENTRY=NO
 PIPWORD ,                    Scan a word
 IF      ZERO                 Nothing there.
   LA    R0,15                Take a default
 ELSE    ,
   PIPDECWD ,                 Convert to decimal
   PIPERM 58,EXIT,COND=NOTZERO Trouble converting?
   LTR   R0,R0                Must be positive
   PIPERM 66,EXIT,COND=NOTPOSITIVE
 FI      ,
 ST      R0,CHOPVAR_LENGTH    Save for main procedure
 PEXIT   RC=(R15),REGS=(R2,R3)
*
 PBEGIN  EP=CHOPVAR_GO,ENTRY=NO,INITWA=NO
 REPEAT  ,                    Begin main loop
   PIPLOCX ,                  Peek at the next record
   C     R0,CHOPVAR_LENGTH    Too long
   COND  HIGH,L,R0,CHOPVAR_LENGTH Truncate if so
   PIPOUTX ,                  Write output record
   PIPINPUT (,0)              Consume input record
 UNTIL   NOTZERO
 PEXIT   RC=(R15)
&MODULE.CV PIPDESC FP=NO,STREAMS=1,STOPABLE=YES,                      *
               SYNTROUT=CHOPVAR_SYNTAX,                               *
               SYNTAX=(CALLSYNT,DONE)
 PROCEND ,
```

Figure 4. Truncating at a Specified Length

The main procedure body (the second procedure body) is almost identical to the one in the first example; it will not be discussed further. Defining the procedure work area is covered in the second example.

The point of this example is to show the use of a syntax exit.

```
 PBEGIN  EP=CHOPVAR_SYNTAX,ENTRY=NO
```

The PBEGIN macro has two new operands, because the procedure has more than one procedure body. Because the name of the procedure is not sufficient to specify the individual procedure bodies, the EP= operand is used to specify the name of the particular procedure body. ENTRY=NO specifies that no external entry is needed, because the syntax exit is specified in the program descriptor.

```
 PIPWORD ,                    Scan a word
```

When the syntax exit is entered, the general registers contains various pointers and numbers. In particular, general register 2 points to the beginning of the argument string and general register 3 contains the count of characters in this string. General register 9 contains the address of the pipeline services transfer vector.

The macro PIPWORD calls a scanning routine that separates the first blank-delimited word from the head of the string defined by the contents of registers 2 and 3. The address of the word is returned in general register 4 and the length of the word is returned in general register 5. Registers 2 and 3 are adjusted to describe the remaining argument string.

```
IF      ZERO                 Nothing there.
  LA    R0,15                Take a default
```

PIPWORD sets the return code to the number of characters in the word just scanned. Thus, if the return code is zero, the argument string contained only blanks and a default record length is loaded into register 0.

```
ELSE    ,
  PIPDECWD ,                 Convert to decimal
  PIPERM 58,EXIT,COND=NOTZERO Trouble converting?
  LTR   R0,R0                Must be positive
  PIPERM 66,EXIT,COND=NOTPOSITIVE
```

A blank-delimited word was found. Use PIPDECWD to convert it from decimal to binary. PIPDECWD requires the address of the input word to convert in register 4 and the length in general register 5. It is of course no coincidence that PIPWORD left those values there.

PIPDECWD returns the binary value of the word in general register 0. It sets the return code to zero if the word contains a signed number that can be represented in binary in 32 bits two's complement notation. It sets a non-zero return code otherwise.

If the return code from PIPDECWD is non-zero, message 58 is issued to diagnose this problem and the syntax exit terminates with a non-zero return code.

But not all valid signed numbers are useful as record lengths. It is a matter of taste whether one should accept zero as a valid record length or not; in this example, zero and negative record lengths are rejected with message 66.

```
ST      R0,CHOPVAR_LENGTH    Save for main procedure
```

Store the default value or the specified value into the work area.

```
PEXIT   RC=(R15),REGS=(R2,R3)
```

Return to the *CMS Pipelines* scanner. The unscanned argument string must be returned. REGS=(R2,R3) specifies that the range of registers from 2 to 3 should not be restored from the save area; instead they should be passed back unmodified. (Actually, the epilogue stores the contents of registers into the save area so that the modified values are returned, but that clearly amounts to the same thing.)

The scanner will issue an error message if the unscanned argument string is not blank; you need not worry about small details like this. But you must worry about returning the registers to the scanner.

```
 PBEGIN  EP=CHOPVAR_GO,ENTRY=NO,INITWA=NO
```

Because the procedure has two bodies, you must supply a unique name for each. This procedure body is also addressed from the program descriptor; no entry point need be generated. INITWA=NO is not important in this example, because there are no constants to be initialised in the work area. Still, it is a good habit to specify INITWA=NO.

```
&MODULE.CV PIPDESC FP=NO,STREAMS=1,STOPABLE=YES,                      *
               SYNTROUT=CHOPVAR_SYNTAX,                               *
               SYNTAX=(CALLSYNT,DONE)
```

Finally, the program descriptor has been modified to specify the location of the syntax exit (SYNTROUT=) and to specify that it should be called (SYNTAX=).

The SYNTAX= operand supports a contorted and rich set of operations, which are not documented formally. You will be well advised to stay clear of it, except for a few simple cases like the one shown here. CALLSYNT tells the scanner to go call your exit; DONE says that you are done scanning the argument string.

Example 4, Truncate at Specified Length, Differently

The previous example was included to highlight the use of a syntax exit. If you study the *CMS Pipelines* source code, you will see that its function could be accomplished by a syntax program. No doubt, you appreciate the compactness of the procedure in Figure 5.

```
CHOPVARS PROC  DCL=NO,SAREA=STACK,ENTRY=NO
 REPEAT  ,                    Begin main loop
   PIPLOCX ,                  Peek at the next record
   LRMIN R0,R8                Truncate if needed
   PIPOUTX ,                  Write output record
   PIPINPUT (,0)              Consume input record
 UNTIL   NOTZERO
 PEXIT   RC=(R15)
&MODULE.CS PIPDESC FP=NO,STREAMS=1,STOPABLE=YES,                      *
               SYNTAX=(WORD,                                          *
               ?Z,(:0,15),                                            *
               (DECWD,?NZ58,0TO15,?NP66),                             *
               =0,8,DONE)
 PROCEND ,
```

Figure 5. Scanning a Number in a Syntax Program

The procedure body is almost identical to the one in the first example. The SYNTAX= operand is the one needing scrutiny.

The syntax program is specified as suboperands of the SYNTAX= keyword (or as positional operands on a separate #SYNTAX macro instruction). The program above performs the same function as the exit program in the previous example:

- A call to the service routine PIPWORD. “PIP” is omitted when calling a pipeline service routine in the syntax program.
- The question mark operation tests the contents of general register 15 and then evaluates one of the two following sublists. The “Z” specifies to test for zero. Thus, the first sublist is evaluated if register 15 is zero (that is, there are no arguments); the second sublist is evaluated if the contents of register 15 are not zero (there is an argument).
- When there are no arguments:
  - The colon operation loads a general register. The number after the colon is the number of the register.
  - The constant to load is specified in the next operand. This can be any constant that can be stored in a four-byte address constant.
  - The program skips the next sublist.
- When there is an argument:
  - PIPDECWD is called to convert the number to binary.
  - Issue message 58 if the word could not be converted.
  - 0TO15 loads the contents of general register 0 into register 15 for testing.
  - Issue message 66 if the number is not positive.
- The equal sign operator stores the contents of a general register into the stage’s initial register set so that the value will be pre-loaded when the main procedure is called. The number of the register is specified after the equal sign.
- The number of the receiving register is specified in the next sub-operand. In this example, the argument is stored into general register 8.
- DONE specifies that the syntax program should terminate. The scanner ensures that only blanks remain in the unscanned argument string.

Example 5, Coerce at Specified Length

This example highlights the buffer handling features of *CMS Pipelines*. You need a buffer to handle records of arbitrary lengths. And the length *will* be arbitrary if you allow it to be specified as an argument. It may even be larger than the dreaded 16M. You need not worry; *CMS Pipelines* copes.

```
COERCEVAR PROC SAREA=STACK,ENTRY=NO
CVBUF    PIPBFR 512           A small buffer
 PBEGIN  ,
 PIPIBFR CVBUF                Ready buffer
 REPEAT  ,                    Begin main loop
   PIPLOCX ,                  Peek at the next record
   CR    R0,R8                What's the score?
   IF    LOW
     PIPBFRLD  CVBUF,((R1),(R0)),EXIT=NOTZERO
     SR        R5,R8          Figure how much to pad (negative         *
                              of)
     LPR       R5,R5          Make number of bytes
     LA        R4,256+C'*'    Get pad
     LNR       R4,R4          Indicate pad
     PIPBFRAP  CVBUF,EXIT=NOTZERO
     PIPBFRSU  CVBUF          Prepare to write
   FI    ,
   PIPOUTX (,(R8))            Write output record
   PIPINPUT (,0)              Consume input record
 UNTIL   NOTZERO
 PEXIT   RC=(R15)
&MODULE.FV PIPDESC FP=NO,STREAMS=1,STOPABLE=YES,                       *
               BUFFER=CVBUF,                                           *
               SYNTAX=(WORD,                                           *
               ?Z,(:0,15),                                             *
               (DECWD,?NZ58,0TO15,?NP66),                              *
               =0,8,DONE)
 PROCEND ,
```

Figure 6. Coercing to a Specified Length

```
CVBUF    PIPBFR 512           A small buffer
```

The PIPBFR macro defines a control structure to manage the contents of the buffer as well as an initial allocation. The label field specifies the name of the buffer; the operand field specifies the number of bytes to allocate initially. In this example, a 512-byte area is allocated after the buffer control block.

```
 PIPIBFR CVBUF                Ready buffer
```

The PIPIBFR macro initialises the addresses in the buffer control block. This cannot be done at assembly time, because the buffer is allocated in the procedure work area, which in turn is allocated in dynamic storage by the *CMS Pipelines* dispatcher.

```
     PIPBFRLD  CVBUF,((R1),(R0)),EXIT=NOTZERO
```

PIPBFRLD loads a string into the buffer starting at the beginning. The first operand specifies the buffer control block. The second operand specifies the address and length of the string to load (in this case the address and length of the record just peeked at). These are loaded into registers 4 and 5 before the service routine is called.

The service routine takes care of buffer overflow and out-of-storage conditions. If the string to be loaded cannot fit within the allocated buffer:

- A new and larger buffer is obtained from system storage. If there is no more free storage, a message is issued and a non-zero return code is set.
- The contents of the old buffer up to the current high-water mark are copied to the beginning of the new buffer. In this case the high-water mark is already set to the beginning of the buffer so that nothing needs to be copied.
- If the old buffer was obtained from CMS, it is returned to CMS. The initial allocation in the procedure work area is, of course, left intact.

The string is then copied into the buffer and the high-water mark is set to the byte after the end of the string.

```
         SR        R5,R8           Figure how much to pad (negative        *
                                   of)
         LPR       R5,R5           Make number of bytes
         LA        R4,256+C'*'     Get pad
         LNR       R4,R4           Indicate pad
         PIPBFRAP  CVBUF,EXIT=NOTZERO
```

Register 5 contains the length of the string loaded into the buffer and therefore also the number of bytes in the buffer. By subtracting the desired record length, the negative of the number of pad bytes is obtained. This number is made positive. The pad character is loaded into register 4. Adding 256 is idiomatic to ensure that the register always contains a positive number. This number is then made negative to indicate that the operation desired is padding.

PIPBFRAP normally appends a string described by registers 4 and 5 to the contents of the buffer, but when register 4 is negative, it contains a pad character, as computed here. In fact, PIPBFRLD expands to instructions to reset the buffer control block to be empty and then issue PIPBFRAP to load the string; thus, the operations described for PIPBFRLD above also apply to PIPBFRAP: The buffer is extended if required and out-of-storage conditions are diagnosed.

```
         PIPBFRSU  CVBUF           Prepare to write
```

PIPBFRSU sets the buffer up to be written into the pipeline. It loads the base of the buffer area into register 1 and the offset to the high-water mark (the number of bytes currently in the buffer) into register 0. It then resets the high-water mark to the buffer's base to be ready for the next cycle. (This is redundant, for so does PIPBFRLD.)

```
       PIPOUTX (,(R8))             Write output record
```

To ensure that the record is truncated, the length in register 8 is used for all writes, even when the record is padded in the buffer. The address of the record is already in register 1; the omitted suboperand specifies that it should be left unchanged.

```
&MODULE.FV PIPDESC FP=NO,STREAMS=1,STOPABLE=YES,                      *
               BUFFER=CVBUF,                                          *
               SYNTAX=(WORD,                                          *
               ?Z,(:0,15),                                            *
               (DECWD,?NZ58,0TO15,?NP66),                             *
               =0,8,DONE)
```

The program descriptor specifies the name of the buffer in the BUFFER= operand. When this operand is specified, the *CMS Pipelines* dispatcher takes care of releasing the buffer after the stage terminates. When STOPABLE=YES is specified, it is imperative that the buffers used are identified in the program descriptor; otherwise they might not be released.

Nested Procedures

For real filters, you will probably wish to divide the program into several procedures. A procedure can be nested within a containing procedure. A nested procedure is coded in the static area of the containing procedure. The work areas and save areas are allocated and managed automatically. You may wish to use WORKBASE=R11 on the PROC macro that opens the main procedure; this makes its work area addressable in the nested procedures.

Use PCALL (procedure call) or PCALLTR (procedure call and test return code) to call a nested procedure.

Because the work areas are allocated in a tree structure that mirrors the structure of the nested procedures, proper hierarchical calls are enforced.

Making Do with “Normal” Assembler

If you would like to restrict yourself to the set of VM/ESA macros that will be documented formally in Version 2 Release 1.0, you must use two more operands on the PIPDESC macro. Specify the first instruction of your prologue with the EP= keyword and specify the size in doublewords of the work area you require with WORKAREA keyword.

You are responsible for saving registers, establishing a base register, initialising your work area, restoring registers, and setting a return code.

These macros will be documented with VM/ESA 2.1.0: PIPCMD, PIPCOMMT, PIPDESC, PIPEPVR, PIPINPUT, PIPLOCAT, PIPOUTP, PIPSEL, PIPSEVER, PIPSHORT, PIPSTRNO, and PIPSTRST.

Using the Toolsmith’s Guide

The *CMS Pipelines* Toolsmith’s Guide documentes the original Assembler interface in *CMS Pipelines* level 1.1.0. The program descriptor was introduced in level 1.1.3. When using a program descriptor, these registers contain different values than when not using a program descriptor:

R1	Base for the work area.
R2	The address of the argument string.
R3	The length of the argument string.

R8	The stage's position in the pipeline. (Was in register 3.)
R9	The Pipeline Services Transfer Vector. (A convenience; it is also in general register 5.)

The PIPNUPL macro should not be used. The remaining macros still work.

Turning a Module into a Filter Package

To turn an Assembler program into a filter package that can be loaded seamlessly with *CMS Pipelines*, you must create an entry point table and generate the module.

You can construct the entry point table in two ways:

- Assemble it into your filter module. This is most handy if you only have one source file and no REXX programs.
- Generate it with the utility PIPGDIR. The input file lists the filter names and the labels on the program descriptors. You can merge entry point definitions for your Assembler filters with the ones for REXX programs.

Assembling the Entry Point Table

```
 ENTRY   PIPEPT
PIPEPT   DS    0D
 PIPEPT  ,
 PIPEPTEN CHOPVAR,FLTPKGCV
 PIPEPTEN CHOPVARS,FLTPKGCS
 PIPEPTEN CHOP20,FLTPKGC2
 PIPEPTEN COERCEVAR,FLTPKGFV
 PIPEPTEN COERCE20,FLTPKGF2
 PIPEPTED ,
```

The entry point table in a filter package must have the entry name PIPEPT. You must not specify operands on the PIPEPT macro, which defines the table prefix. The PIPEPTEN macros must be ordered in ascending order by the entry point verb.

Generating an Entry Point Table from a Source File

You can also store the entry point table as a text file and use the utility PIPGDIR to generate an object module that contains the table. The source file would look like this:

```
chop20     fltpkgc2      *** Chop after 20
coerce20   fltpkgf2      *** Chop after 20 or pad to 20
chopv      fltpkgcv      *** Chop to specified length
chopvx     fltpkgcs      *** Chop to specified length
coercev    fltpkgfv      *** Chop/pad to specified length
```

The input records may be in any order. If this file is called FLTPKG EPTABLE (the file type must be EPTABLE), you can generate the file PIPEPT TXTPARM by issuing the command PIPGDIR FLTPKG.

DMSPFP Glue Module

The object module DMSPFP contains the code that makes the filter package known to *CMS Pipelines*. It also contains code to install the filter package and attach it when it is invoked as a CMS command.

Due to the way CMS works, DMSPFP must be the first control section in a filter package. The EXEC in Figure 7 can be used to assemble the filter module, generate the entry point table, generate the module, and attach the filter package to *CMS Pipelines*:

```
/* Generate the sample filter package                                */
/*                               John Hartmann  14 Feb 1995 12:27:14 */
Signal on novalue
signal on error
Address COMMAND                  /* Real programmers address Command */

'GLOBAL MACLIB DMSGPI DMSOM'          /* Name of macro libraries     */

'HASM FLTPKG'                         /* Compile module              */

'EXEC PIPGDIR FLTPKG'                 /* Generate entry point table  */

'PIPE (name FLTPKG)',                 /* Make composite text deck    */
   '|literal dmspfp.text fltpkg.text pipept.txtparm',
   '|split',
   '|xlate . blank',
   '|getfiles',
   '|> $$TEMP TEXT A Fixed'

'LOAD $$TEMP ( CLEAR RLDSAVE'         /* Make module                 */
'GENMOD PIPUSERF ( FROM DMSPFP'

signal off error
'NUCEXT *PIPUSERF'                  /* Is package already installed? */
If RC=0
   Then 'NUCXDROP *PIPUSERF'          /* Drop if so                  */

'PIPMOD install'                      /* Load new package            */

error: Exit RC
```

Figure 7. Generating a Filter Package

Rather than coping with the quirks of loading several object modules one after the other, the EXEC generates a composite object module and loads a single file whence it generates the module.

Because the module file has one of the magic filter package names, it is installed by the main pipeline when the PIPMOD command is issued with the INSTALL operand.

When the module has a more pedestrian name, you must invoke the module as a CMS command to attach it to *CMS Pipelines*. This operation is called *filter package self-install*.

Adding REXX Filters to a Filter Package

To add some REXX filters to the filter package, you must first create a file that lists the file names of the individual REXX programs. The file type of this file should be REXXES.

If the file is called RXFLTPKG, you can issue this command to generate an object module:

```
pipgrexx rxfltpkg ( compress nodir
```

The options specify that the REXX filters should be one-lined and compressed as much as possible (COMPRESS); the object module is not to contain an entry point table for the REXX filters (NODIR).

These files are created: RXFLTPKG TXTPARM, which contains the object module, and REXXES EPTABLE, which contains the source entry point table for the REXX filters in the package. You would then use XEDIT to append the REXXES EPTABLE to the other entry point table, run this through the utility, and generate a module that contains the composite entry point table and both object modules.

How-tos

This section describes some finer points that you may find useful when you embark on your first filter package.

Copyright Notice.

To add your own DCs to set a copyright notice of your own:

```
MYMODULE MODBEG FREETYPE=NONE,CNOTE=NO
 PDATA   ,                    To main control section
        DC    C'COPYRIGHT 1887, ACME CO. KLONDIKE USA. '
        DC    C'ALL RIGHTS RESERVED'
 DMSPDEFS VECTOR=R9
```

The PDATA macro ensures that the following instructions are in a location counter that is beyond all other location counters used so far.

Literal Pools

The *CMS Pipelines* macros assume reasonably small procedures and also reasonably small modules. You must insert literal pools at strategic points in large programs.

The standard LTORG assembler instruction may not be effective in a procedure, because *CMS Pipelines* uses many location counters to do its magic and mirrors. Use PLTORG in a static data area to generate a literal pool:

```
 PEXIT   RC=(R15)
 PLTORG  ,
```

Constants at the End of the Module

To generate constants after the final literal pool:

```
 PROCEND ,
 PDATA   ,
        LTORG ,
BIGDATA  DS    0D
        DC    3765X'DEADBEAF'
 MODEND  ,
```

The PDATA macro ensures that the following code is in a location counter behind all currently defined location counters. The LTORG assembler instruction generates the final literal pool. The beginning of the constant will be addressable; there are no labels beyond the beginning of the large constant.

Scanning the Argument String

The examples at the beginning of this paper show how to scan for a decimal number. You can process much more complex arguments with *CMS Pipelines* services.

The scanning routines have adopted these register conventions:

R0	Integer return value. Single character in the loworder byte. For a range, the lower bound.
R1	Upper bound on a range of integers or characters. Address of an eight-byte token.
R2	Beginning of the remaining unscanned argument string.
R3	The length of the remaining unscanned argument string.
R4	Beginning of a word scanned off the argument string.
R5	The length of the word.

Some of the scanning routines are:

PIPSTRLB	Strip leading blanks from the argument string. This is useful to test for the presence of an operand following a keyword; the operand remains intact in registers 4 and 5, ready for substitution into an error message. Message 15 is popular for a missing keyword value.
PIPSTRTB	Strip trailing blanks from the argument string.
PIPBUWD	Back up to the last scanned word. This is useful to recover from speculative scanning that went wrong.
PIPMTKN	Load a word into an eight-character token, padded or truncated as necessary.
PIPUTKN	Load a word in uppercase into an eight-character token, padded or truncated as necessary.
PIPDRNG	Scan a word for a decimal range. This was originally intended to scan for column numbers. The defaults associated with asterisks are preloaded into registers 0 and 1, respectively.
PIPXCHR	Scan a word for being a single character or a two-character hexadecimal representation of a character.

PIPXRNG	Scan a character range.
PIPSDEL	Scan a delimited string from the arguments. This is normally preceded by PIPSTRLB to ensure that the delimiter is not blank. When called from a syntax exit, PIPSDEL does not handle hexadecimal or binary literals, but it does when used in a syntax program.

Message Numbers

The PIPERM macro is clearly a handy way to issue a message, but it uses the original message numbering scheme used by the *CMS Pipelines* Program Offering. It is not obvious which number to use unless you have a copy of the documentation for the original Program Offering. (*CMS Pipelines* User's Guide, SL26-0018-1.)

You can find the original message numbers and message texts in the module DMSPRM beginning at the label MSG0. The numeric part of the label is the original message number. You will not see the message text shown in DMSPRM if a CMS message has been assigned. The table at label DMSPRM@R relates the original message numbers to VM/ESA messages.

Other Dispatcher Services

The *CMS Pipelines* dispatcher supports many more services than the peek/write/consume functions shown earlier:

PIPSHORT	Connect the currently selected input and output streams directly, bypassing the stage. Use PIPSHORT when you wish to copy all remaining input unchanged to the output.
PIPSEL	Select a stream. Specify the side and the stream number or stream identifier.
PIPSTRNO	Return the stream number corresponding to a stream identifier.
PIPREXX	Run a REXX subroutine with access to the pipeline. Be sure to make registers 2 and 3 zero before issuing this macro, unless you wish to specify an in-storage program.
PIPCMD	Issue a pipeline command.
PIPSEVER	Sever a stream.
PIPSTRST	Return stream's status.

More Ways to Treat a Buffer

CMS Pipelines supports more macros to handle buffers than have been highlighted in the examples:

- The pre-allocated area need not be contiguous with the buffer control block; you can specify a remote location by the AREA= keyword.
- You can use the PIPBFROP macro to manipulate the contents of the buffer control block with RX-type and RS-type machine instructions:

  ```
  PIPBFROP <buffer>,<opcode>,<register>,<field>
  PIPBFROP CVBUF,L,R1,BASE
  ```

  The first positional operand specifies the label of the PIPBFR macro that defines the buffer control block. The remaining operands specify the operation to perform. The second positional operand specifies the operation code to generate; the third operand specifies the source or destination register; and the fourth operand specifies the symbolic name for the particular field you wish to reference.

LEN contains the number of bytes in the buffer; BASE contains the address of the first byte of the buffer; and NEXT contains the current high-water mark for the buffer. **Do not** change the base when the buffer has been extended; *CMS Pipelines* will issue an erroneous storage release request if this field is corrupted.

Determining Whether a Stage Is Stopable

Specifying STOPABLE=YES on a program descriptor allows the pipeline dispatcher to terminate the stage immediately it determines that all input streams are at end-of-file or that all output streams are severed. The dispatcher can even terminate a stage that is waiting for input when all its output streams are severed. This allows end-of-file to travel backwards faster than it would otherwise do; thus, you should strive to make your filters summarily stopable.

But beware that a stopable stage might not resume control after any call to the *CMS Pipelines* dispatcher. These are indications that you should not specify STOPABLE=YES:

- You wish to write a record after you receive end-of-file on your input; for example, a summary record.
- You have allocated resources that you must de-allocate before the stage can terminate.

In these situations, you can specify STOPABLE=RC8 to indicate that you are prepared to handle return code 8 on a read as meaning that all outputs have been severed. Clearly, if the outputs are severed, you need not spend much time computing a summary record, but you must still de-allocate whatever resources you have allocated.

```
PIPLOCAT EXIT=MINUS          Terminate if stall
IF       POSITIVE            EOF of some sorts?
  C      R15,=F'8'           Outputs gone
  GOTOEXIT 0,COND=EQUAL      Yes, terminate now
  PCALL WRITE_SUMMARY        Go do the summary
FI       ,
```

Use Commit Levels when You Allocate CMS Resources

You must be particularly vigilant when your program allocates a resource from CMS, for example a CPIC conversation.

Do **not** allocate any resources in a syntax exit. You will not be able to de-allocate them if some other syntax exit returns with an error.

Instead, allocate the resource on a negative commit level. Specify a negative commit level in the PIPDESC macro. COMMIT=-2000000000 is a popular value, but any negative number will do. Allocate the resource at the beginning of the program. You can signal any error in obtaining the resource by a non-zero return code at this low commit level, before other stages have taken any irreversible action.

When you have allocated the resource, commit to level zero. If the return code on the PIPCOMMT macro is non-zero, some other stage failed to commit; you should de-allocate the resource and terminate.

When using commits and allocating resources, it is customary to put the actual filter into a subroutine, which is only called when conditions are favourable:

```
 PBEGIN  ,
 PCALLTR ALLOCATE_RESOURCE    Try to get it
 PIPCOMMT 0                   Got the resource
 IF      NOTZERO              Abort
   LA    R15,0                Make return code zero
 ELSE    ,
   PCALL PROCESS_DATA         Go do real work
 FI      ,
 LR      R10,R15              Save return code
 PCALL   DEALLOCATE_RESOURCE
 LR      R15,R10              Restore return code
 PEXIT   RC=(R15)
&MODULE.FT PIPDESC COMMIT=-2000000000,                                *
               ...
```

Anatomy of the Syntax Abstract Machine

The program that you specify in the SYNTAX= operand of the PIPDESC macro is for an abstract machine with these characteristics:

- A location counter, which addresses the current instruction.
- General registers 0 through 5, and 15.
- Read-only storage for constants.
- Write-only storage, which is mapped to the first 1024 bytes of the stage's work area.
- A read/write register set, which contains the register contents that are loaded into the general registers when the stage is dispatched. This area contains nineteen registers; the first twelve map to the stage's general registers 0 through 11.
- The ability to conditionally and unconditionally issue messages and terminate.
- The ability to branch conditionally and unconditionally.
- The ability to call one level of subroutine.

An Entry Point Can Resolve to More than Program Descriptors

When the *CMS Pipelines* scanner resolves an entry point, it scans its entry point tables for the specified name. If it finds the name in a table, it gets the address of the entry point from the table.

The scanner then inspects storage at the location resolved so far. If the byte does not contain X'00', it is assumed to be an executable instruction. The scanner looks to see if it represents a compiled REXX program, but apart from that it looks no further.

When the first byte of the entry point contains X'00', it is clearly not an executable instruction. The scanner then looks into the next three bytes for a code to describe what follows.

pip—Program Descriptor

The PIPDESC macro expands to begin with the magic constant `A(C'pip')`. You have already seen how a program descriptor is used.

rex—An In-storage REXX Program

This is the format generated by the PIPGREXX utility. The next fullword contains the length, in bytes, of the descriptor list, which follows. The descriptor list contains eight bytes for each line of the REXX program, the address and the length.

cmd—A Pipeline Command To Be Issued

There is no utility to generate this format. The following fullword contains the length of the pipeline command, which follows. This is *getfiles*:

```
 ENTRY   &MODULE.GF                                            GETF
         DS    0D              Align                           GETF
&MODULE.GF DC  A(C'cmd',GF_LENGTH)                             GETF
         DC    C'callpipe (name getfiles) *:'                  GETF
         DC    C'|change 1.7 / &&1 &&2 //'                     GETF
         DC    C'|strip'                                       GETF
         DC    C'|locate 1'   Discard blanks and null          GFDELBLK
         DC    C'|change /"/""/|change /+/"+/'                 NOTPLUS
         DC    C'|spec ,callpipe '                             NOTPLUS
         DC    C'(name GETfile stagesep + escape ")'           NOTPLUS
         DC    C'<,1 w1.3 nextword ,+'                         GETF
         DC    C'unpack+'                                      GETF
         DC    C'*:, next'                                     GETF
         DC    C'|pipcmd'                                      GETF
         DC    C'|*:'                                          GETF
GF_LENGTH EQU  *-&MODULE.GF-8                                  GETF
```

ept—A Sub-keyword Table

A sub-keyword table causes the scanner to scan one more word from the stage's argument string and look this word up in the entry point table, which is headed by this magic constant.

Such an entry point table is generated by PIPEPT when a positional operand is specified.

This example is from a filter package that contains both a primary and a secondary entry point table:

```
 ENTRY   PIPEPT              Define entry point table
PIPEPT   DS    0D
 PIPEPT  ,
 PIPEPTEN PRTASM,GOPRT
 PIPEPTEN PRTASMF,EPTABLE
 PIPEPTED ,                  End of table
                                                 SPACE 1
 PIPEPT  EPTABLE,ENTRY=NO
 PIPEPTEN ADDHEAD,EPDOHEAD   Add headings and page breaks
 PIPEPTEN CLSECT,EPCLSECT    Classify sections
 PIPEPTEN CONDLIT,EPCOND     Conditional literal
 PIPEPTEN DIAGDIR,EPDIAG     Diagnostic
 PIPEPTEN GENCODE,EPGEN      Generated code to test
 PIPEPTEN PXREF,EPPXREF      Procedure cross ref
 PIPEPTEN TESTTITL,EPTITLE   Remove unwanted ejects
 PIPEPTEN XREF,EPXREF        Cross reference
 PIPEPTED ,                  End of table
```

lup—A Look-up Routine

The lookup routine is capable of looking an entry point up in some CMS table. For example, the *ldrtbls* program is a look-up routine. Using a look-up routine rather than a program descriptor allows the resolution process to continue after the CMS object has been located; it could in turn be a program descriptor.

The scanner scans the next word and calls the look-up routine to find the entry point.

rou—A Router

A router inspects the remainder of the argument string and selects an appropriate program descriptor to process the particular request. This allows for “overloading” of the built-in programs. The router function was introduced in *CMS Pipelines* level 1.1.9 to support the selection of a device driver that is appropriate to the particular request. For example, in level 1.1.9, the device driver < can cope with:

- A one-word path name or a path name enclosed in parentheses, which identifies an OpenEdition file.
- A two-word argument string or a longer argument string where the third word indicates a mode letter and optionally a number, which is processed by the original minidisk device driver.
- Other arguments strings, which indicate that the file is referenced through a name definition or an SFS directory. The device driver to read SFS files is then used.

Coping with Older Level of CMS

This paper was written and the samples were run using an early version of the macro library that will be shipped with VM/ESA 2.1.0. The sections below highlight the considerations for mixing and matching filter packages and *CMS Pipelines* levels.

Filter Package Transportability

In most cases, a filter package generated on any CMS release will work with any other version of *CMS Pipelines*.

The exception is specifying multiple buffers in the BUFFER= operand of FPEP/PIPDESC. This support was added in *CMS Pipelines* level 1.1.7, which shipped in ESA 1.2.1. An ABEND is likely if a filter that specifies multiple suboperands on BUFFER= is run on VM/ESA 1.2.0, VM/ESA 1.1.1, or on level 1.1.6 of the Program Offering/PRPQ.

Source Program Portability

If you wish to be able to assemble the module on all current CMS releases, you need to be more careful, because some macros have changed names even though the code they generate is unchanged. (Some of the reasons for this are “non-technical”.)

The easiest way to cope with assembling on older levels of CMS is to provide the 2.1.0 level of the relevant macros, which in the examples shown in this paper amounts to DMSPIPID, PIPDESC, FPEP, FPI, DMSPDEFS, DEFCHARS, #CHAREQU, and FPL#VER.

The sections below are structured as *antinews*; that is, they explain how to cope with function that has been removed when one adopts the standpoint of a subsequent release.

CMS 11 (1.2.2)

Use FPEP rather than PIPDESC. You must also code the flags as suboperands to FLAGS= rather than as individual keyword operands. The encoding of these flags is easily determined by inspection of the definition of the flags in the FPI macro, which is the mapping macro for the program descriptor.

CMS 10 (1.2.1) and CMS 7 (1.1.5)

No function was removed that is relevant to the examples in this paper.

CMS 9 (1.2.0)

DMSPIPID has been renamed to PGMID.

No sublist is supported on the BUFFER= operand of FPEP.

CMS 8 (1.1.1)

No *CMS Pipelines* function was removed in the transition from VM/ESA 1.2.0 to VM/ESA 1.1.1.

The pipeline shipped in CMS9 is identical to the one shipped in CMS8, which in turn is identical to level 1.1.6 of the *CMS Pipelines* Program Offering.

Conclusion

CMS Pipelines provides a well-honed set of macros to interface to its service routines. This paper has shown five examples of not quite trivial Assembler filters and provided much background information, which will be useful for the Assembler programmer who wishes to explore writing Assembler filters or wishes to explore the *CMS Pipelines* source code.

Bibliography

- *CMS Pipelines Toolsmith's Guide*, IBM Corp 1987, form number SB11-6605 or SL26-0020.
- *VM/Enterprise Systems CMS Pipelines User's Guide*, IBM Corp 1994, form number SC24-5609.
- John P. Hartmann, *CMS Pipelines Explained*, Proceedings of SHARE 78, Anaheim, CA, March, 1992, pp. 590-607.
- John P. Hartmann, *Using High-Level Language Concepts with Assembler Programs*, Proceedings of SHARE 74, Anaheim, CA, March, 1990, pp. 983-989.

Appendix. Issuing a PIPE Command from an Assembler Program

Though it is in some sense the opposite of the stated objectives for this paper, it may still be illustrative to see how one invokes a pipeline from an assembler program. You can use the *storage* device driver to write data from your program into the pipeline and to take them out again.

To obtain the pipeline level:

```
FIGURE_LEVEL PROC SAREA=NO
 LA      R0,FWD               Address to store Pipes level.
 PCALL   MAKEHEX              Convert to hex in DWD field.
 MVC     QADDR,DWD            Store as "storage" address parm.
 CMSCALL CALLTYP=EPLIST,PLIST=PL,EPLIST=EQL Level into FWD.
 ...
 PEXIT   RC=(R15)
                                                       SPACE 1
MAKEHEX  PROC  SAVE=NO,SAREA=NO,BASE=NO
 ST      R0,FWD               Make fullword
 UNPK    DWD(9),FWD(5)        Unpack
 NC      DWD,=8X'0F'          Strip
 TR      DWD,=C'0123456789abcdef'
 PROCEND RC=
                                                       SPACE 1
FWD      DS    F
DWD      DS    D
         DS    D              slop
PL       DC    CL8'PIPE',2F'-1'
EQL      DC    A(QPIPE,QARGB,QARGE,0)
QPIPE    DC    C'pipe '       Determine Pipelines level:
QARGB    DC    C'(listerr) '
         DC    C'query level |' Issue query.
         DC    C'storage '    Store response in FWD field.
QADDR    DS    CL8            Hex address of FWD field.
         DC    C' 4 e0'       Length and storage key.
QARGE    DS    0C
 PROCEND FIGURE_LEVEL
```