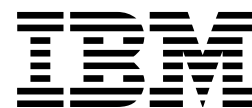




IBM BatchPipes OS/390 V2R1 BatchPipeWorks Users Guide



IBM BatchPipes OS/390 V2R1 BatchPipeWorks Users Guide

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

First Edition, April 2000

This edition applies to Version 2 Release 1 of IBM BatchPipes for OS/390 (5655-D45) and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 914+432-9405

FAX (Other Countries):

Your International Access Code +1+914+432-9405

IBMLink (United States customers only): KGNVMC(MHVRCFS)

IBM Mail Exchange: USIB6TC9 at IBMMAIL

Internet: mhvrcfs@vnet.ibm.com

World Wide Web: <http://www.s390.ibm.com/os390>

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

© Copyright International Business Machines Corporation 1992, 2000. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	vii
Programming Interface Information	vii
Trademarks	viii
About This Book	ix
Who Should Use This Book	ix
How to Use This Book	ix
Where to Find More Information	x
Chapter 1. Introduction to BatchPipeWorks	1
What is BatchPipeWorks?	1
When to Use BatchPipeWorks	2
Benefits of Using BatchPipeWorks with BatchPipes	2
Chapter 2. BatchPipeWorks Concepts	3
Specifying a Simple Pipeline	3
Stages and Stage Commands	4
A Simple Pipeline	5
Using Stage Separators	6
Specifying a Simple Pipeline	6
Device Driver Stage Commands	7
Filter Stage Commands	8
Multistream Pipelines	9
Examples of Multistream Pipelines	9
Specifying Multistream Pipelines	11
Defining Multistream Pipelines through Labels	12
Defining Multistream Pipelines that Have Secondary Output Streams	13
Defining Multistream Pipelines that Have Secondary Input Streams	16
Defining Multistream Pipelines with Secondary Input and Output Streams	17
Using Several Secondary Streams	17
Global Options on the Pipeline Specification	19
Storage Considerations with BatchPipeWorks	20
Avoiding Pipeline Stalls	20
Chapter 3. Using BatchPipeWorks with BatchPipes	23
BatchPipeWorks Terminology	26
Using a BatchPipeWorks Fitting	27
Making the JCL Changes for a Fitting	29
Adding the FIT Subparameter to the DD Statement	29
Adding the FITDD Subparameter to the DD Statement	30
Using BPREAD, BPWRITE, and BPCOPY in Fittings	31
Considerations for Fittings	32
Examples of Typical BatchPipes Fittings	34
Making Multiple Copies of Data	34
Changing Generation Data Groups (GDGs)	35
Counting Records	35
Using Fittings Outside the BatchPipes Pipeline	36
Example of a Half-Pipe Fitting that Routes Records	36
Example of a Half-Pipe Fitting that Select Records	38

Chapter 4. Filter Stage Commands	39
Selecting Records by Content	39
Looking Everywhere in the Record (LOCATE, NLOCATE)	39
Looking at the Beginning of a Record (FIND, NFIND, TOLABEL, FRLABEL)	43
Looking at the End of a Record	46
Discarding Duplicate Records (UNIQUE)	46
Discarding Unique Records (UNIQUE MULTIPLE)	47
Selecting Records by Position (TAKE, DROP)	48
Changing Contents of Records	50
Translating Characters (XLATE)	51
Splitting and Joining (SPLIT, JOIN)	53
Padding and Chopping (PAD, CHOP)	56
Removing Leading or Trailing Characters (STRIP)	57
Changing and Rearranging Contents (CHANGE, SPECS)	58
Duplicating Records (DUPLICATE)	73
Counting Characters, Words, and Records (COUNT)	73
Sorting Records (SORT)	74
Buffering Records (BUFFER)	77
 Chapter 5. Device Driver Stage Commands	 79
Sending Data to and from a Terminal (TERMINAL)	79
Writing Literal Character Strings to a Pipeline (LITERAL)	80
Reading from and Writing to Data Sets	81
The < Stage Command	82
The > Stage Command	82
The >> Stage Command	83
The QSAM Stage Command	84
Device Drivers Used with BatchPipes (BPREAD, BPWRITE, and BPCOPY)	84
The BPREAD Stage Command	84
The BPWRITE Stage Command	85
The BPCOPY Stage Command	85
Determining if Data Sets Exist (STATE)	85
Packing and Unpacking Data Sets	86
Reading and Writing REXX Exec Variables	87
STEM Stage Command	87
VAR Stage Command	89
Combining Records from Other Device Drivers	89
APPEND Stage Command	89
PREFACE Stage Command	91
 Chapter 6. Using Stage Commands for Multistream Pipelines	 93
FANOUT Stage Command	93
FANINANY Stage Command	96
Identifying Streams	97
FANIN Stage Command	98
OVERLAY Stage Command	100
SPECS Stage Command	101
COUNT Stage Command	103
MERGE Stage Command	105
LOOKUP Stage Command	107
Fixing Pipeline Stalls	110
Maintaining the Relative Order of Records	112
How Stages of a Pipeline Run	112
How Stages Delay the Records	113

How to Predict Relative Record Order of Records	113
Chapter 7. Using BatchPipeWorks in Batch Applications	121
Running BatchPipeWorks in a Batch Job	121
Placing a Pipeline Specification on an EXEC Statement	121
Placing a Pipeline Specification on a DD Statement	123
Using BatchPipeWorks in Filter Jobs	124
Example of Selecting Records	125
Selecting Records Based on Record Content	126
Reformatting or Combining the Contents of Data Records	127
Changing the Data Flow within a Job	128
Using User-Written Stage Commands in Batch Applications	129
Using a User-written Stage Command in a BatchPipes Filter Job	130
Chapter 8. Using BatchPipeWorks in TSO/E	131
Using BatchPipeWorks in REXX Execs	132
Saving a BatchPipeWorks Pipeline for Reuse	132
Continuing BatchPipeWorks on Several Exec Lines	132
Using BatchPipeWorks As Part of an Exec	133
Return Codes in BatchPipeWorks	135
Using BatchPipeWorks to run TSO Commands	136
TSO Stage Command	136
COMMAND Stage Command	137
Putting MVS Command Results in REXX Variables	137
Executing Pipeline Records as Commands	137
Including BatchPipeWorks in TSO Batch Jobs	138
Chapter 9. Writing Your Own Stage Commands	139
User-Written Stage Command Concepts	139
How BatchPipeWorks Processes a Pipeline Specification	141
How a Pipeline Runs	141
How a Pipeline Ends	142
An Example of a User-Written Command — HOLD	142
Using Your Stage Command	143
Pipeline Subcommands	144
READTO Subcommand	144
OUTPUT Subcommand	144
PEEKTO Subcommand	145
SHORT Subcommand	145
STAGENUM Subcommand	147
SELECT Subcommand	149
MAXSTREAM Subcommand	150
STREAMNUM Subcommand	152
SEVER Subcommand	153
CALLPIPE Subcommand	153
ADDDPIPE Subcommand	154
Processing Arguments	163
An Example of a Stage Command — TITLE	164
Using CALLPIPE to Write Subroutine Pipelines	165
Storing Sequences of Stage Commands	167
Other Formats of Connectors	169
Using Connectors with CALLPIPE	169
Using CALLPIPE with Other Subcommands	171
Additional CALLPIPE Examples	175

Testing Stage Commands	176
Tracing Stage Commands	177
Chapter 10. Blocking and Deblocking	179
Fixed Format	180
Variable Format	182
Line-End Character Format	183
Packed Format (PACK, UNPACK)	184
Creating Fixed-Format Records with FBLOCK	186
Chapter 11. Debugging BatchPipeWorks	189
Tracing BatchPipeWorks	189
Tracing to a Data Set	191
Tracing Individual Stages	195
Controlling Trace Messages	196
Taking Snapshots of Data	196
Naming BatchPipeWorks (NAME Option)	197
Displaying All Nonzero Return Codes (LISTERR Option)	198
Appendix A. Stage Commands and Pipeline Subcommands	201
Stage Commands that are Device Drivers	201
Stage Commands that Change the Contents of Records	202
Stage Commands that Select Records	203
Stage Commands that Support Multiple Streams	204
Stage Commands that Block and Deblock Records	205
Stage Commands that Work With REXX Execs	205
Stage Commands that are Event-Driven	205
Miscellaneous Stage Commands	206
Pipeline Subcommands	207
Appendix B. Additional Examples	209
Displaying Block Comments	209
Adding Sequence Numbers to a Data Set	209
Reversing the Order of Records	210
Isolating Words	212
Writing the First Comment Lines of Data Sets	212
Marking Selected Lines	214
Reversing the First and Last Records	215
Appendix C. BatchPipeWorks Messages	217
Glossary of BatchPipeWorks Terms	265
Index	269

Notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594
USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Any pointers in this publication to non-IBM Web sites are provided for convenience only, and do not in any manner serve as an endorsement of these Web sites.

Programming Interface Information

This publication is intended to help system programmers and application developers work with IBM BatchPipes/MVS BatchPipeWorks.

This book also documents General-use Programming Interface and Associated Guidance Information and Diagnosis, Modification or Tuning Information provided by IBM BatchPipes/MVS BatchPipeWorks.

General-use programming interfaces allow the customer to write programs that obtain the services of IBM BatchPipes/MVS BatchPipeWorks.

General-use Programming Interface and Associated Guidance Information is identified where it occurs, by an introductory statement to a chapter or section.

Diagnosis, Modification or Tuning Information is provided to help the customer diagnose problems with IBM BatchPipes/MVS BatchPipeWorks.

Warning: Do not use this Diagnosis, Modification or Tuning Information as a programming interface.

Diagnosis, Modification or Tuning Information is identified where it occurs, either by an introductory statement to a chapter or section.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

- BatchPipes
- BatchPipeplex
- BatchPipeWorks
- BookManager
- DB2
- DFSMS
- DFSMS/MVS
- IBM
- IBMLink
- MVS
- MVS/ESA
- OS/390
- Parallel Sysplex
- QMF
- RACF
- Sysplex Timer
- S/390

About This Book

This book describes IBM BatchPipes/MVS BatchPipeWorks and offers guidance on how to use it with BatchPipes jobs and in other environments.

Who Should Use This Book

The book is intended for system programmers and application developers who work with BatchPipes. These readers should be familiar with BatchPipes and with the batch jobs that are using BatchPipes. Additionally, others your installation might find the BatchPipeWorks function helpful in the non-BatchPipes batch environment and the TSO environment.

How to Use This Book

The book describes how to use BatchPipeWorks with:

- Jobs that are part of a BatchPipes pipeline
- Batch jobs that are not part of a BatchPipes pipeline
- TSO, both as part of a REXX exec and as a command on the TSO command line.

The book is organized as follows:

- Chapter 1, “Introduction to BatchPipeWorks” on page 1 introduces how you use BatchPipeWorks in different environments.
- Chapter 2, “BatchPipeWorks Concepts” on page 3 introduces the BatchPipeWorks concepts that apply in all environments.
- Chapter 3, “Using BatchPipeWorks with BatchPipes” on page 23 describes how to use BatchPipeWorks with BatchPipes, describing the JCL parameters that support BatchPipeWorks and introducing some important considerations.
- Chapter 4, “Filter Stage Commands” on page 39 introduces filter stage commands and shows how to use them.
- Chapter 5, “Device Driver Stage Commands” on page 79 introduces the stage commands that interface with data sources or targets outside the BatchPipeWorks pipeline and shows how to use them.
- Chapter 6, “Using Stage Commands for Multistream Pipelines” on page 93 builds on multistream concepts introduced in the second chapter and offers many examples.
- Chapter 7, “Using BatchPipeWorks in Batch Applications” on page 121 describes how to use BatchPipeWorks with batch applications that are not defined to BatchPipes.
- Chapter 8, “Using BatchPipeWorks in TSO/E” on page 131 describes several ways to use BatchPipeWorks in the background and foreground in the TSO environment.
- Chapter 9, “Writing Your Own Stage Commands” on page 139 describes how you can use BatchPipeWorks subcommands to write your own BatchPipeWorks stage commands.

- Chapter 10, “Blocking and Deblocking” on page 179 describes the stage commands that you can use to change the blocking of records.
- Chapter 11, “Debugging BatchPipeWorks” on page 189 offers help for those who diagnose errors.

Three Appendixes follow the chapters:

- Appendix A, “Stage Commands and Pipeline Subcommands” on page 201
- Appendix B, “Additional Examples” on page 209
- Appendix C, “BatchPipeWorks Messages” on page 217.

A glossary of BatchPipeWorks terms follow the Appendixes.

Where to Find More Information

Where necessary, this book references information in other books, using shortened versions of the book title. The following tables show the complete titles and the order numbers of books you might need while you are using this book:

Short Title Used in This Book	Title	Order Number
<i>BatchPipes Introduction</i>	<i>IBM BatchPipes OS/390 V2R1 Introduction</i>	GA22-7459
<i>BatchPipes Users Guide and Reference</i>	<i>IBM BatchPipes OS/390 V2R1 Users Guide and Reference</i>	SA22-7458
<i>BatchPipeWorks Reference</i>	<i>IBM BatchPipes OS/390 V1R2 BatchPipeWorks Reference</i>	SA22-7456
<i>BatchPipeWorks Users Guide</i>	<i>IBM BatchPipes OS/390 V1R2 BatchPipeWorks User Guide</i>	SA22-7457

Chapter 1. Introduction to BatchPipeWorks

This chapter contains General-Use Programming Interface and Associated Guidance Information.

This chapter introduces BatchPipeWorks — for the developers who support BatchPipes batch applications and also for those who support non-BatchPipes applications and for TSO users. It suggests uses of BatchPipeWorks in those environments and refers you to sections in this book where you can find more information.

What is BatchPipeWorks?

BatchPipeWorks consists of over 100 commands, called **stage commands**, that you can use to change the way data flows. Using stage commands, you can:

- Select records based on certain criteria
- Reformat, combine, or change the contents of records
- Change the data flow within a job
- Make multiple copies of records
- Create reusable REXX programs that alter the contents of records
- Retrieve data from or write data to various locations such as data sets, REXX variables, or the terminal.

You can combine the stage commands to extend the processing of an existing application, or the combination of stage commands can themselves be an application. If the stage commands that BatchPipeWorks provides are not sufficient for your needs, you can write your own stage commands; see Chapter 9, “Writing Your Own Stage Commands” on page 139.

You can use BatchPipeWorks in the following environments:

- With BatchPipes jobs:

You can add function to the BatchPipes pipeline, either between the BatchPipes writer and the pipe, or between the pipe and the BatchPipes reader. Chapter 3, “Using BatchPipeWorks with BatchPipes” on page 23 introduces BatchPipeWorks to the BatchPipes user. Read it to understand how BatchPipeWorks fits into your BatchPipes pipelines.

- As a batch application

You can use stage commands to write a BatchPipes program, rather than using COBOL or some other language. This use of BatchPipeWorks is described in Chapter 7, “Using BatchPipeWorks in Batch Applications” on page 121.

- In TSO

You can use the stage commands and operands on the PIPE command from the TSO command line. Or, you can place the stage commands and operands into a REXX exec and run the exec in TSO. Chapter 8, “Using

BatchPipeWorks in TSO/E” on page 131 describes this use of BatchPipeWorks.

Regardless of how you use BatchPipeWorks, read the concepts described in Chapter 2, “BatchPipeWorks Concepts” on page 3 to get started. Then, turn to the chapters that apply to the specific environments. To see the wide variety of function performed by the stage commands, see Appendix A, “Stage Commands and Pipeline Subcommands” on page 201.

When to Use BatchPipeWorks

Once you understand the variety of function in BatchPipeWorks, you will find many uses for it. To begin with, look for application that need to:

- Combine data from more than one data set
- Reformat records
- Select records
- Write data sets to other data sets or jobs
- Count records.

Benefits of Using BatchPipeWorks with BatchPipes

In a batch environment, there is often a need for additional function in the form of programs that manipulate records or change the contents of records. BatchPipeWorks gives you this capability. With BatchPipeWorks:

- You can change what the applications do, without having to change the application code. BatchPipeWorks increases your development and prototype capability, as well as your ability to respond to single-instance use of function.
- Because BatchPipeWorks is added to the JCL, rather than to the application code, you do not need to recompile, assemble, and linkedit the changed code.
- You can build and reuse code that hardens data defined as generation data groups (GDGs).
- Scheduling of jobs is simplified because the BatchPipeWorks commands run as part of a job, rather than being a separate job.
- Fittings that select and exclude records can reduce the number of records that jobs process.
- Under certain circumstances, BatchPipeWorks can help you avoid using an initiator, lower the use of processor cycles and processor storage, and reduce the amount of swapping.

BatchPipeWorks offers a simple approach to changing the data flow through the BatchPipe pipeline.

Chapter 2. BatchPipeWorks Concepts

This chapter contains General-Use Programming Interface and Associated Guidance Information.

Regardless of how you use BatchPipeWorks, you need to understand the same concepts. This chapter describes them, beginning with the idea that data flows through a pipeline in a series of operations called stages. A **stage** consists of the processing of a single **stage command**. BatchPipeWorks provides over 100 stage commands; some of them read data from or write data to system sources or targets, such as a BatchPipes pipe, a sequential data set, or a partitioned data set. Others change and refine that data in some way. You can combine many stage commands into a single **pipeline** to create the results you need. To grasp the wide variety of actions the stage commands perform, see Appendix A, “Stage Commands and Pipeline Subcommands” on page 201.

BatchPipeWorks allows you to create your own stage commands when the BatchPipeWorks stage commands are not sufficient for your needs. User-written stage commands are REXX programs that read data from the pipeline, work on the data, and place the data back in the pipeline. You can use user-written stage commands and BatchPipeWorks stage commands in the same pipeline. See Chapter 9, “Writing Your Own Stage Commands” on page 139 for information on user-written stage commands.

Specifying a Simple Pipeline

With BatchPipeWorks, data enters the pipeline from a data source, flows through the pipeline, and exits to a target. In Figure 1, the data moves from a data set, through the pipeline, to another data set.

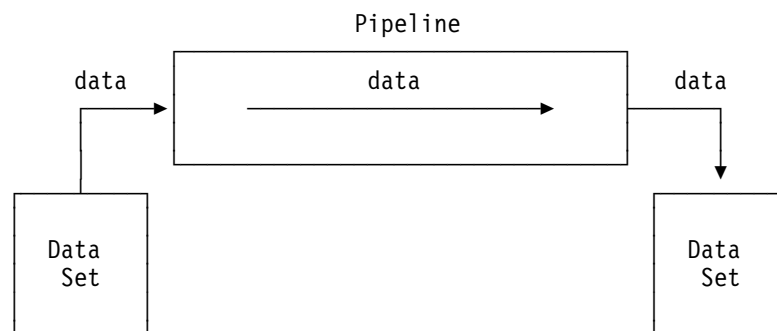


Figure 1. Data Flowing through a Pipeline

The pipeline consists of stages, where each stage affects the data that flows through the pipeline or changes the way the data flows.

In Figure 2, the pipeline consists of three stages.

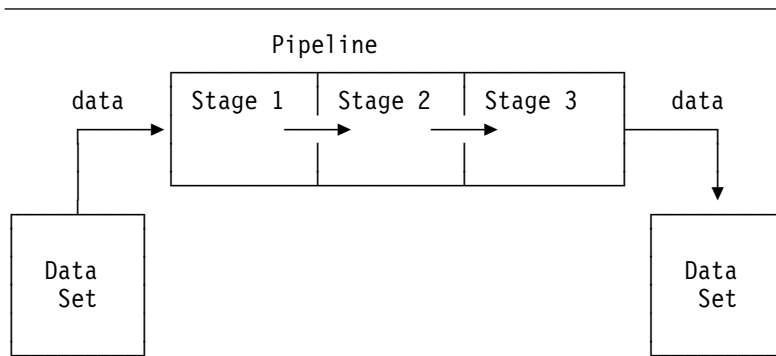


Figure 2. Stages within a Pipeline

Stages in a pipeline run one at a time. Generally they run sequentially; in other words, a record passes through Stage 1, then through Stage 2, then through Stage 3, before another record enters Stage 1.

Stages and Stage Commands

In a pipeline, output of one stage is input to the next, where the output and the input is in the form of discrete records. Figure 3 shows input records on the left. The stage command reads the records, processes them, and produces output records. Output records appear on the right.

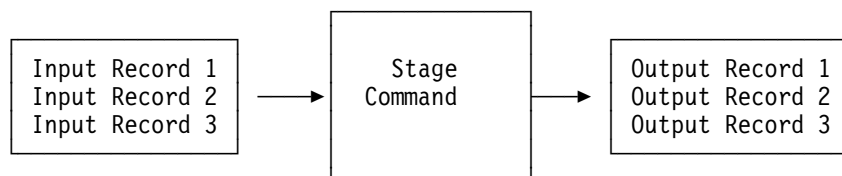


Figure 3. Records Flowing through a Stage

While in the stage, the records change, depending on the stage command used. For example, they can be modified, discarded, joined, or split. Many stage commands write one output record for each input record, as examples of CHOP and COUNT stage commands in Figure 4 and Figure 5 show.

Records entering a stage are called an **input stream**; records leaving a stage are called an **output stream**. When a stage command reads input from more than one source or writes records to more than one target, multiple data streams are involved. When a pipeline contains at least one stage command with multiple streams, the pipeline is a **multistream pipeline**. This topic is described in "Multistream Pipelines" on page 9.

Example of Using the CHOP Stage Command

In Figure 4, the CHOP stage command truncates records at a specified length of 5 characters. Like many stage commands, CHOP writes one output record for every input record.

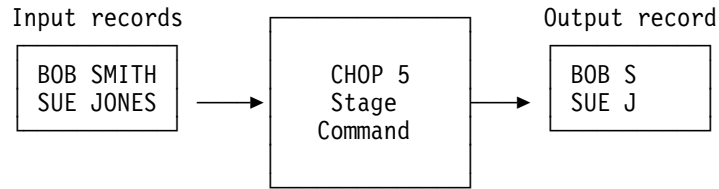


Figure 4. Records Flowing through a CHOP Stage Command

Example of Using the COUNT Stage Command

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

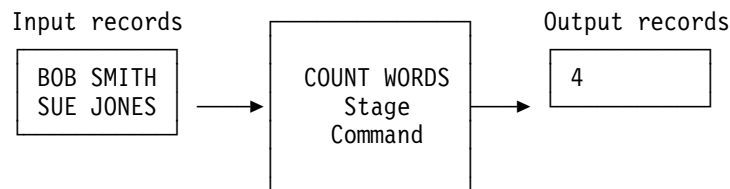


Figure 5. Records Flowing through a COUNT Stage Command

A Simple Pipeline

A simple pipeline consists of input streams, stage commands, and output streams. Figure 6 shows how records flow through two stages that consist of the LOCATE and COUNT stage commands.



Figure 6. Records Flowing through Multiple Stages

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

Using Stage Separators

When you specify a pipeline, you separate one stage command from another with a **stage separator**, as in the following pipeline:

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

Do not place a stage separator after the last stage.

For the default stage separator, BatchPipeWorks expects the character X'4F'. You must determine which key on your terminal generates the character X'4F'. It is a solid vertical bar (|) on American and English 3270 terminals. In some countries, this character is displayed as an exclamation mark (!). Some workstation terminal emulator programs map the solid vertical bar to the split vertical bar (|). The solid vertical bar is the logical-or operator in PL/I and REXX programs. In a pipeline, it indicates where one stage ends and another one begins. If you aren't sure what character to use, create and run the exec in Figure 7.

```
/* **** */
/* Finding the default stage separator */
/* **** */
say 'The stage separator is:' '4f'x'.
exit
```

Figure 7. Finding the Stage Separator

For more information about how to define the stage separator, see “Global Options on the Pipeline Specification” on page 19.

Specifying a Simple Pipeline

Specifying a simple pipeline means specifying stage commands, separated by stage separators. This book uses the term **pipeline specification** to describe the combination of stage commands and stage separators that tell BatchPipeWorks what it is to do.

Consider the pipeline specification that consists of the following three stage commands:

```
< 'data.profile'
    Reads the DATA.PROFILE data set and writes records to its output
    stream. Note that, in pipeline specifications in a batch environment, you
    place single quotes around the data set name.

count records
    Counts the records it reads from its input stream. COUNT with the
    RECORDS operand tallies the count of records in your DATA.PROFILE
    data set and writes a single record to its output stream.

joblog
    Reads the tally from its input stream and writes the single number to its
    joblog. (The output of this pipeline specification appears in the joblog if
    you run it within a batch job. If you run it in TSO, the output shows up
    on the terminal.)
```

Figure 8 shows the data flow through this three-stage pipeline.

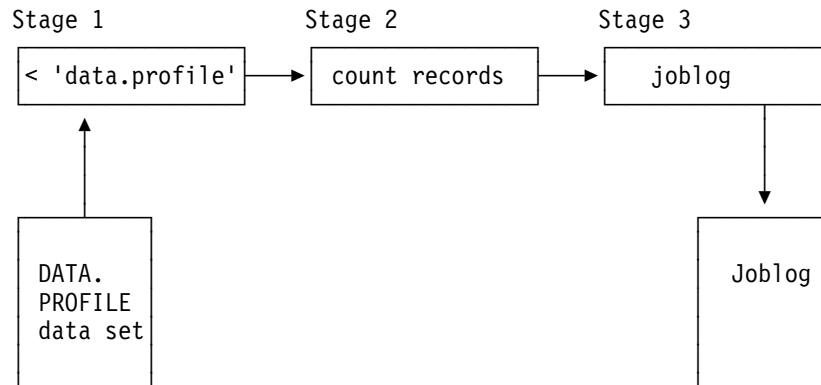


Figure 8. Diagram of a Pipeline

The specification for this three-stage pipeline would be as follows:

```
< 'data.profile' | count records | joblog
```

Notes:

1. The examples in this chapter do not include the required definitions of continuation lines and line comments. Both of those definitions depend on whether you are using BatchPipeWorks with BatchPipes, with MVS batch jobs, or in TSO. In other words, these examples will not work in isolation.
2. If you make a mistake typing the command, an error message displays.
3. 'DATA.PROFILE' is the fully-qualified name of this data set; it is enclosed in single quotations and with a space after <, as in the following example:

```
< 'prefix.data.profile' | count records | joblog
```

If you are in TSO and you do not use the single quotations, BatchPipeWorks adds your TSO userid to the data set name.

4. It is easy to request another device by changing one stage command. For example, to change the above example to write the count to an existing data set named COUNT, substitute a > stage command for JOBLOG.

```
< 'data.profile' | count records | > 'profile.count'
```

Now, the records are written to the existing sequential data set named PROFILE.COUNT.

Device Driver Stage Commands

Stage commands, such as < and JOBLOG, that transfer data between the pipeline and a source or target not in the pipeline are called **device drivers**. They provide the interface between the pipeline and sources or targets outside the pipeline.

Every pipeline you use will consist of at least two device drivers. One or more device drivers bring data into a pipeline: one or more device drivers send data out of a pipeline. For example, the following two-stage pipeline reads records from the data set TEST.DATA and writes each record to its output stream. JOBLOG reads the records from its input stream and writes the records to the joblog.

```
< 'test.data' | joblog
```

Figure 9 shows a diagram of this pipeline.

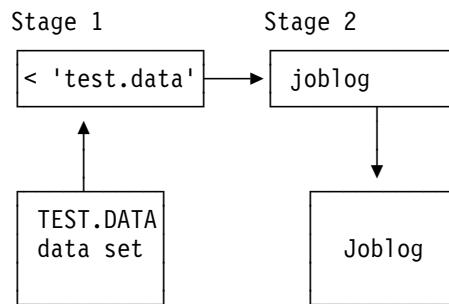


Figure 9. Diagram of a Pipeline with Two Device Drivers

Device drivers work with data sets, pipes, the terminal, REXX variables, and other sources or targets of data outside a pipeline.

Example of Using LITERAL Device Driver

The LITERAL stage command writes a string to its output stream. In the following pipeline specification, LITERAL writes the string Testing 1, 2, 3 to its output stream. The JOBLOG stage command reads the string and writes it to the joblog:

```
literal Testing 1, 2, 3 | joblog
```

To write the same string to a data set, substitute a > (rewrite a sequential data set) stage command for JOBLOG.

The >> (append sequential data set) device driver adds records to the end of an existing physical sequential data set. The following pipeline specification adds the record The End to the data set USERS.DATA:

```
literal The End | >> 'users.data'
```

Frequently-used device drivers are described in Chapter 5, “Device Driver Stage Commands” on page 79.

Filter Stage Commands

Filter stage commands work on data that is passing through the pipeline. They:

- Select records based on the content of the record or on the position of the record in the stream flowing through the pipeline
- Change or rearrange records, including sorting records.

The COUNT stage command, used in “Specifying a Simple Pipeline” on page 6, is a filter stage command. It counts every record that flows into it from its input stream. Then, it writes one record containing that count to its output stream.

The LOCATE stage command is also a filter. It examines the records from its input stream, looking for those that match a specified string. If the record matches, LOCATE writes the record to its output stream. LOCATE can discard records that do not match. See Figure 5 on page 5.

Commonly used filter stage commands are described in Chapter 4, “Filter Stage Commands” on page 39.

Multistream Pipelines

So far in this chapter, the stage commands have one input and one output stream, as Figure 10 shows. Each of these streams is a **primary stream**.

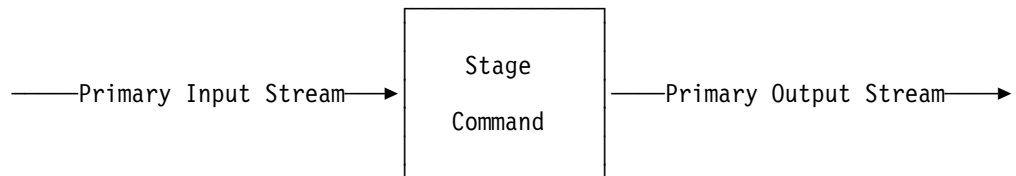


Figure 10. Stage Command with One Input and One Output Stream

Many BatchPipeWorks stage commands read from multiple input streams or produce multiple output streams, or both. Figure 11 shows a stage command that reads two input streams and produces two output streams. The additional streams in this illustration are called **secondary streams**.

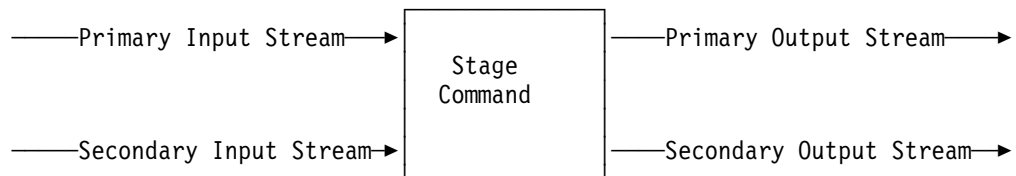


Figure 11. Stage Command with Two Input and Two Output Streams

Some stage commands use even more than two input and output streams. Because the number of streams can be numerous, BatchPipeWorks uses a numbering system to identify the streams. The number 0 identifies the primary stream; the number 1 identifies the secondary stream, and so forth. To see an example of the numbering of streams, see Figure 28 on page 21 and to see more information about numbering multiple streams, see “Identifying Streams” on page 97. The use of multiple streams greatly increases the numbers of problems you can solve with BatchPipeWorks.

Examples of Multistream Pipelines

This section shows that stage commands use multiple input and output streams in different ways. In our first example, the LOCATE stage command looks for instances of the word “Bob” and writes those records to its primary output stream. Records that do not have instances of “Bob” are written to its secondary output stream, as Figure 12 on page 10 shows.

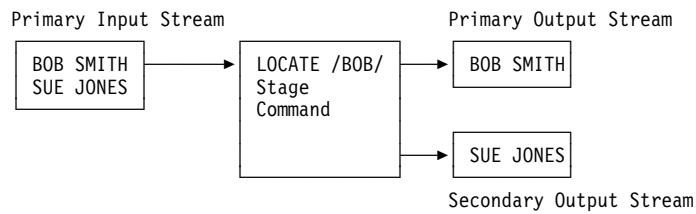


Figure 12. LOCATE with a Secondary Stream

In the second example, the FANOUT stage command, which is described on “FANOUT Stage Command” on page 93, writes each record it reads from its primary input stream to two output streams, as Figure 13 shows.

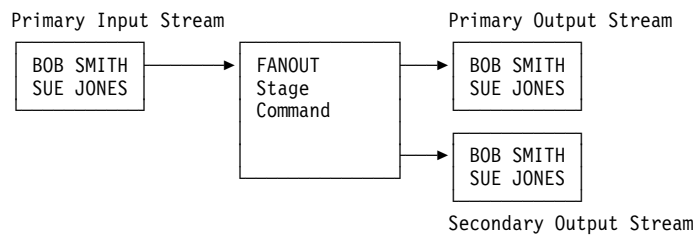


Figure 13. FANOUT with Multiple Output Streams

Each secondary output stream can be processed by other stage commands. In Figure 14, the output of the > 'NOTBOB.DATA' stage command could be input to another stage command. You would have to tell BatchPipeWorks to connect that output stream to another input stream.

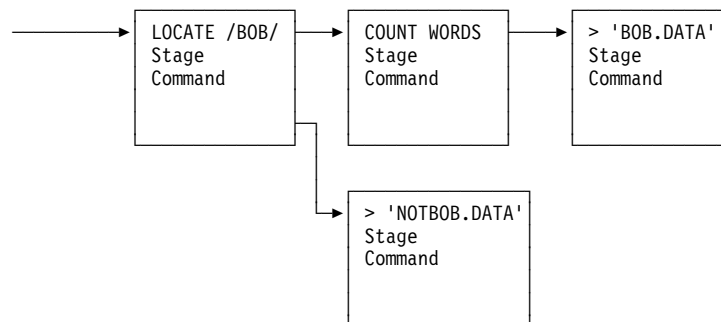


Figure 14. Processing Secondary Outputs

Like LOCATE, most filter stage commands write the records that fail the specified criteria to their secondary output streams. DROP and CHOP are examples of filter stage commands that write the “nonselected” records to their secondary output streams.

Some filter stage commands, for example SPECS, do not have a secondary output stream. The stage command descriptions in *BatchPipeWorks Reference* contain information about the number of input and output streams.

Specifying Multistream Pipelines

When more than one input or output stream is used, we have multiple pipelines. This section describes how to write pipeline specifications where one or more stage commands have multiple input or output streams. Let's start with a simple example of two independent pipelines, as shown in Figure 15:

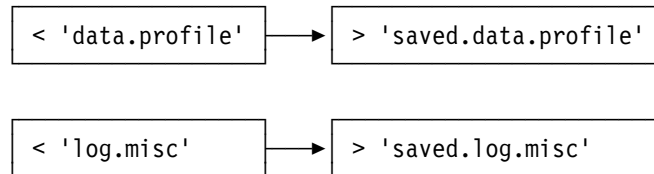


Figure 15. Diagram of Two Independent Pipelines

When you define two pipelines on the same pipeline specification, the two pipelines are called a **pipeline set**.

To tell BatchPipeWorks where one stream ends and another begins, start the pipeline specification by defining an **end character**. Then, use that character to separate one pipeline from another, as Figure 16 shows. Do not use an end character after the last pipeline. Specifying one would cause an error. Use any alphanumeric character for the end character, providing you have not defined that character as a stage separator or an escape character. See “Global Options on the Pipeline Specification” on page 19 for information about specifying the ENDCHAR.

In examples in this book, a question mark is used as the end character.

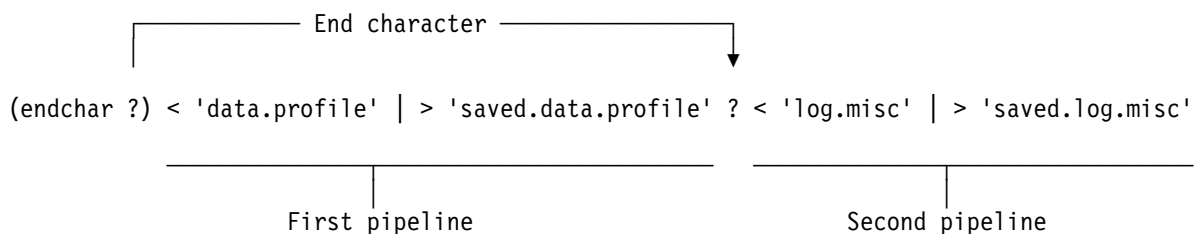


Figure 16. Specifying Two Independent Pipelines

You would get the same results by using two pipeline specifications:

```
< 'data.profile' | > 'saved.data.profile'

< 'log.misc' | > 'saved.log.misc'
```

Now we will learn how to specify pipeline sets, where the pipelines are not independent.

Defining Multistream Pipelines through Labels

This section describes the different ways that you define streams. The diagrams in this section differ from the earlier diagrams; here, the direction of flow is vertical. It is recommended that you use the vertical-flow diagram as the first step in writing your own pipeline specifications.

Let's consider a multistream pipeline in which:

```
< 'test.data'
    Reads the TEST.DATA data set and writes records to its output stream.

LOCATE /Bob/
    Finds all records that contain the name "Bob," and produces a primary
    output stream of those records and a secondary output stream of all
    other records.

< 'Bob.DATA'
    Writes records from its primary input stream to an existing data set
    named BOB.DATA.

< 'NOTBOB.DATA'
    Writes records from its secondary input stream to an existing data set
    named NOTBOB.DATA.
```

Another important character in your pipeline specifications is the **label**; BatchPipeWorks uses it to connect streams. The first instance of the label is called the *label reference*.

A label can have from one to eight alphanumeric characters. It must be followed by a colon (for example, a:). Keep in mind that if you have two of the same stages (for instance two LOCATE stages) in one pipeline, each LOCATE stage must have a unique label, as Figure 26 on page 19 shows.

Figure 17 shows a vertical-flow diagram that is a pipeline set. BatchPipeWorks uses the character "a" to connect the stage command LOCATE with its two output streams.

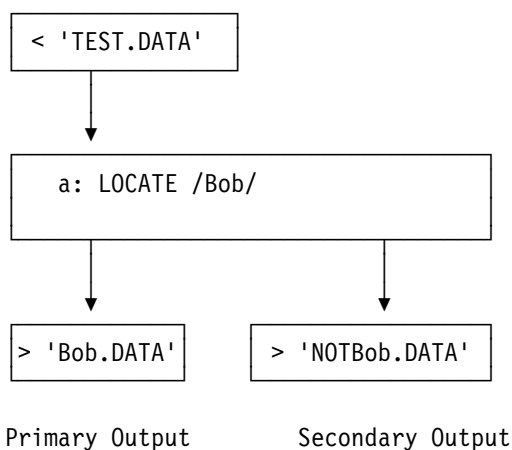


Figure 17. Defining and Referencing Labels

In vertical-flow diagrams, the primary input and output streams appear on the left; secondary streams appear to the right of the primary streams. The labels appear with the stage command that reads from multiple input streams or produces multiple streams.

The label definition on a stage allows BatchPipeWorks to connect the multiple streams of intersecting pipelines, and each label reference in a subsequent pipeline defines a new input and output stream for the stage command. Pipelines are defined by input and output streams, even if one of the streams are not used.

Figure 18 shows how to write the two-pipeline pipeline set, including the important characters that tell BatchPipeWorks how the streams connect.

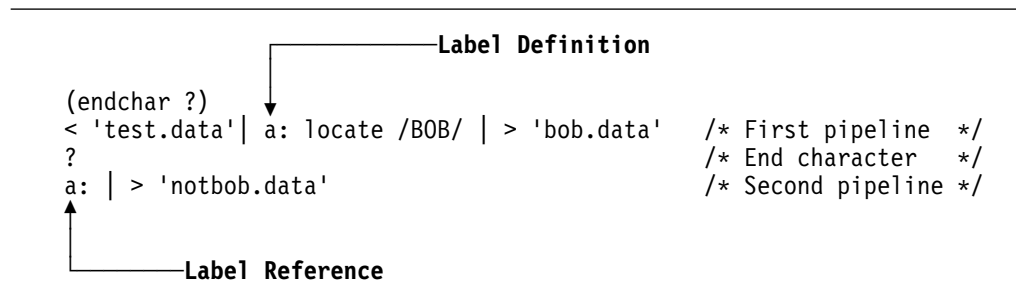


Figure 18. Defining Labels

Notice that the second time you specify the label, you omit the stage commands associated with the label; in our example, the LOCATE /Bob/ stage command is not repeated, only the label is used.

Defining Multistream Pipelines that Have Secondary Output Streams

Figure 19 shows the vertical-flow diagram for a pipeline set where you are using the secondary output of a stage.

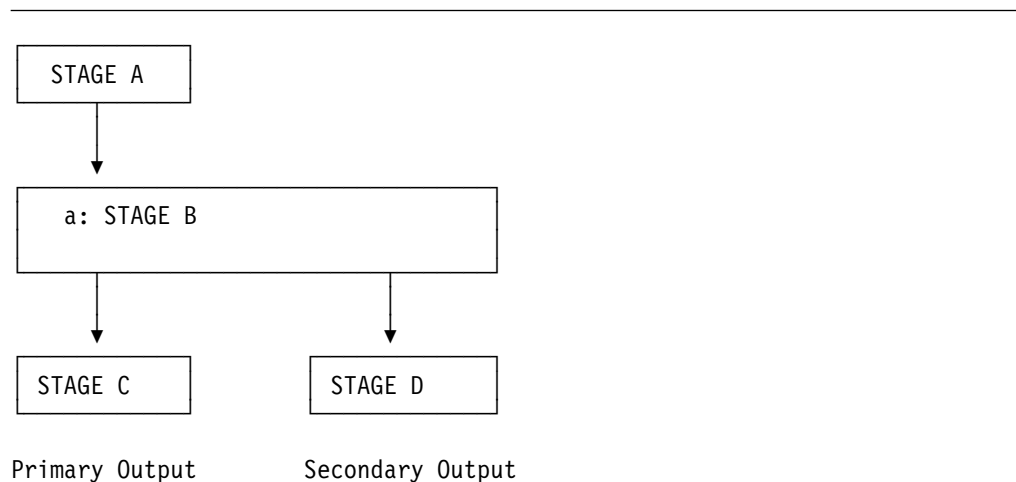


Figure 19. Secondary Output Streams

The diagram shows that the secondary output of STAGE-B is the primary input of STAGE-D. The following steps outline the process of specifying this pipeline:

1. Draw the vertical-flow diagram so that the primary streams and secondary streams line up top-to-bottom. Draw the primary stream on the left with

subsequent streams to the right. Define a label for every stage command that either reads multiple input streams or produces multiple output streams. The vertical-flow diagram for our example is in Figure 20 on page 14, where the shaded lines indicate the two pipelines.

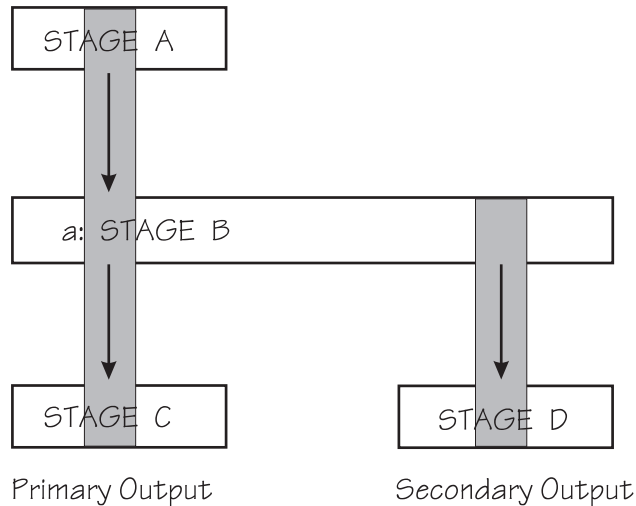


Figure 20. Vertical-Flow Diagram of a Pipeline Set with Two Pipelines

Note that records do not flow directly from STAGE A to STAGE B to STAGE C. Some records that come into STAGE B from STAGE A might be written to STAGE D.

2. Write the two pipelines, placing stage separators between stage commands. Define the end character and place the end character between pipelines, but not at the end of the pipeline.

```

/*****
/* Two pipelines
/*****
(endchar ?)
stage-a | a:stage-b | stage-c      /* First pipeline
?                                  /* End character
a: stage b | stage-d              /* Second pipeline

```

3. Find the second mention of the label and remove the stage command associated with it. Use the label by itself.

```

/*****
/* The pipeline specification is complete and accurate!
/*****
(endchar ?)
stage-a | a:stage-b | stage-c      /* First pipeline
?                                  /* End character
a: | stage-d                      /* Second pipeline

```

Example of a Two-Pipeline Specification

The pipeline set shown in Figure 21 on page 15 consists of two pipelines in which:

```
< 'VOTE.DATA'
```

Reads the VOTE.DATA data set and writes records to its output stream.

LOCATE /YES/

Finds all records that contain the string YES and writes these records to its primary output stream; it writes all other records to its secondary output stream.

> 'YES.DATA'

Writes the records from its input stream to the existing data set YES.DATA.

> 'NO.DATA'

Writes the records from its input stream to the existing data set NO.DATA.

BatchPipeWorks uses the label `a` to connect LOCATE's secondary output stream to the primary input of the `> 'no.data'` stage.

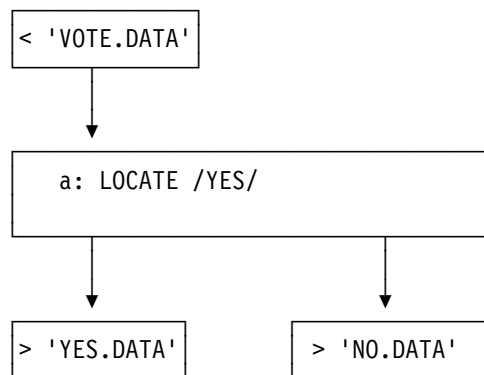


Figure 21. Vertical-Flow Diagram of Secondary Output Streams

The pipeline specification is as follows:

```
/* Pipeline specification */
(endchar ?)
< 'vote.data' | a: locate /YES/ | > 'yes.data' /* First pipeline */
? /* End character */
a: | > 'no.data' /* Second pipeline */
```

You might find the following specification clearer.

```
/* Using meaningful comments */
(endchar ?)
< 'vote.data' /* Read the VOTE.DATA data set */
| a: locate /YES/ /* Select records containing YES */
| > 'yes.data' /* Write them to YES.DATA */
?
a: /* Process LOCATE rejects */
| > 'no.data' /* Write them to NO.DATA */
```

Defining Multistream Pipelines that Have Secondary Input Streams

Figure 22 shows the vertical-flow diagram for a pipeline set where a stage command uses multiple input streams, in this case primary and the secondary streams. INLAB is the label for the stage.

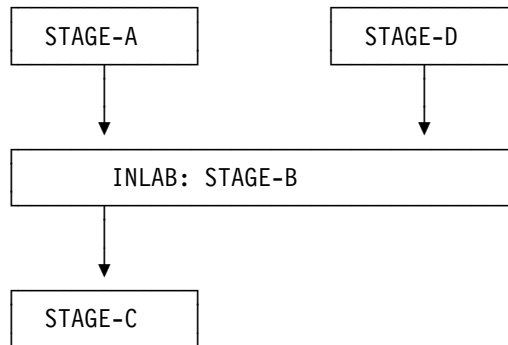


Figure 22. Secondary Input Streams

The vertical-flow diagram shows that the primary output stream of STAGE-D is the secondary input to STAGE-B. The following example shows how you would tell BatchPipeWorks to make the correct connections.

```
/* *****  
/* The pipeline specification  
/* *****  
    (endchar ?)  
    stage-a | inlab: stage-b | stage-c  
    ?  
    stage-d | inlab:
```

Because the matching label is used at the end of the second pipeline, BatchPipeWorks knows that the primary output stream of STAGE-D is the secondary input stream of STAGE-B; any record that STAGE-D writes becomes the secondary input to STAGE-B.

Defining Multistream Pipelines with Secondary Input and Output Streams

At times, you might want to use both the secondary input stream and the secondary output stream of a stage command. Figure 23 shows a stage command that has both secondary input and secondary output streams. CON is the label.

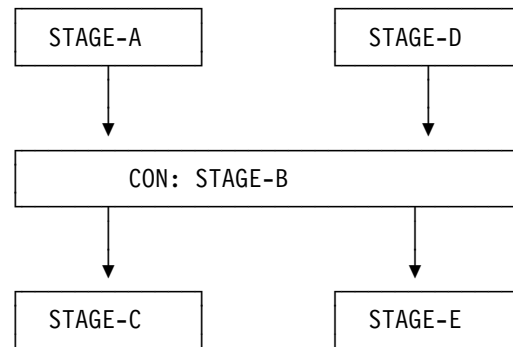


Figure 23. Vertical-Flow Diagram of Secondary Input and Secondary Output Streams

The vertical-flow diagram shows the following connections:

- Primary output from STAGE-A becomes primary input to STAGE-B
- Primary output from STAGE-D becomes secondary input to STAGE-B
- Primary output from STAGE-B becomes primary input to STAGE-C
- Secondary output from STAGE-B becomes primary input to STAGE-E

The following example shows the pipeline specification that makes the appropriate connections:

```
(endchar ?)
stage-a | con: stage-b | stage-c
?
stage-d | con: | stage-e
```

The label reference defines a secondary input and a secondary output stream for STAGE-B. In this case, both of the new pipelines are connected.

Using Several Secondary Streams

Previous examples used the secondary streams of a single stage command. To solve complex problems, however, you might need secondary streams that consist of several stage commands.

To use secondary streams of several stages commands, you define different labels. For example, suppose you want to write all records containing the string BOB to one data set. Of the remaining records, you want to write those containing SUE to another data set, and all remaining records to a third data set. Figure 24 shows the vertical-flow diagram.

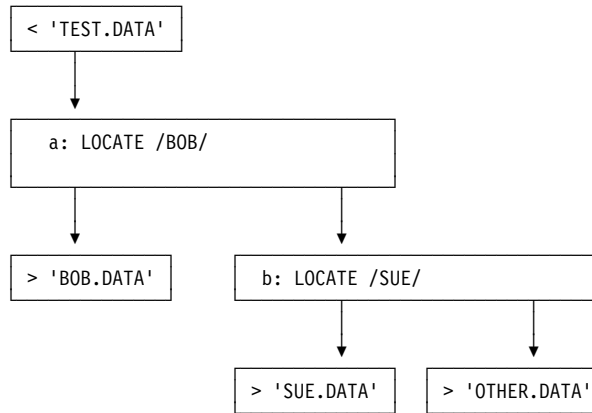


Figure 24. Using Several Secondary Streams

The three pipelines are shown in the following illustration:

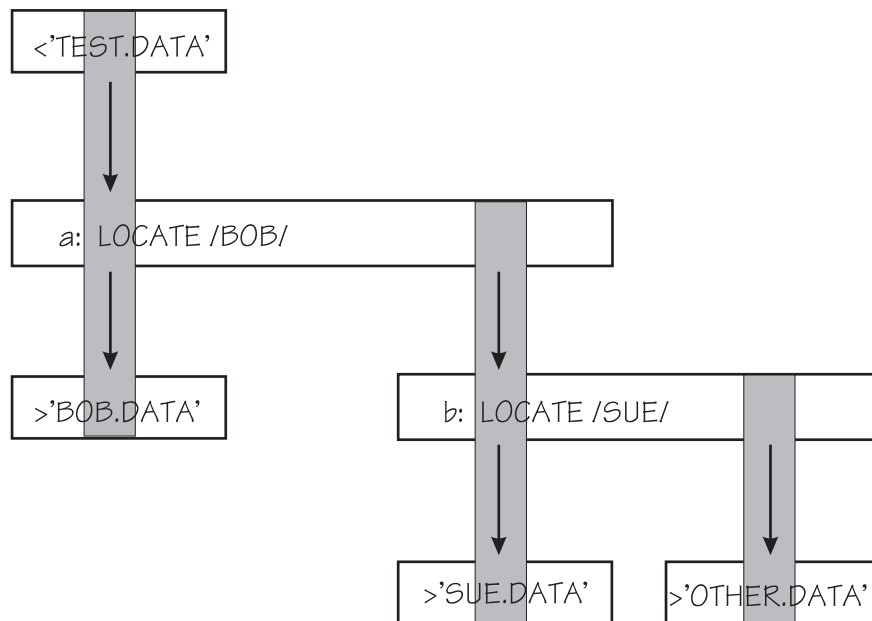


Figure 25. Vertical-Flow Diagram of a Pipeline Set with Three Pipelines

Remembering the rule that when you specify the label for the second time, you drop the stage commands associated with the label. Figure 26 shows the pipeline specification:

```
/*-----*/
/*      -- Writing records to three streams      */
/*-----*/
(endchar ?)
< 'test.data'
| a: locate /BOB/
| > 'bob.data'
?
a:
| b: locate /SUE/
| > 'sue.data'
?
b:
| > 'other.data'
```

Figure 26. Three Pipelines in One Pipeline Specification

The LOCATE stage in the second pipeline (`locate /SUE/`) defines a new label (b). The secondary output stream of this LOCATE consists of all records that have neither BOB nor SUE, which is precisely what we want in the last output data set. The third pipeline writes the records to OTHER.DATA. The label reference in the third pipeline causes BatchPipeWorks to connect the secondary output stream of `locate /SUE/` with the primary input stream of `> 'other.data'`.

Global Options on the Pipeline Specification

The pipeline specification includes some options that define some characteristics of the specification, such as the stage separator, labels, and end characters.

These options in the pipeline specification are called **global options**. Additional global options allow you to:

- Trace execution of the pipeline
- Control the level of messages displayed

A complete list of the global options is in the *BatchPipeWorks Reference*.

To show you how to specify a global option, let's use the STAGESEP option. STAGESEP assigns the stage separator to a different character. By default, the stage separator is a vertical bar (|). To use a question mark as the stage separator, for example, specify `stagesep ?` as shown in Figure 27.

```
pipe (stagesep ?) literal one two three ? count words ? terminal
3
READY
```

Figure 27. Specifying a Global Option

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

Storage Considerations with BatchPipeWorks

Because BatchPipeWorks usually lets each stage process only one record at a time, only several records are in the pipeline at any moment and minimal storage is used. This is an important characteristic of pipelines. It means if you process a data set that contains one million records, you do not need to provide a large enough region to hold all those records.

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

Avoiding Pipeline Stalls

With multistream pipelines you may see an error known as a stall. A stall occurs when BatchPipeWorks cannot run any of the stages because every stage is waiting for another stage to perform some function. Usually stalls are caused by stages that read multiple input streams in a particular order or that need records to be available on more than one stream at the same time. A stall occurs when the preceding stages do not deliver records in the order needed or do not provide multiple records concurrently.

When a stall occurs, you receive a return code of -4095, messages saying the pipeline is stalled, and messages listing the state of all the stages in the pipeline.

What can you do when a pipeline stalls?

1. First, look at any stages that have multiple input streams. Of these stages, identify any stages that need records in a particular order (such as FANIN) or that need more than one record at a time (such as SPECS and OVERLAY).
2. Next, analyze the pipeline to see what order the records are being delivered to these stages. You need to look at the stages that are supplying records. It helps to draw a diagram of the pipeline. Look for earlier stages that have secondary outputs connected. These stages often deliver records in a particular order and that order is not what the stage combining the streams needs.
3. After finding the problem, you can either change the order of delivery of the records, or you can change the stage combining the streams to match the supplied order.

A common stall is shown in Figure 28. There may be many other stages between FANOUT and FANIN, but in many cases these other stages don't matter. It is the FANOUT/FANIN combination that causes the problem. FANIN needs to read all the records of one stream before reading the next. FANOUT, however, writes one record to each of its output streams.

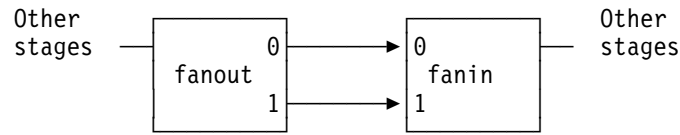


Figure 28. Example of a Stall Involving a Stage that Needs Records in Order

FANOUT writes a record on its output stream 0. FANIN reads this record. Then FANOUT tries to write a copy of the record on its output stream 1. It waits for FANIN to read the record, but FANIN is waiting for FANOUT to write another record on stream 0. FANIN will not read from its input stream 1 until input stream 0 is empty. The pipeline is stalled.

To learn how to fix pipeline stalls once they occur, see “Fixing Pipeline Stalls” on page 110.

Chapter 3. Using BatchPipeWorks with BatchPipes

This chapter contains General-Use Programming Interface and Associated Guidance Information.

BatchPipes is an excellent product for installations that want to reduce the elapsed time of their batch applications. But, using BatchPipes may solve only part of the installation's needs. Sometimes, BatchPipes users need to customize their jobs to meet specific requirements; they need additional function in the form of programs that:

- Harden data; that is, write a copy of the intermediate data to an external device
- Write duplicate records to multiple readers
- Select records according to certain criteria
- Change the contents of each record as it is in the pipeline
- Consolidate records passed from multiple jobs.

That's where BatchPipeWorks comes in.

This chapter assumes that you are familiar with BatchPipes and the typical BatchPipes pipeline, shown in Figure 29. Data flows from JOBA to JOBB through a BatchPipes pipe and JOBA and JOBB run in parallel.

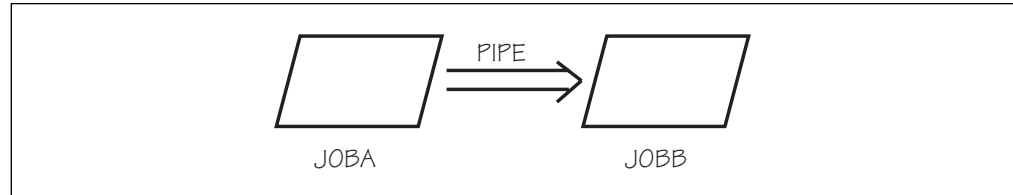


Figure 29. A Typical BatchPipes Jobstream

You can use BatchPipes with BatchPipeWorks in three ways:

- As a **fitting**, one or more BatchPipeWorks stage commands that extend the work of a writer or reader in a BatchPipes pipeline.
- As a fitting associated with a job that is not in the BatchPipes pipeline; this program is known as a **half-pipe fitting**
- As a filter job in the BatchPipes pipeline.

Illustrations in this section show how the stage commands apply in each of these cases: they mention stage commands that you will become familiar with as you read the chapters that describe how to use BatchPipeWorks.

Using BatchPipeWorks as a Writer or Reader Fitting

When used within the BatchPipes pipeline, BatchPipeWorks intercepts records in one of two places: immediately after the writer writes to the pipe or immediately before the reader reads the records from the pipe. In effect, BatchPipeWorks gets control “between” the writer and the pipe (for a **writer fitting**) or “between” the pipe and the reader (for a **reader fitting**).

Figure 30 on page 24 shows a writer fitting that consists of three stage commands:

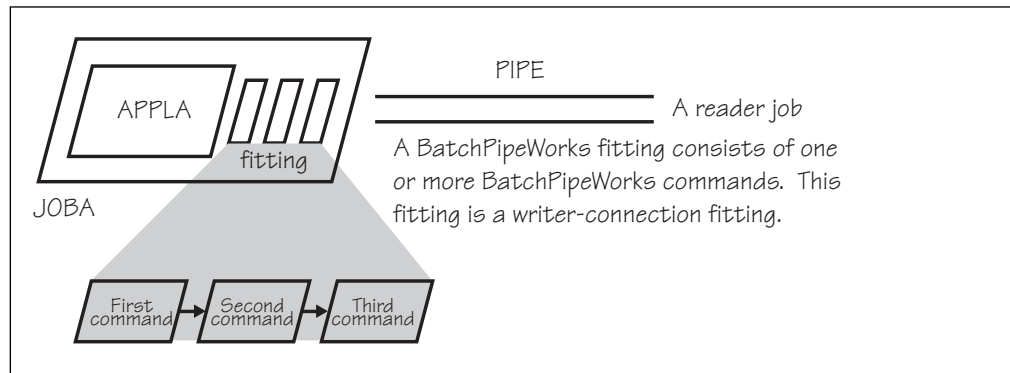


Figure 30. A BatchPipeWorks Fitting

In Figure 30, JOBA consists of APPLA, first command, second command, and third command run serially. JOBA and the reader job run in parallel.

You might, for example, use a fitting if you want a second copy of the output records sent to tape for backup purposes. This process, known as **hardening data**, is illustrated in Figure 31. The three stage commands that comprise this fitting are BPREAD, QSAM, and BPWRITE.

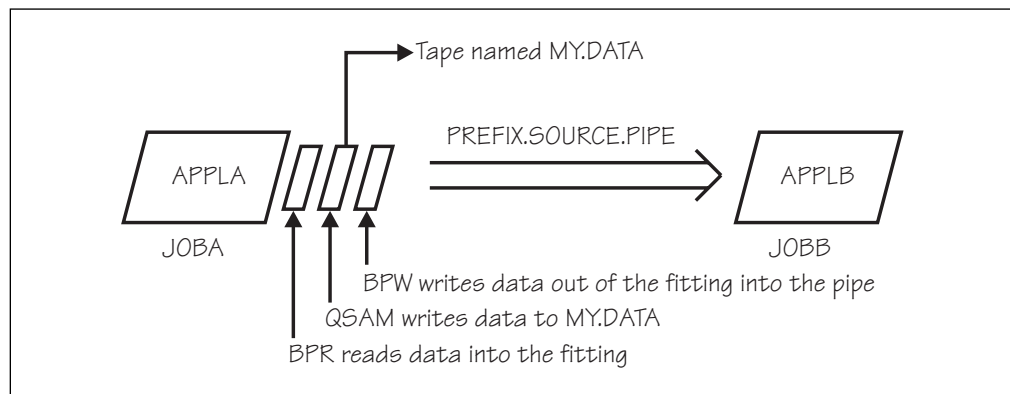


Figure 31. Example of a Fitting that Hardens Data

“Using a BatchPipeWorks Fitting” on page 27 describes fittings and the stage commands you specify on the JCL DD statements.

Using BatchPipeWorks with Jobs Not in the BatchPipes Pipeline

You can use BatchPipeWorks and BatchPipes together with jobs that are not in a pipeline to extend the function of a batch job. An example would be the non-BatchPipes job APPLC that processes records from the large data set named USA.ACCT and produces a mailing. Suppose you want to produce a mailing for California residents. You could use a BatchPipeWorks fitting to select only California records, as Figure 32 on page 25, shows:

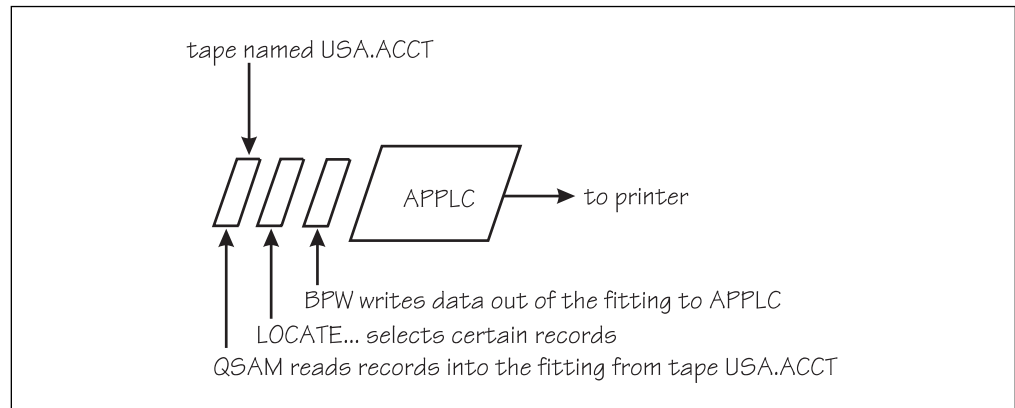


Figure 32. Example of a Half-Pipe Fitting

You define APPLC to BatchPipes and use BatchPipeWorks stage commands to do the work. This kind of fitting, called a **half-pipe fitting**, is described in “Using Fittings Outside the BatchPipes Pipeline” on page 36.

Using BatchPipeWorks as a Filter Job in a BatchPipes Pipeline

The third way BatchPipeWorks can be used by BatchPipes users is as a series of stage commands that actually **comprise** a job, in this case a BatchPipes filter job. An example of using BatchPipeWorks as a job would be if you wanted a filter job to harden data, as Figure 33 shows:

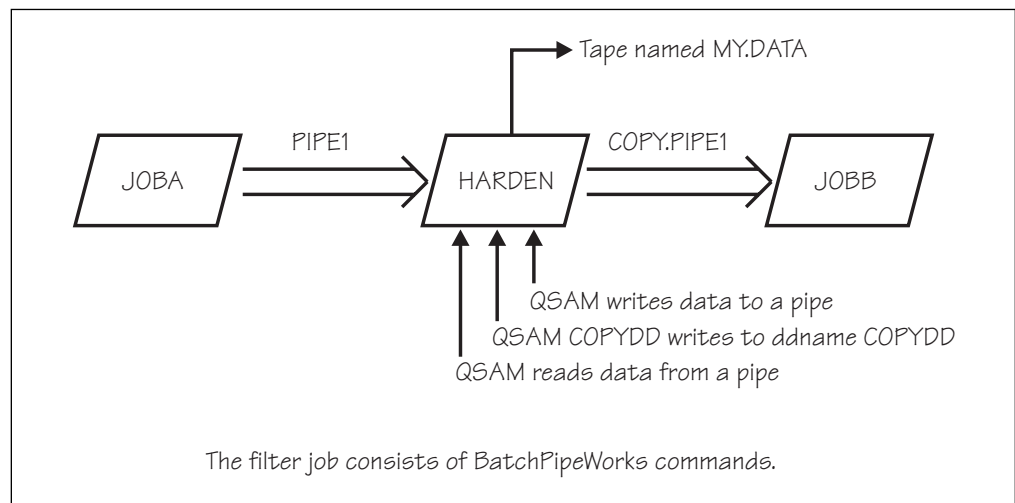


Figure 33. Example of BatchPipeWorks as a BatchPipes Filter Job

You would use the stage commands as a filter job, rather than a fitting, if you saw value in increasing (or maintaining) the parallelism of jobs in the pipeline. The benefits and the tradeoffs of using BatchPipeWorks in this way are described in “Using BatchPipeWorks in Filter Jobs” on page 124.

BatchPipeWorks Terminology

BatchPipes and BatchPipeWorks both use terms associated with pipes and plumbing. Two terms appear in the *BatchPipes Users Guide and Reference* and the two books that support BatchPipeWorks with slightly different meanings. They are “pipeline” and “stage.” The following graphics illustrate the meanings of these terms. After reading through this section, you will find the terminology in this book and the companion reference book to be fairly easy.

BatchPipes Pipeline versus BatchPipeWorks Pipeline: The *BatchPipes Users Guide and Reference* uses the term “pipeline” to mean a job-->pipe-->job association, as described in Figure 34. Data flows from JOBA to JOBB; JOBA and JOBB run concurrently.

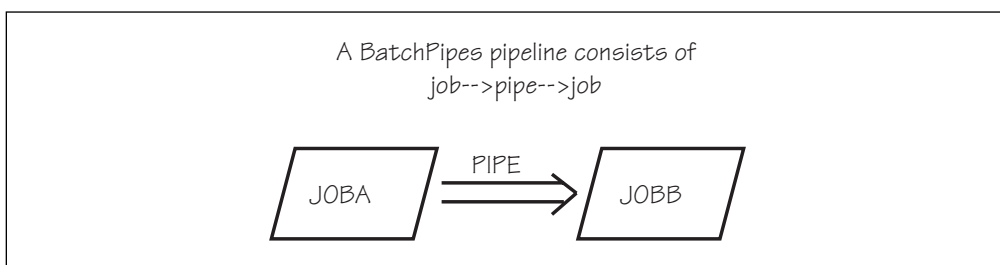


Figure 34. A BatchPipes Pipeline

The BatchPipeWorks “pipeline” is the series of stage commands that comprises a fitting, as Figure 35 shows. Data flows from command A to B to C; Commands A, B, and C run one at a time.

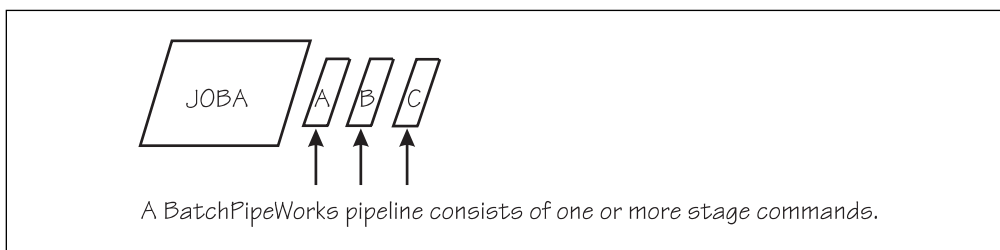


Figure 35. Comparing a BatchPipes Pipeline with a BatchPipeWorks Pipeline

A BatchPipes pipeline and BatchPipeWorks pipeline are similar in that both are conduits for data flow.

BatchPipes Stage versus the BatchPipeWorks Stage: The *BatchPipes Users Guide and Reference* uses the term “stage” to mean a processing element, one or more jobs, that performs a particular function on the data. Figure 36 represents a two-stage pipeline.

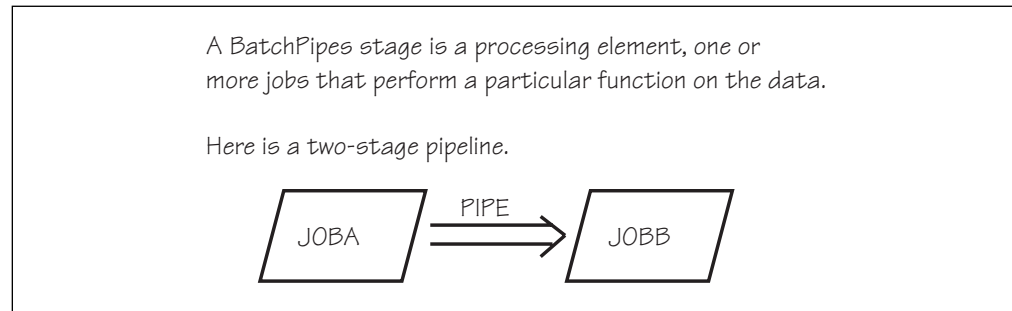


Figure 36. A BatchPipes Stage

A BatchPipeWorks stage is the processing element, one stage command, that performs a particular function on the data, as Figure 37 shows. The term “stage” command is used because the processing occurs at a stage in the BatchPipeWorks pipeline.

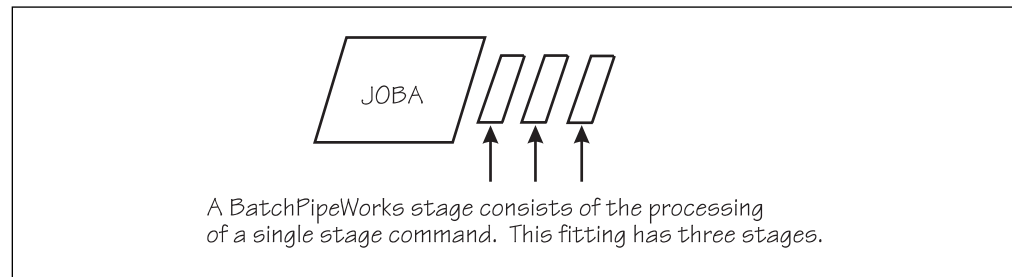


Figure 37. Comparing a BatchPipes Stage with a BatchPipeWorks Stage

A BatchPipes stage and a BatchPipeWorks stage are similar in that both represent a processing step. In BatchPipes, stages run in parallel; in BatchPipeWorks, stages run one after another.

Using a BatchPipeWorks Fitting

As mentioned earlier in this chapter, a fitting is a series of BatchPipeWorks stage commands that are added to an existing BatchPipes job. The fitting is placed on either the writer or reader end of a BatchPipes pipe, or both. A writer fitting intercepts data before it flows into the pipe. Figure 38 on page 28 shows a writer fitting in a two-job BatchPipes pipeline:

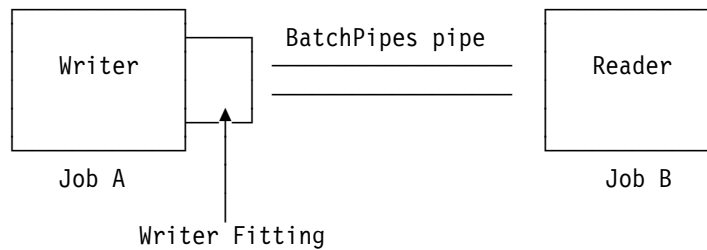


Figure 38. Example of a Writer Fitting

A reader fitting intercepts data after it flows through the pipe, as Figure 39 shows:

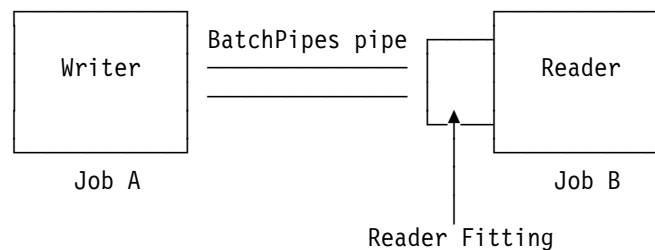


Figure 39. Example of a Reader Fitting

You can have fittings on any or all connections of a pipe.

Compared with a filter, a fitting might reduce CPU time because a fitting does not require an additional pipe and, for some fittings, the number of records piped is less. Also, a fitting does not need an initiator and a separate address space.

A fitting runs in the same address space as the reader or writer job that establishes the fitting. Fittings provide an easy way to add flexibility to an existing job-pipe-job sequence without the need to change application code.

A fitting has advantages over adding a BatchPipeWorks filter job to a BatchPipes pipeline. Although a filter job can add function and flexibility to the BatchPipes pipeline, it requires additional resources. To add a filter job you need to allocate at least one extra pipe, add a connection and a job, and initiate that job. The new job contends for resources with the current jobs. A fitting can provide the same function with less overall resource consumption. For more information, see “Using BatchPipeWorks in Filter Jobs” on page 124.

A fitting can change the rate the data flows through the pipe. If the reader and writer connections are evenly paced so records are produced and consumed at the same rate, introduction of the fitting can result in more waits by one of the connections. If the writer is waiting longer, an increase in the pipe depth (BUFNO= value) can sometimes improve the situation. See “Where to Place the Fitting” on page 32.

If possible, you might need to change the number of cloned readers and writers to balance the pipeline.

Data from the BatchPipes ISPF monitor and STATUS commands do not provide information about the fitting itself; rather they provide information about the entire job, including the fitting. The monitor panels and DISPLAY messages reflect the existence of a fitting.

Making the JCL Changes for a Fitting

To use a fitting, you need to change the JCL of the BatchPipes job associates with the fitting. To change the JCL, use the FIT or FITDD subparameters on the JCL SUBSYS parameter, and use the BPREAD and BPWRITE stage commands to read data into and out of the fitting.

Adding the FIT Subparameter to the DD Statement

To change the JCL to include a fitting, add a 'FIT=*fitting*' subparameter on the SUBSYS parameter. This tells the BatchPipes subsystem to send the data through the defined fitting before sending the data to the next stage of the BatchPipes pipe. The fitting contains BatchPipeWorks stage commands separated by stage separators that specify the function that is to be performed on the data flowing through the fitting.

Figure 40 shows the JCL for a writer job that uses BatchPipes:

```
//PROCESSA JOB ...  
//stepname EXEC PGM=pgmname,...  
//pipeout DD DSN=some.pipe,DCB=(...),SUBSYS=BP01
```

Figure 40. JCL for a BatchPipes Writer

To use a fitting with this job, add a FIT subparameter on the SUBSYS parameter:

```
//PROCESSA JOB ...  
//stepname EXEC PGM=pgmname,...  
//pipeout DD DSN=some.pipe,DCB=(...),SUBSYS=(BP01,'FIT=fitting')
```

Figure 41. Example of the FIT Subparameter in JCL

Notes on using the FIT subparameter:

- Enclose the SUBSYS subparameters in parentheses, separate FIT parameters with a comma, and enclose the FIT=*fitting* in single quotation marks.
- Do not enclose the parameters on FIT in parenthesis. In other words, the following is not valid:
SUBSYS=(BP01,'FIT=(BPCOPY SPOUT02)').
- If the pipeline specification exceeds a single line, use the FITDD subparameter, as described in “Adding the FITDD Subparameter to the DD Statement” on page 30.

Example of using the FIT Subparameter

To make one copy of the data flowing through the pipe and send the copy to a data set, change the JCL to include a fitting, as Figure 42 shows:

```
//PROCESSA JOB ...
//stepname EXEC PGM=pgmname,...
//pipeout DD DSN=some.pipe,DCB=(...),
//  SUBSYS=(BP01,'FIT=BPREAD|QSAM COPYDD|BPWRITE')
//COPYDD DD DSN=....
```

Figure 42. Example of Using FIT to Make a Copy of Piped Data

The BatchPipeWorks stage commands on the FIT subparameter indicate the stage commands that comprise the fitting:

Table 1. Example of BatchPipeWorks Commands	
BatchPipeWorks command	What the command does
BPRead	Reads data into the fitting
QSAM ddname	Writes data to tape <i>ddname</i>
BPWrite	Writes data out of the fitting

See “Using BPREAD, BPWRITE, and BPCOPY in Fittings” on page 31 for more about BPREAD and BPWRITE.

Adding the FITDD Subparameter to the DD Statement

Use the FITDD instead of the FIT subparameter when:

- The pipeline specification uses more than one line.
- You want the stage commands to have comments associated with them. For information on adding comments to your pipeline, see “Adding Comments to Your Pipeline” on page 31.

The subparameter is FITDD=*ddname*, where *ddname* is a DDNAME given to the DD statement that contains the BatchPipeWorks stage commands. Figure 43 shows how to use the FITDD subparameter with a BatchPipes job:

```
//PROCESSA JOB ...
//stepname EXEC PGM=pgmname,...
//pipeout DD DSN=some.pipe,DCB=(...),
//  SUBSYS=(BP01,'FITDD=PIPESPEC')
//COPYDD DD DSN=....
//PIPESPEC DD *
//  BPREAD |
//  QSAM COPYDD |
//  BPWRITE
/*
```

Figure 43. Example of Using FITDD to Make a Copy of Piped Data

The FITDD subparameter identifies PIPESPEC as the DDNAME that contains the pipeline specification. When you use the FITDD subparameter, enclose the

SUBSYS subparameters in parentheses, separate the parameters with a comma, and enclose the FITDD=*ddname* in single quotation marks.

Note about sequence numbers

When you use the FITDD subparameter, sequence numbers can cause problems with the fitting. Adding a comment to every line successfully avoids this problem.

Adding Comments to Your Pipeline

You can add comments to your pipeline for use in the future. The CMT=*delimiter* parameter establishes the comment delimiter, which can be 1 to 2 characters in length. Because the CMT=*delimiter* must be enclosed in single quotation marks, you cannot use a single quotation mark (') as a delimiting character. Figure 44 shows the previous fitting with comments delimited by the characters --. Comments following the delimiting characters are stripped prior to execution of the pipeline.

```
//PROCESSA JOB ...
//stepname EXEC PGM=pgmname,...
//pipeout DD DSN=some.pipe,DCB=(...),
//  SUBSYS=(BP01,
//    'FITDD=PIPESPEC','CMT=--')
//COPYDD DD DSN=....
//PIPESPEC DD *
    BPR |                -- Read data from the pipe
    QSAM COPYDD |        -- Copy the data to file COPYDD
    BPW          -- Send the data along the pipe
/*
```

Figure 44. Example of Adding Comments to a Fitting

The CMT parameter applies only when you specify the fitting using the FITDD subparameter.

Using BPREAD, BPWRITE, and BPCOPY in Fittings

The examples in this chapter use the BPREAD (BatchPipes READ) and BPWRITE (BatchPipes WRITE) stage commands in fittings. To interface with BatchPipes, the first stage command in the fitting must be BPREAD, which intercepts the data and reads the data into a fitting. To pass the data back to BatchPipes, use BPWRITE as the last stage of the fitting. BPREAD and BPWRITE can be shortened to BPR and BPW, which is the way they are used in the rest of the chapter.

Use BPCOPY when you want to harden the data in the fitting. BPCOPY intercepts data from the BatchPipes pipe data set, writes the data to one or two files, and returns the data to the BatchPipes pipe data set.

Considerations for Fittings

This section helps you choose where to place a fitting and what specific stage commands to use or avoid. Also, it can help you avoid the condition called a pipeline stall. The advice given in this section does not apply to half-pipe fittings.

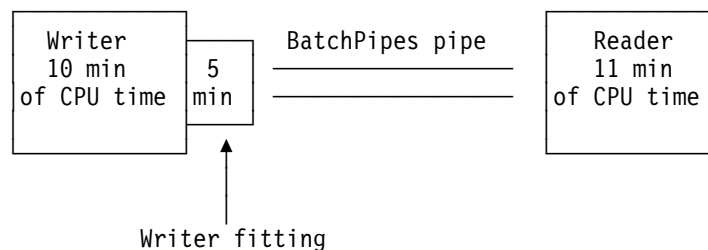
Where to Place the Fitting

A fitting can be on a writer connection or a reader connection. If a fitting causes the pipeline to become unbalanced (one job uses more CPU time than its partner), consider splitting the function between two fittings. This process might require that you try several approaches and see which is the most efficient. Here are some suggestions for where to place the fitting:

- For better pipe balance, place the fitting on the connection that consumes the least amount of CPU time.

In a balanced pipeline, each job consumes about the same amount of CPU time. This limits the amount of wait time spent by the jobs that don't need as much CPU time. If you have a filter that consumes enough CPU time to unbalance the BatchPipes pipeline, consider making the fitting into a filter job.

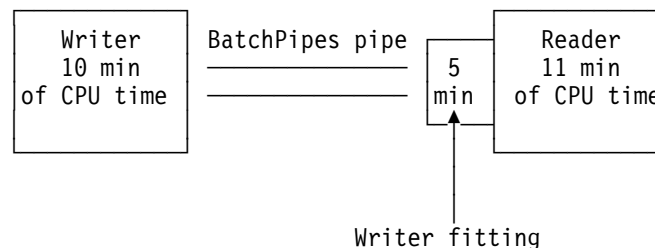
For example, suppose a BatchPipes pipe consists of a writer that uses 10 minutes of CPU time and a reader that uses 11 minutes. You would like to add a fitting that uses 5 minutes of CPU time. If you add the fitting to the writer connection, you have the following imbalance, where the writer uses 15 minutes of CPU time and the reader uses 11 minutes:



Total time of longest-running job: 15 minutes

Figure 45. Balancing a BatchPipes Pipeline With a Writer Fitting

If you were to add the fitting to the reader connection, you have an imbalance, with the writer using 10 minutes of CPU time and the reader using 16 minutes:



Total time of longest-running job: 16 minutes

Figure 46. Balancing a BatchPipes Pipeline With a Reader Fitting

In this case, you should consider adding a filter job to the pipeline, as follows:

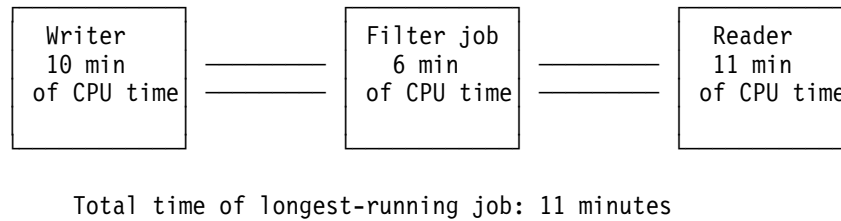


Figure 47. Example of a Balanced BatchPipes Pipeline Using a Filter Job

Overall CPU time does not increase because the jobs run in parallel. Note, however, that you are adding another initiator.

- To minimize the amount of data piped through the BatchPipes pipe, place the following types of fittings on the writer-connection rather than the reader-connection: fittings that drop records or select records.
- If possible, run “large” fittings as separate filter jobs.

Specific Stage Commands to Avoid

Some BatchPipeWorks stage commands do not work well in fittings. The following list describes these stage commands:

- Don't use stage commands that buffer records until all records are brought into the stage command. Use of these stage commands defeats the purpose of BatchPipes jobs running in parallel. These stage commands are BUFFER, ELASTIC, INSTORE, OUTSTORE, and SORT.
- Don't use <, >, >> stage commands in a fitting.
- Use QSAM to perform I/O in fittings. One word of caution when using QSAM in a fitting: do not use the RLSE subparameter on the JCL SPACE parameter for the files that are output from QSAM. That subparameter changes resource allocations that could cause problems processing the fitting.
- Be careful about changing the record length in a fitting. For example, if the reader job expects 80-byte records, be sure the records leave the fitting in that length. You can use CHOP and PAD to change the record length. (Remember that the LRECL values of BatchPipes writer/reader partners must be the same.)
- Don't use a stage command that reorders records, unless the reader from the fitting doesn't care about the order. It is easier to keep records in order than to sort them at the end. For more information, see “Maintaining the Relative Order of Records” on page 112.

Avoiding BatchPipes Pipeline Stalls in Multistream Pipelines

If you have a multistream pipeline as part of your fitting, you should test your fitting to insure that a pipeline stall will not happen. A pipeline stall will stop your fitting from running, which stops the flow of data into the BatchPipes pipe. To learn how to fix pipeline stalls once they occur, see “Fixing Pipeline Stalls” on page 110.

Recommended Practices When You Use Fittings

This section lists three ways you can avoid confusion as you use fittings **and** half-pipe fittings:

- Keep all your fittings and half-pipe fittings in a central location, such as a partitioned data set.
- Use a single comment delimiter throughout the installation. **IBM recommends** you use the characters “/*.”
- Use naming conventions that differentiate a fitting from a half-pipe fitting and clearly identify the data sets that are neither a fitting nor a half-pipe fitting.

Examples of Typical BatchPipes Fittings

As you have seen, BatchPipeWorks fittings provide an easy way to change your business applications *without* changing your application programs. Let's see some other examples of using BatchPipes fittings.

Making Multiple Copies of Data

Suppose you need three copies of the same data to use with three different applications. You can attach a fitting to your existing job to accomplish this task. Figure 48 shows the sample JCL for a fitting that reads records from a pipe and writes three copies of those records to three jobs:

```
//PROCESSA JOB ...
//stepname EXEC PGM=pgmname,...
//pipeout DD DSN=some.pipe,DCB=(...),
//  SUBSYS=(BP01,'FITDD=PIPESPEC', 'CMT=--')
//PIPEOUT1 DD DSN=pipe1,SUBSYS=BP01,...
//PIPEOUT2 DD DSN=pipe2,SUBSYS=BP01,...
//PIPESPEC DD *
    BPR |          --Read data from the pipe
    QSAM PIPEOUT1 |  --Copy the data to the 1st pipe.
    QSAM PIPEOUT2 |  --Copy the data to the 2nd pipe.
    BPW          --Send the data along the 3rd pipe.
/*
```

Figure 48. Example 1: Writing One Output Stream to Three BatchPipes Pipes

Notice the data in the output streams of the first job becomes the data for the input streams of subsequent jobs.

A more efficient way to do the same thing is to replace BPR, QSAM ..., QSAM ..., and BPW with the single stage command BPCOPY ..., as Figure 49 shows:

```
//PROCESSA JOB ...
//stepname EXEC PGM=pgmname,...
//pipeout DD DSN=some.pipe,DCB=(...),
//  SUBSYS=(BP01,'FIT=BPCOPY PIPEOUT1 PIPEOUT2')
//PIPEOUT1 DD DSN=pipe1,SUBSYS=BP01,...
//PIPEOUT2 DD DSN=pipe2,SUBSYS=BP01,...
/*
```

Figure 49. Example 2: Writing One Output Stream to Three BatchPipes Pipes

Changing Generation Data Groups (GDGs)

You may have discovered that using BatchPipes with Generation Data Groups (GDG) is not easy to do. Once a single job is broken into multiple jobs, it is no longer possible to share the GDGs with jobs running in parallel. You can use a fitting similar to the example shown in “Making Multiple Copies of Data” on page 34 to solve this problem. Figure 50 shows how to use a writer fitting to retain and pipe a +1 GDG using level 0 as input:

```
//PROCESSA JOB ...
//stepname EXEC PGM=pgmname,...
//PROCAOUT DD DSN=pipe.plus1.data,DCB=(...),
//  SUBSYS=(BP01,
//  'FIT=BPR|QSAM newgen|BPW')
//PROCAIN DD DSN=processa.gdg(0),...
//newgen DD DSN=processa.gdg(+1),DISP=(NEW,CATLG),...
/*
```

Figure 50. Example of the Fitting that Copies and Pipes a New GDG Level

The example shows that PROCESSA reads in the current generation through input DD PROCAIN, creates a new generation via output to DD newgen in the fitting, and the new generation data is passed along the pipe to the next job.

You could use the BPCOPY stage command instead of BPW and BPR, as follows:

```
FIT=BPCOPY newgen
```

Counting Records

Using fittings, you have the capability of counting records as they flow through a pipe. Figure 51 shows JCL for counting records using a reader fitting in the BatchPipes pipeline:

```
//PROCESSB JOB ...
//stepname EXEC PGM=pgmname,...
//pipein DD DSN=some.pipe,DCB=(...),
//  SUBSYS=(BP01,
//  'FIT=(endchar ?)BPR|C: COUNT RECORDS|BPW ? C:|JOBLOG')
/*
```

Figure 51. Example of Counting Records Using a Fitting

This example shows a multistream pipeline. COUNT copies the input records to its primary output stream which is connected to the BPW stage command. BPW then writes the records back to the BatchPipes pipe. COUNT writes a single record containing the number of lines in the input records to its secondary output stream which is connected to the JOBLOG stage command. In this case, JOBLOG writes the count of the records in the joblog for PROCESSB.

Using Fittings Outside the BatchPipes Pipeline

You can also use a fitting with a job or jobstep that is not in a BatchPipes pipeline. For example, the fitting can read data from the BatchPipes pipe using BPR and then process the data. You can then pass the data to an application not defined to BatchPipes. In this case, the BPW stage command is omitted from the end of the fitting and the pipeline is called a **half-pipe fitting**. Because there is no partner for the pipe, you must use the OPENNOW subparameter on the SUBSYS parameter to complete the half-pipe fitting. BatchPipes normally prevents a process from opening a pipe until a partner opens the other side of the pipe.

Example of a Half-Pipe Fitting that Routes Records

You may have an existing job that sends data to a router job, which in turn, distributes data depending on a specific key within the record. The router job sends all records with key X to job X, all records with key Y to job Y, and all records with key Z to job Z. Figure 52 illustrates this routing process without using a fitting:

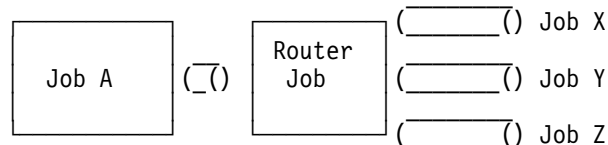


Figure 52. Example of Routing Records to Multiple Pipes

Fortunately, you can use a fitting in place of the router job. The fitting reads the data from the pipe and routes the data to the appropriate job according to specific key fields in the data.

The vertical-flow diagram for this pipeline would be as follows:

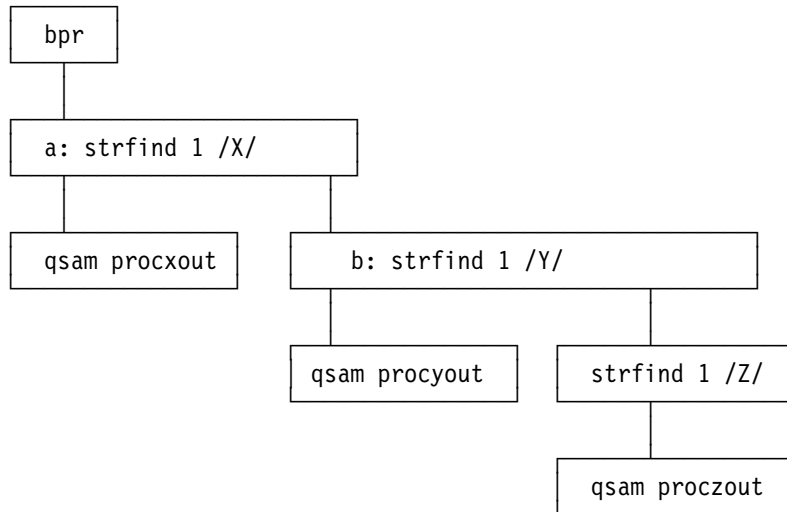


Figure 53 illustrates a BatchPipes pipeline that uses a half-pipe fitting instead of a router job:

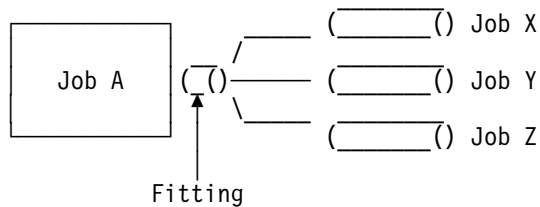


Figure 53. Example of Routing Records to Multiple Pipes Using a Half-Pipe Fitting

To change the JCL for Job A to include a half-pipe fitting that provides a routing function, use JCL similar to the one shown in Figure 54:

```

//PROCESSA JOB ...
//stepname EXEC PGM=pgmname,...
//pipeout DD DSN=some.pipe,DCB=(...),
//  SUBSYS=(BP01,'FITDD=PIPESPEC','CMT=--','OPENNOW)
//PROCXOUT DD DSN=processx.pipe,SUBSYS=BP01,...
//PROCYOUT DD DSN=processy.pipe,SUBSYS=BP01,...
//PROCZOUT DD DSN=processz.pipe,SUBSYS=BP01,...
//PIPESPEC DD *
(endchar ?)
  bpr|                                     --Get records from pipe
  a:strfind 1 /X/|QSAM PROCXOUT ?         --key 'X' to job X
  a:b:strfind 1 /Y/|QSAM PROCYOUT ?       --key 'Y' to job Y
  b:strfind 1 /Z/|QSAM PROCZOUT           --key 'Z' to job Z
/*
  
```

Figure 54. JCL that Routes Records to Multiple Pipes Using a Fitting

This half-pipe fitting does not need a BPW stage command to pass the data along the pipe. In this example, records are routed to new pipes depending on the key in column 1 of each record. Records that do not contain key X, Y, or Z are discarded.

The OPENNOW subparameter on the SUBSYS parameter is required for the half-pipe fitting to open for data flow. Without this parameter, BatchPipes waits until both reader and writer partners have opened before it builds pipe connections. OPENNOW opens the connection immediately.

Example of a Half-Pipe Fitting that Select Records

You might have jobs or jobsteps that select certain data for a subsequent job. For example, Process A might generate a master file, and Process B might be a sort program that selects records for another file. These processes may be two steps within a job, as Figure 55 shows:

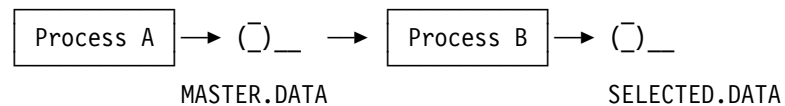


Figure 55. Example of Selecting Data Using Two Processes

This can be done using a half-pipe fitting on process A, as Figure 56 shows:

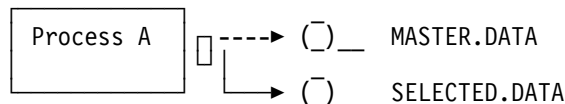


Figure 56. Example of Selecting Data Using a Half-Pipe Fitting

The JCL that defines this pipeline specification might be as follows:

```

//PROCESSA JOB ...
//stepname EXEC PGM=pgmname,...
//pipeout DD DSN=some.pipe,DCB=(...),
//  SUBSYS=(BP01,'FITDD=PIPESPEC','CMT=--',OPENNOW)
//MSTRFILE DD DSN=master.data...
//SELFILE DD DSN=selected.data...
//PIPESPEC DD *
  (endchar ?)
    BPR                                -- Get data from process
    |QSAM MSTRFILE                      -- Write it to the master.data file
    |STRFIND 1 /key/                    -- Select data for selected.data file
    |QSAM SELFILE                      -- Write the selected data
/*
  
```

Figure 57. JCL that Selects Records for Multiple Pipes Using a Half-Pipe Fitting

Chapter 4. Filter Stage Commands

This chapter contains General-Use Programming Interface and Associated Guidance Information.

BatchPipeWorks has many *filter stage commands*. A filter stage command reads data from its input stream, does some work with that data, and writes the results to its output stream. The output of a filter stage command is often quite different from its input. The CHANGE stage command, for example, writes a record to its output stream for every record it reads. LOCATE, on the other hand, reads all the records, but writes only those that match a search string. COUNT reads every record in its input stream, but writes only a single record to its output stream. By stringing filter stage commands together, you can transform raw data into useful results.

This chapter describes some commonly used filter stage commands that:

- Select records by content; see 39.
- Select records by position; see 48.
- Change contents of records; see 50.
- Duplicate records; see 73.
- Count records, words, and characters; see 73.
- Sort records; see 74.
- Buffer records; see 77.

Examples in this chapter apply to the TSO environment.

Selecting Records by Content

Several stage commands select records. That is, they read all records in the pipeline, but write only those that meet a certain criteria. This section describes filter stage commands that select records based on the content of the record. Those filter stage commands include:

LOCATE and NLOCATE
FIND and NFIND
TOLABEL and FRLABEL
UNIQUE.

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

Looking Everywhere in the Record (LOCATE, NLOCATE)

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

LOCATE Stage Command

LOCATE writes only the records containing a specific string. Suppose you have a data set named WINTER.THUGHTS that contains the following records:

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

Figure 58 shows an example of LOCATE. The < stage command reads the WINTER.THUGHTS data set and writes the records to its output stream. The next stage, LOCATE, examines the records, looking for the string winter. If the record contains the string, LOCATE writes it to its output stream. If it does not, LOCATE discards it. The next stage, TERMINAL, displays the records that LOCATE writes.

```
pipe < winter.thoughts | locate /winter/ | terminal  
I like winter.  
winter sun  
READY
```

Figure 58. LOCATE Stage Command Example: Locating Records Containing the String

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

What would happen if you looked for summer? Figure 59 shows the result.

```
pipe < winter.thoughts | locate /summer/ | terminal  
READY
```

Figure 59. LOCATE Stage Command Example: Not Locating Records Containing the String

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

A few words about the LITERAL stage command are necessary before we continue with the next example of LOCATE. Many examples in this chapter use LITERAL, which provides an easy way to put data in a pipeline so that you can see what a filter command does.

When LITERAL is first in a pipeline, it writes a record to its output stream. That record contains whatever you type as the LITERAL operand. When LITERAL is not first, it writes a record containing the operand, and then copies any records in its input stream to its output stream. This is a very important characteristic of LITERAL, and one that is easily forgotten. Remember: LITERAL writes before

copying. There is more about LITERAL in “Writing Literal Character Strings to a Pipeline (LITERAL)” on page 80.

Look again at the example of LOCATE in Figure 58 on page 40. Notice that slashes (/) are used in LOCATE to delimit the string you are searching for. You can use any character to delimit the string that is not itself in the string. To select records containing x/y for example, you might use commas as delimiters, as shown in Figure 60.

```
pipe literal z=x/y | locate ,x/y, | terminal
z=x/y
READY
```

Figure 60. LOCATE Stage Command Example: Using Delimiters

Using Input Ranges with LOCATE: To limit the search area inspected by LOCATE, enter an input range before the string operand. An input range defines a particular location of the record on which LOCATE operates. The first example in Figure 61 uses words as its input range, displaying records with the letter o in the second word. A word, by default, is defined as characters delimited by a blank. The second example defines the input range as columns, looking for o anywhere in columns 1 through 7.

```
pipe literal red shoe|literal orange hat|locate w2 /o/| terminal
red shoe
READY
pipe literal red shoe|literal orange hat|locate 1-7 /o/|terminal
orange hat
red shoe
READY
```

Figure 61. LOCATE Stage Command Example: Using an Input Range

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

See the *BatchPipeWorks Reference* for a complete description of input ranges under the LOCATE stage command.

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

```
pipe < novel.file | locate /mercurial/ | locate /saturnine/ | terminal
Mary was mercurial. Sam, saturnine. Yet on this day ten years ago they wed.
READY
```

Figure 62. LOCATE Stage Command Example: Locating Multiple Strings

Selecting Records By Length: You can select records that have some minimum length by using LOCATE without a string. Instead, just specify a column. Figure 63 selects records having a length of 2 or more bytes.

```
pipe literal a | locate 2 | terminal
READY
pipe literal ab | locate 2 | terminal
ab
READY
```

Figure 63. LOCATE Stage Command Example: Specifying Data Length

Be careful about trailing blanks when you use LITERAL. Notice that a stage separator immediately follows the strings in Figure 63—there are no trailing blanks. Try putting a blank between a and the stage separator. LITERAL puts any trailing blanks on records it writes. The blank is counted as a character, which means the entire string is 2 characters long. LOCATE will then select the string.

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

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

and robins to herald the coming of spring
```

Figure 64 reads in your data set, uses the STRIP stage command to change all blank lines to null (zero length) records, then uses LOCATE with no operands to remove all those records in the data set with a length of 0, and displays the remaining records as output to the terminal.

```
pipe < blnknstf.file | strip | locate | terminal
now is the time
for your heart to skip a beat
and robins to herald the coming of spring
READY
```

Figure 64. LOCATE Stage Command Example: Removing Blank Lines from Data Set

NLOCATE Stage Command

NLOCATE does the opposite of the LOCATE stage command. It writes all records that do not contain the string specified as the operand. Figure 65 shows an example of NLOCATE that reads the data set WINTER.THoughts. Suppose WINTER.THoughts contains the following records:

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

```
pipe < winter.thoughts | nlocate /winter/ | terminal  
Winter is cold.  
WINTER FUN  
READY
```

Figure 65. NLOCATE Stage Command Example: Locating Records

Except for the use of NLOCATE instead of LOCATE, the pipeline is identical to Figure 58 on page 40. As you would expect, the results are exactly opposite. The first and last lines are not echoed because the string `winter` precisely matches the NLOCATE argument. The second and third lines are echoed because `Winter` and `WINTER` do not match the NLOCATE argument.

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

You can select short records with NLOCATE. Figure 66 shows how to select all records having a length less than 5.

```
pipe literal too long| literal okay| nlocate 5 | terminal  
okay  
READY
```

Figure 66. NLOCATE Stage Command Example: Specifying Data Length

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

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

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

FIND and NFIND Stage Commands

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

```
pipe literal abc | literal def | find ab| terminal
abc
READY
```

Figure 67. FIND Stage Command Example: Finding Records Containing the String

If you do not care what characters certain columns hold, use a blank. Blanks in the argument string indicate columns whose contents are arbitrary, *not* columns containing only a blank character. (This differs from the way LOCATE works—to LOCATE, a blank is a blank.) Figure 68 shows an example that selects records beginning with the most, where any character can be between the and most.

```
pipe literal thermostat | literal the most interesting| find the most| terminal
the most interesting
thermostat
READY
```

Figure 68. FIND Stage Command Example: Using Arbitrary Characters in the String

How do you find a blank? Use an underscore (_) in the argument string to indicate that a column must have a blank. See how the results change in Figure 69.

```
pipe literal thermostat | literal the most interesting| find the_ most| terminal
the most interesting
READY
```

Figure 69. FIND Stage Command Example: Specifying a Blank in the String

Because an underscore indicates a required blank, you cannot search for underscores directly. Instead, use XLATE to transpose underscore and some other character before FIND or NFIND and restore them afterward. See “Translating Characters (XLATE)” on page 51 for a description of XLATE.

Be careful with blanks before the stage separator. If there are one or more blanks between the string you want to find and the stage separator, these positions must be present in the input record even if their contents are ignored. This is why Figure 70 on page 45 shows responses you might not expect. The input record is only one character long, but the argument to FIND (and NFIND) is two bytes.

```
pipe literal a | find a | terminal
READY
pipe literal a | nfind a | terminal
a
READY
```

Figure 70. FIND and NFOUND Stage Commands Examples

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

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

TOLABEL and FRLABEL Stage Commands

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

FRLABEL skips records until one is met with the required beginning; the record with the matching string and the remaining records are copied to the output. It doesn't matter what is in the rest of the data set, even if there is another matching record. Figure 71 shows examples of FRLABEL and TOLABEL selecting records from a fixed format data set called TEST.DATA.

```
pipe < test.data | terminal
Text before START
START
Text in the middle
END
Text after END.
READY
pipe < test.data | frlabel START | terminal
START
Text in the middle
END
Text after END.
READY
pipe < test.data | tolabel END | terminal
Text before START
START
Text in the middle
READY
```

Figure 71. FRLABEL and TOLABEL Stage Commands Examples

FRLABEL and TOLABEL are often used in combination. For instance, Figure 72 shows how to select records from the label START and to (but not including) the label END.

```
pipe < test.data | frlabel START | tolabel END | terminal
START
Text in the middle
READY
```

Figure 72. FRLABEL and TOLABEL Stage Commands Example

Note the blank after START and END. In this example, the blank after each argument string will select the same records as it would if each argument string did not have a trailing blank. Your results could vary from what is shown in Figure 72 if the pipeline reads in a variable format data set rather than a fixed format data set and the lines in the data set did not have trailing blanks. With the trailing blanks, the pipeline would stop without any records being matched.

Looking at the End of a Record

There is no stage command to select records that end with some particular string, but several filter stage commands can be combined to do it. The trick is to copy the end of each record to the beginning of each record. For that, you need to use the SPECS stage command, which has not yet been discussed. Refer to Figure 134 on page 72 to see an example of looking at the end of a record.

Discarding Duplicate Records (UNIQUE)

Two stage commands remove duplicate records: SORT (with the UNIQUE operand specified) and UNIQUE. They produce different results. Use SORT UNIQUE when you have unsorted records and you want to discard duplicates while sorting those records. SORT UNIQUE is described in “Discarding Duplicates When Sorting” on page 76.

The UNIQUE stage command removes duplicate records that are adjacent to each other. UNIQUE compares each input record with the next one. With no operands, it discards a record that has the same contents as the following one. That is, series of identical records are replaced with the last occurrence of the record.

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

Suppose you have a data set named WEATHER.INFO that contains the following records:

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

The records are not sorted, so not all duplicates are adjacent. UNIQUE will not remove all duplicates, as Figure 73 shows:

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

Figure 73. UNIQUE Stage Command Example

Discarding Unique Records (UNIQUE MULTIPLE)

UNIQUE (with the MULTIPLE operand specified) discards records that do not have adjacent duplicates. It keeps records that occur at least twice together. The REXX exec named DUPMEM, shown in Figure 74, uses UNIQUE with the MULTIPLE operand. It creates a list of members that exist in two partitioned data sets.

```
/***** REXX *****/
/* PREFIX.PIPE.EXEC(DUPMEM) */
/* -- Find the same member name in two PDS's and report */
/* the first of each */
/*****/
arg dsn1 dsn2 .
if dsn2='' then exit 24 /* Incomplete parameters? */

'PIPE',
  '| listispf" dsn1, /* Execute LISTISPF */
  '| append listispf" dsn2, /* Execute second LISTISPF */
  '| sort ', /* Put duplicates together */
  '| unique 1.8 multiple', /* Select only duplicates */
  '| unique 1.8 first', /* Take only the first */
  '| terminal' /* Display them */
exit
```

Figure 74. UNIQUE MULTIPLE Stage Command Example (DUPMEM EXEC Contents)

In DUPMEM, two LISTISPF commands are executed. The first LISTISPF command executed reads the directory of a partitioned data set (specified by the variable *dsn1*) and writes the names of each member of the partitioned data set to its primary output stream. The primary output stream of the first LISTISPF

command is connected to the primary input stream of the APPEND stage command. The APPEND stage command copies the records from its primary input stream (which contain the names of each member of the partitioned data set identified by *dsn1*) to its primary output stream, and then invokes the second LISTISPF command in the exec. Since the second LISTISPF command is an operand of the APPEND stage, LISTISPF reads the directory of a partitioned data set (specified for the variable *dsn2*) and appends the names of each member of the partitioned data set to the primary output stream of APPEND. The resulting primary output stream of APPEND contains the names of each member of *dsn1* followed by the names of each member of *dsn2*. The results are sorted to put duplicates together. Then UNIQUE MULTIPLE is executed. A column range of 1.8 is specified on UNIQUE MULTIPLE to restrict the test to the member name. (The column range 1.8 specifies a field that begins in column 1 and is 8 characters long.) Now the pipeline contains records having duplicate data set names.

A second UNIQUE eliminates duplicate lines. The result, displayed by TERMINAL, is one line for each member on both partitioned data sets. Figure 75 shows a sample run.

```
dupmem prefix.pipe.* prefix2.mvspipes.*
NONVSAM ----- PREFIX      PIPE      EXEC
NONVSAM ----- PREFIX      PIPE      LOAD
NONVSAM ----- PREFIX      PIPE      REXX
READY
```

Figure 75. UNIQUE MULTIPLE Stage Command Example (DUPMEM EXEC Results)

Selecting Records by Position (TAKE, DROP)

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

```
harp
lyre
lute
viola
violin
bouzouki
oud
guitar
mandolin
harpsichord
```

The TAKE stage command picks the first or last *n* records (where *n* is zero, or a positive number, or * for all) for output to the pipeline. By default, TAKE selects the first records. Figure 76 demonstrates taking the first 3 records.

```
pipe < string.list | take 3 | terminal
harp
lyre
lute
READY
```

Figure 76. TAKE Stage Command Example

To take the last records, specify the LAST operand on TAKE. Figure 77 shows how to take the last 4 records.

```
pipe < string.list | take last 4 | terminal
oud
guitar
mandolin
harpsichord
READY
```

Figure 77. TAKE LAST Stage Command Example

The DROP stage command is the converse of TAKE. It lets you discard the first or last *n* records (where *n* is zero, or a positive number, or * for all). Figure 78, shown next, drops the first two records from the data set STRING.LIST.

```
pipe < string.list | drop 2 | terminal
lute
viola
violin
bouzouki
oud
guitar
mandolin
harpsichord
READY
```

Figure 78. DROP Stage Command Example

To drop the last records in the pipeline, use the LAST operand. Figure 79, shown next, shows how to drop the last three records of the STRING.LIST data set.

```
pipe < string.list | drop last 3 | terminal
harp
lyre
lute
viola
violin
bouzouki
oud
READY
```

Figure 79. DROP LAST Stage Command Example

By combining TAKE and DROP, you can get records in the middle. Figure 80 shows how to get the fifth and sixth lines of the STRING.LIST data set.

```
pipe < string.list | drop 4 | take 2 | terminal
violin
bouzouki
READY
```

Figure 80. TAKE and DROP Stage Command Example

Changing Contents of Records

BatchPipeWorks includes stage commands that change the contents of records. You can:

- Translate characters to uppercase or to lowercase
- Substitute one character for another
- Split and join records
- Expand and truncate records
- Remove leading or trailing characters from records
- Replace a string with another string
- Rearrange the contents of a record.

REXX programmers will notice that many of these stage commands have counterparts in the REXX language. The difference is that these stage commands work on all data flowing through the stage, while REXX requires extra coding to operate beyond a single expression.

Translating Characters (XLATE)

XLATE translates data passing through the pipeline on a character-by-character basis. You can translate characters to lowercase or to uppercase. You can also replace all occurrences of one character with another. Figure 81 shows how to translate characters to uppercase:

```
pipe literal I'm NOT mad about it. | xlate upper | terminal
I'M NOT MAD ABOUT IT.
READY
```

Figure 81. XLATE Stage Command Example: Translating a String to Uppercase

Translating a string to lowercase is just as easy, as Figure 82 shows.

```
pipe literal I'm NOT mad about it. | xlate lower | terminal
i'm not mad about it.
READY
```

Figure 82. XLATE Stage Command Example: Translating a String to Lowercase

To limit the translation to certain columns, specify a column range after XLATE, as Figure 83 shows:

```
pipe literal Play piano, not forte. | xlate 17.5 upper | terminal
Play piano, not FORTE.
READY
```

Figure 83. XLATE Stage Command Example: Using a Column Range

The 17.5 in Figure 83 defines a range of 5 columns starting on column 17. (Specifying 17-21 would yield the same result.)

With the UPPER and LOWER operands, many characters have been translated with a single XLATE stage command. It is also possible to translate individual characters. Instead of specifying UPPER and LOWER operands, specify pairs of characters after XLATE. The example in Figure 84 changes all e's to X's.

```
pipe literal Extra eggplants free. | xlate e X | terminal
Extra Xggplants frXX.
READY
```

Figure 84. XLATE Stage Command Example: Translating Individual Characters

Notice that the capital E in Extra was not changed. XLATE is case sensitive. To change both small and capital E's to X's, specify two pairs of characters after XLATE, as Figure 85 shows:

```
pipe literal Extra eggplants free. | xlate e X E X | terminal
Xxtra Xggplants frXX.
READY
```

Figure 85. XLATE Stage Command Example: Translating Multiple Characters

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

When you are specifying numbers, you may also need to specify a column range. Otherwise, XLATE might mistake your first pair of numbers for a range of characters. Figure 86 shows a column range (1-*) on the XLATE stage command, followed by a number. The string 1-* indicates a range from column 1 through the last column. This means that the complete record should be translated. The string 1-* is specified to ensure that BatchPipeWorks parses the stage command properly.

```
pipe literal 370 | xlate 1-* 3 E 7 S 0 A | terminal
ESA
READY
```

Figure 86. XLATE Stage Command Example: Specifying a Range

Instead of specifying characters directly, you can use hexadecimal values. This is useful when you want to specify characters that you cannot type on your keyboard. It is also useful when you want to specify a blank, which is the hexadecimal value X'40'. Specify hexadecimal values by typing the two characters that make up the byte. Figure 87 changes all parentheses to blanks.

```
pipe literal x(x)x | xlate 1-* ( 40 ) 40 | terminal
x x x
READY
```

Figure 87. XLATE Stage Command Example: Specifying Hexadecimal Values

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

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

Figure 88 on page 53 uses a character range to do the same translation. The second PIPE command in Figure 88 on page 53 also shows a character range. It

removes punctuation from a record by translating the punctuation to blanks. Hexadecimal values are used in the second example.

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

Figure 88. XLATE Stage Command Example: Using Ranges of Characters

You can specify ranges for both the input and the output. Figure 89 translates characters 1 through 9 into the corresponding letters A through I.

```
pipe literal 123 12389 | xlate 1-* 1-9 A-I | terminal
ABC ABCHI
READY
```

Figure 89. XLATE Stage Command Example: Using Ranges for Input and Output

To translate all but a few of the characters in a range, specify the range of characters and also the characters you wish to omit. Before XLATE does any translation, it determines whether any character is specified more than once in the operands. XLATE uses only the last occurrence when translating the data. Figure 90 shows an example. The characters from c to g are translated to equal signs, except for the character e. The character e is translated to itself.

```
pipe literal abcdefghi | xlate c-g = e e | terminal
ab==e==hi
READY
```

Figure 90. XLATE Stage Command Example: Overriding a Character Range

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

Splitting and Joining (SPLIT, JOIN)

The SPLIT and JOIN stage commands split and join records.

SPLIT Stage Command

By default, SPLIT creates an output record for each blank-delimited word in its input records, as Figure 91 shows:

```
pipe literal A phrase with five words | split | terminal
A
phrase
with
five
words
READY
```

Figure 91. SPLIT Stage Command Example

SPLIT has operands to define other kinds of splitting. See the *BatchPipeWorks Reference* for a complete description of SPLIT.

JOIN Stage Command

JOIN creates a single record from one or more input records. By default, JOIN puts together pairs of input records, as Figure 92 shows:

```
pipe literal A phrase with five words | split | join | terminal
Aphrase
withfive
words
READY
```

Figure 92. JOIN Stage Command Example: Joining Pairs of Records

LITERAL puts a record into the pipeline. SPLIT puts each of the five words on a separate record. Then JOIN combines pairs of records. Because there is an odd number of input records, JOIN puts words by itself in an output record. Notice that JOIN puts records together without intervening blanks. To put a blank between the joined records, specify / / as shown in Figure 93.

```
pipe literal A phrase with five words | split | join / / | terminal
A phrase
with five
words
READY
```

Figure 93. JOIN Stage Command Example: Putting a Space between Joined Records

Between the delimiters (/) you can specify a string you want to insert between the joined records. In Figure 94 on page 55, dashes (--) are inserted.

```
pipe literal A phrase with five words | split | join /--/ | terminal
A--phrase
with--five
words
READY
```

Figure 94. JOIN Stage Command Example: Putting Strings between Joined Records

You can use any character not in the string itself as the delimiter. Figure 95 shows question marks (?) used as delimiters.

```
pipe literal A phrase with five words | split | join ?--? | terminal
A--phrase
with--five
words
READY
```

Figure 95. JOIN Stage Command Example: Using String Delimiters

You can also use JOIN to join more than two records by specifying the number of records to be added to the first record. Put the number after the JOIN keyword and before any delimited string. If you want to join every three records, specify 2 *not* 3. Figure 96 joins every three input records into one output record.

```
pipe literal A phrase with five words | split | join 2 / / | terminal
A phrase with
five words
READY
```

Figure 96. JOIN Stage Command Example: Joining More than Two Input Records

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

To put all input records in a single output record, specify an asterisk (*) instead of a number. Figure 97 splits apart the record written by LITERAL and puts it back together again.

```
pipe literal Break up and make up | split | join * / / | terminal
Break up and make up
READY
```

Figure 97. JOIN Stage Command Example: Joining All Pipeline Records

Padding and Chopping (PAD, CHOP)

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

CHOP Stage Command

CHOP truncates each record after a column. Specify the column number after CHOP, as Figure 98 shows:

```
pipe literal She loves me; she loves me not. | chop 12 | terminal
She loves me
READY
```

Figure 98. CHOP Stage Command Example

PAD Stage Command

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

Figure 99 chops the record at column 12 and then pads the record to column 20 with question marks (?). The pad character must follow the column number.

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

Figure 99. PAD Stage Command Example

In its simplest form, PAD *n* adds *n* blanks to the right side of the string. To add them to the left, type *left* after *pad*, as shown in Figure 100.

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

Figure 100. PAD Stage Command Example: Padding on the Left

Figure 101 on page 57 combines CHOP and PAD to create records with fixed lengths (here with a length of 10). The TEST.DATA data set being read into the pipeline is a variable format data set.

```

pipe < test.data | terminal
Short
This record will be truncated
READY
pipe < test.data | pad 10 ? | chop 10 | terminal
Short????
This recor
READY

```

Figure 101. PAD and CHOP Stage Commands Example

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

Combine PAD and CHOP to create a fixed format data set. See Chapter 5, “Device Driver Stage Commands” on page 79 for more about creating fixed format data sets.

Removing Leading or Trailing Characters (STRIP)

Use the STRIP stage command to remove blanks from the beginning and the end of records. Figure 102 shows a simple STRIP example.

```

pipe literal Hello? | strip | terminal
Hello?
READY

```

Figure 102. STRIP Stage Command Example

STRIP can also remove only leading or trailing blanks, as shown in Figure 103.

```

pipe literal c| literal b | literal a| join *| terminal
a b c
READY
pipe literal c| literal b | literal a| strip leading| join *| terminal
ab c
READY
pipe literal c| literal b | literal a| strip trailing| join *| terminal
a bc
READY
pipe literal c| literal b | literal a| strip| join *| terminal
abc
Ready:

```

Figure 103. STRIP Stage Command Example: Stripping Leading or Trailing Characters

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

```
pipe literal 0000120| strip leading string /0/| terminal
120
READY
pipe literal bread meat bread| strip string /bread/| terminal
meat
READY
pipe literal bread meat bread lettuce bread| strip string /bread/| terminal
meat bread lettuce
READY
```

Figure 104. STRIP Stage Command Example: Stripping Nonblank Characters

Changing and Rearranging Contents (CHANGE, SPECS)

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

CHANGE Stage Command

The CHANGE stage command replaces one character or group of characters with another. For example, suppose you want to change the name John to Martin in a data set named STORY.SCRIPT. You might enter:

```
pipe < story.script | change /John/Martin/ | terminal
```

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

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

John Smith and Bob Johnson ate johnnycake.

to:

Martin Smith and Bob Martinson ate johnnycake.

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

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

You can also limit CHANGE by specifying a range of columns. In Figure 105 on page 59, CHANGE searches for John only in the first four columns of each record. Consequently, Johnson is not changed.

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

Figure 105. CHANGE Stage Command Example: Using Column Ranges

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

Figure 106 shows two column ranges. CHANGE looks for John in columns 1 through 4 and in columns 20 through 23. Use parentheses as shown when entering more than one column range. This time Johnson is changed because it begins in column 20:

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

Figure 106. CHANGE Stage Command Example: Using Several Column Ranges

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

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

```
pipe < yourid.data | change /MIKE /BARBARA / | terminal
```

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

SPECS Stage Command

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

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

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

Here is the SPECS stage command to do it:

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

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

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

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

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

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

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

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

input output

Input Operands—Columns Numbers and Column Ranges: You have already seen isolated SPECS stage commands that put pieces of input records in the output records. Now let's look at a complete PIPE command in Figure 107.

```
pipe literal abcdefghij | specs 3-5 1 | terminal
cde
READY
```

Figure 107. SPECS Stage Command Example: Using Numbers to Identify Input

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

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

To put additional data from each input record into each output record, use additional groups of operands, as shown in Figure 108.

```
pipe literal abcdefghij | specs 3-5 1 1 5 | terminal
cde a
READY
```

Figure 108. SPECS Stage Command Example: Using More than One Group of Numbers

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

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

You can use any of the usual formats for specifying column ranges. Figure 109 uses the operand 4.5 to identify a column range of 5 characters starting with column 4. It also uses the operand 4-6 to identify columns 4 through 6.

```
pipe literal 000Television | specs 4.5 5 4-6 1 | terminal
Tel Telev
READY
```

Figure 109. SPECS Stage Command Example: Using Various Column Ranges

Figure 109 also shows that you do not need to build the output record in any particular order. First SPECS puts data starting in column 5, then SPECS puts data starting in column 1. There isn't anything wrong with doing this. It is also valid to refer to columns of the input record more than once. Notice that SPECS refers twice to columns 4, 5, and 6.

An asterisk (*) in the second position of a column range means end of record, just as it does on other stage commands. The example in Figure 110 uses an asterisk in the column range. It removes characters from columns 1 through 8 of the input record. All characters from column 9 to the end of the input record are copied to the output record starting at column 1.

```
pipe literal 12345678An input record | specs 9-* 1 | terminal
An input record
READY
```

Figure 110. SPECS Stage Command Example: Specifying the End of the Record

Using numbers to identify columns or column ranges on input records is useful when the data is structured. In inventory records, for example, all part numbers are likely to occur within a specific column range. However, not all data is structured in

this way. For processing other forms of data, the WORDS operand is often more useful.

Input Operands—WORDS: The WORDS operand of SPECS lets you select data from input records by word number (or word range) rather than column number (or column range). To SPECS, a *word* is any delimited string of characters. A word cannot be null. The default delimiter is a blank. To change the delimiter character use the WORDSEPARATOR operand. (WORDSEP and WS are abbreviations for this operand).

Figure 111 shows two examples of WORDS. In both examples, the third word of the input record is put in the output record starting at column 1. The TERMINAL stages display the record. Note that each example, because it is not specified otherwise, uses the default word delimiter of blank.

```
pipe literal See Joe compute. | specs words 3 1 | terminal
compute.
READY
pipe literal See Joe compute. | specs w3 1 | terminal
compute.
READY
```

Figure 111. SPECS Stage Command Example: Using the WORDS Operand

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

You can also specify a range of words. Indicate word ranges in the same way that you have been indicating column ranges. Figure 112 shows two examples.

```
pipe literal fish dog cat horse | specs word2-3 1 | terminal
dog cat
READY
pipe literal fish dog cat horse | specs word 2-* 1 | terminal
dog cat horse
READY
```

Figure 112. SPECS Stage Command Example: Specifying WORD Ranges

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

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

Figure 113 shows an example using FIELDSEPARATOR and FIELDS. In this example, the input data contains two fields separated by a dash (-): The TERMINAL stage displays the record.

```
pipe literal torn-asunder | specs fieldsep - f1 1 f2 10 | terminal
torn      asunder
READY
```

Figure 113. SPECS Stage Command Example: Using the FIELDS Operand

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

You can also specify a range of fields. Indicate field ranges in the same way that you have been indicating column or word ranges. Figure 114 shows two examples.

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

Figure 114. SPECS Stage Command Example: Specifying FIELDS Ranges

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

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

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

To specify a literal on SPECS, use a delimited string as the input operand instead of a column or word reference to the input record. Delimit the string the same way you delimit strings on other stage commands (such as LOCATE). Use any character that is not in the string itself and that does not have a special meaning (such as a stage separator). SPECS puts the string on every output record. For example, Figure 115 puts the words Space Captain in the output record.

```
pipe literal Bob | specs /Space Captain/ 1 1-* 15 | terminal
Space Captain Bob
READY
```

Figure 115. SPECS Stage Command Example: Specifying A Literal String

As before, read the SPECS arguments in groups:

specs	/Space Captain/	1	1-*	15
	Group 1		Group 2	

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

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

```
Bob
Mark
Joe
Sue
```

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

```
pipe < space.cadets | specs ?Space Captain? 1 1-* 15 | terminal
Space Captain Bob
Space Captain Mark
Space Captain Joe
Space Captain Sue
READY
```

You are not required to use any of the input record in the output record. In Figure 116 on page 65, for example, SPECS reads two records from its input

stream. Data from these input records is not copied to the output records. Instead, only the string `fruit salad` is put in each output record.

```
pipe literal banana | literal melon | specs /fruit salad/ 1 | terminal
fruit salad
fruit salad
READY
```

Figure 116. SPECS Stage Command Example: Not Using Data from the Input Records

Usually you will use literal strings that consist of character data. Occasionally you might need to use hexadecimal characters (perhaps because you cannot type the desired characters on your terminal). To do it, introduce the hexadecimal string as an operand on SPECS, prefix the string of hexadecimal digits with the character `x` or `h`, make sure you have an even number of digits, and end the string with a blank. Because a blank marks the end of the hexadecimal string, there can be no blanks in the hexadecimal string itself.

Figure 117 shows how to put a vertical bar (`|`) in the output record without it being interpreted as a stage separator when the pipeline is processed. The contents of the input record are put after the vertical bar.

```
pipe literal a line | specs x4f 1 1-* 2 | terminal
|a line
READY
```

Figure 117. SPECS Stage Command Example: Using a Hexadecimal Literal

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

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

Figure 118 shows an example in which the 10-character record number is positioned at column 1 of the output record (`RECNO 1`). The record number is padded on the left with 9 blanks. The input record is put immediately after the record number (`1-* 11`).

```
pipe literal line2|literal line1|specs recno 1 1-* 11 | terminal
      1line1
      2line2
READY
```

Figure 118. SPECS Stage Command Example: Using the `RECNO` Operand

Output Operands—Columns and Column Ranges: The next few sections describe various operands you can use to position data in output records. You have already seen how to use column numbers. Figure 119 shows another example.

```
pipe literal Start me at column 15 | specs 1-* 15 | terminal
      Start me at column 15
READY
```

Figure 119. SPECS Stage Command Example: Using Column Numbers for Output

Figure 120 shows what happens when you miscalculate. In this case, Jones is copied to columns 1 through 5, and Raymond is copied to column 5. Raymond overlays the s in Jones. Because you might want to do something like this intentionally, no error message is produced.

```
pipe literal Raymond Jones | specs 9-13 1 1-8 5 | terminal
JoneRaymond
READY
```

Figure 120. SPECS Stage Command Example: Overlaying Data

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

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

In Figure 121, the operands `recno 1` put the record number at the beginning of the record. The operands `1-* next` put the contents of the input record in the output record at the next available space. In this case, the next available space is immediately after the record number.

```
pipe literal My record | specs recno 1 1-* next | terminal
      1My record
READY
```

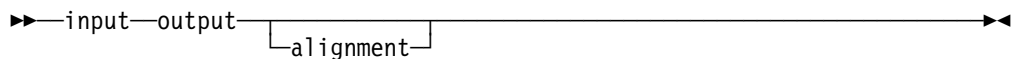
Figure 121. SPECS Stage Command Example: Using the NEXT Operand

Output Operands—NEXTWORD: NEXTWORD is similar to NEXT. The difference is that NEXTWORD puts a blank before the input data. Unless the output record contains no data, NEXTWORD copies the input in column 1, as Figure 122 shows:

```
pipe literal flip flop | specs word2 nextword word1 nextword | terminal
flip flop
READY
```

Figure 122. SPECS Stage Command Example: Using the NEXTWORD Operand

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



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

```
pipe literal My Summer Vacation | specs 1-* 1-80 center | terminal
My Summer Vacation
READY
```

Figure 123. SPECS Stage Command Example: Aligning Data

When aligning data, SPECS strips the input field of leading and trailing blanks and aligns what remains of the input field in the output field, truncated or padded as necessary. Figure 124 shows how to align lines on the right. The output field is from column 1 for 50 columns.

```
pipe literal shorter|literal a long line|specs 1-* 1.50 right| terminal
a long line
shorter
READY
```

Figure 124. SPECS Stage Command Example: Aligning Data to the Right

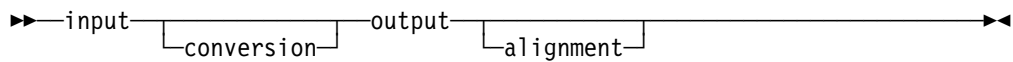
By default, data is aligned to the left when a column range is specified for the output operand. Figure 125 on page 68 shows an example in which the LEFT operand is specified.

```
pipe literal Aligned left | specs 1-* 1.50 left | terminal
Aligned left
READY
```

Figure 125. SPECS Stage Command Example: Aligning Data to the Left

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

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



The input and output operands must always be specified. The conversion and alignment operands are optional. If desired, a conversion operand and an alignment operand can be specified for a single data item.

Figure 126 shows the first eight bytes of a packed data set in hexadecimal. Two output data items are specified. The group of operands for the first item is 1.4 c2x 1. The group of operands for the second item is 5.4 c2x 10. The operand c2x is the conversion operand for both items.

```
pipe < proc.copy | take 1 | specs 1.4 c2x 1 5.4 c2x 10 | terminal
00140C6F 00000050
READY
```

Figure 126. SPECS Stage Command Example: Converting Data

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

Figure 127 on page 69 shows several other conversions. The conversion operand C2B converts data from character to binary. B2C reverses the conversion. X2C converts from hexadecimal to character—it requires an even number of hexadecimal characters.

```

pipe literal 911 | specs 1-* c2b 1 | terminal
111110011111000111110001
READY
pipe literal 911 | specs 1-* c2b 1 | specs 1-* b2c 1 | terminal
911
READY
pipe literal F9F1F1 | specs 1-* x2c 1 | terminal
911
READY

```

Figure 127. SPECS Stage Command Example: Additional Conversions

The *BatchPipeWorks Reference* contains more information about using the many conversion operands.

Advanced Uses of SPECS

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

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

```

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

```

Suppose for each group of records you want to write one output record that contains the state followed by the name. To do it, you would use the READ operand of SPECS to get the name from the first record, skip the second record, and get the state from the third. SPECS with the READ operand reads the next record from the input stream without writing a record to the output stream. Look at the example in Figure 128.

```

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

```

Figure 128. SPECS Stage Command Example: Using the READ Operand

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

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

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

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

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

Figure 129 shows an example that produces two output records for every input record read.

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

Figure 129. SPECS Stage Command Example: Using the WRITE Operand

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

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

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

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

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

Instead, you need to refer to the last column by giving some number relative to the end of the record. To do so, use a negative column number. When negative column numbers are used in a column range, the column numbers must be separated by a semicolon (;). The usual hyphen (-) or period (.) cannot be used. The example in Figure 130 shows a SPECS stage command that displays the last column of each record.

```
pipe literal ABCDE| literal abc| specs -1;-1 1 | terminal
c
E
READY
```

Figure 130. SPECS Stage Command Example 1: Using Negative Relative Column Numbers

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

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

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

Figure 131 shows a similar example. The third column relative to the end of the input record is put in the output record at column 5.

```
pipe literal ABCDE| literal abc| specs -3;-3 5 | terminal
a
C
READY
```

Figure 131. SPECS Stage Command Example 2: Using Negative Relative Column Numbers

Suppose you want to see the last two columns. The input range should then be -2;-1. Figure 132 shows the result.

```
pipe literal ABCDE| literal abc| specs -2;-1 1 | terminal
bc
DE
READY
```

Figure 132. SPECS Stage Command Example 3: Using Negative Relative Column Numbers

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

Figure 133 shows that the entire record is returned when you use a negative column number that is too high. The same thing happens when the column number is too high in a positive column range.

```
pipe literal ABCDE| literal abc| specs -600;-1 1 | terminal
abc
ABCDE
READY
```

Figure 133. SPECS Stage Command Example: Specifying Range Beyond the Input Record

Figure 134 shows you how to filter records by looking at the ends of the records. The example finds all records ending in x. It is assumed that the data set INPUT.FILE contains variable-length records.

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

Figure 134. SPECS Stage Command Example: Looking at the End of a Record

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

Duplicating Records (DUPLICATE)

DUPLICATE makes copies of input records. It reads an input record and writes that record one or more times to its output stream. For DUPLICATE's operand, specify the number of additional copies desired. Figure 135 makes two additional copies of each input record.

```
pipe literal Are we almost there? | literal Dad | duplicate 2 | terminal
Dad
Dad
Dad
Are we almost there?
Are we almost there?
Are we almost there?
READY
```

Figure 135. DUPLICATE Stage Command Example

Counting Characters, Words, and Records (COUNT)

The COUNT stage command counts characters (bytes), words, or records in its input stream. It writes a single record containing the count to its output. Figure 136 shows a dialog in which three PIPE commands are entered. The first counts the number of bytes in the LOG.MISC data set, the second counts the number of words, and the third counts the number of lines.

```
pipe < log.misc | count bytes | terminal
2456
READY
pipe < log.misc | count words | terminal
347
READY
pipe < log.misc | count lines | terminal
67
READY
```

Figure 136. COUNT Stage Command Examples

When counting words, the COUNT stage command considers any blank-delimited string to be a word. To COUNT, the TEST.DATA data set in Figure 137 contains 10 words.

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

Figure 137. COUNT Stage Command Example: Counting Words

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

COUNT lets you count several things at a time, as shown in Figure 138. The operand CHARS is a synonym for BYTES.

```
pipe < log.misc | count chars words lines | terminal
2456 347 67
READY
```

Figure 138. COUNT Stage Command Example: Counting Several Items

When you specify more than one item, COUNT always returns results in this order: characters, words, lines. The order in which you specify the operands does not change the order of the results, as Figure 139 shows:

```
pipe < log.misc | count lines chars | terminal
2456 67
READY
```

Figure 139. The Order of COUNT Results

Sorting Records (SORT)

The SORT stage command orders pipeline records. SORT *buffers* the pipeline records; that is, it reads all its input records before writing output records. The sorting is done in storage. SORT gives a message and a nonzero return code if the input is too large to fit in storage. For example, the following pipeline sorts the names of all data sets in your catalog. The names of only the first 10 data sets are displayed.

```
pipe tso listcat | change ./ x05 | untab -9 | sort | take 10 | terminal
IN CATALOG:ICFCAT HSML04
PREFIX  BROADCAST
PREFIX  MVSPICES CNTL
PREFIX  INPUT    DATA
PREFIX  INPUT    TESTER
PREFIX  INPUT2   DATA
PREFIX  ISPF     PROFILE
PREFIX  LAST     ACCESS
PREFIX  LOG      MISC
PREFIX  PIPE     EXEC
READY
```

Figure 140. SORT Stage Command Example

By default, records are sorted in ascending order. (You can use the ASCENDING operand if you wish.) To sort records in descending order, specify the DESCENDING operand, which is abbreviated to DESC in Figure 141.

```
pipe tso listc | change ./ x05 | untab -9 | sort desc | take 10 | terminal
PREFIX  TESTER
PREFIX  TEST      EXEC
PREFIX  TEMP3
PREFIX  TEMP2
PREFIX  TEMP1
PREFIX  TEMP
PREFIX  TEE        OUTPUT3
PREFIX  TEE        OUTPUT2
PREFIX  TEE        INPUT1
PREFIX  SPFLOG1    LIST
READY
```

Figure 141. SORT DESCENDING Stage Command Example

Using Column Ranges When Sorting

You can also specify column ranges to be used in the sort. The records are sorted by the contents of the column ranges. For example, Figure 142 sorts the names of the data sets with your PREFIX by data set type, and displays the first ten data set names. (The data set types in a LISTCAT response start at column 10 and can be up to 8 characters long.)

```
pipe tso listc | change ./ x05 | untab -9 | sort 19-27 | take 10 | terminal
PREFIX  BROADCAST
PREFIX  TEMP
PREFIX  TEMP1
PREFIX  TEMP2
PREFIX  TEMP3
PREFIX  TESTER
PREFIX  LAST      ACCESS
PREFIX  MVSPICES  CNTL
PREFIX  INPUT     DATA
PREFIX  INPUT2    DATA
READY
```

Figure 142. SORT Stage Command Example: Using a Column Range

To sort in descending order when a column range is used, use DESCENDING after the column range, as Figure 143 shows:

```
pipe tso listc | change ./ x05 | untab -9 | sort 19-27 descending | take 4 | terminal
PREFIX INPUT TESTER
PREFIX PIPE REXX1
PREFIX PIPE REXXV
PREFIX PIPE REXX
READY
```

Figure 143. SORT DESCENDING Stage Command Example: Using a Column Range

Discarding Duplicates When Sorting

Use SORT UNIQUE to discard duplicate records during a sort. Figure 144 shows how to display a list of unique words in a data set.

```
pipe < input.file | split | sort unique | terminal
```

Figure 144. SORT UNIQUE Stage Command Example

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

Counting and Discarding Duplicates While Sorting

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

Figure 145 on page 77 shows two PIPE commands. The first command shows the contents of the data set that is sorted in the second PIPE command.

```
pipe < sample.data | terminal
one
two
two
three
three
three
four
four
READY
pipe < sample.data | sort count | specs 1-10 1 11-* nextword | terminal
    2 four
    1 one
    3 three
    2 two
READY
```

Figure 145. SORT COUNT Stage Command Example: Counting and Discarding Duplicates

Buffering Records (BUFFER)

The SORT stage command buffers records in the course of its processing. At other times, you might want to buffer records yourself, which requires you to use the BUFFER stage command. BUFFER holds all the records until it has read the last input record. Then BUFFER writes the records to the next stage. Use BUFFER any time the records must be delayed until all input is read.

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

In the PIPE command in Figure 146, the TERMINAL stage reads records entered at the terminal. The BUFFER stage holds all the records until it reads the last input record. The last input record is the final record the user types before pressing enter twice. Once BUFFER has read all the input records, it writes the records to the STACK stage.

```
pipe terminal | buffer | stack
```

Figure 146. BUFFER Stage Command Example: Stacking Terminal Input Lines

If you remove the BUFFER stage from the PIPE command in Figure 146, TERMINAL reads an input record and writes it immediately to STACK, which places the record on the data stack. Because TERMINAL reads not only from the terminal but also from the data stack, the command loops.

Chapter 5. Device Driver Stage Commands

This chapter contains General-Use Programming Interface and Associated Guidance Information.

Device drivers are stage commands that move data between your pipeline and a target or source outside of the pipeline. Most device drivers can be anywhere in the pipeline. Some must be first, however, and others cannot be first. Be careful when using device drivers that can be placed anywhere in a pipeline, such as STEM and VAR. When first in a pipeline, these device drivers read from the system resource. When used anywhere else in the pipeline, they write to the system resource, often replacing existing data. You can overwrite or destroy data when you misplace these device drivers.

This chapter describes the device drivers that:

- Send data to and from a terminal; see page 79.
- Write literal character strings to a pipeline; see page 80.
- Read from and writing to data sets; see page 81.
- Read to and write from a BatchPipes pipe; see page 84.
- Read and Write REXX exec variables; see page 87.
- Combine records from other device drivers; see page 89.

Sending Data to and from a Terminal (TERMINAL)

The TERMINAL stage command writes to or reads from a terminal. When TERMINAL is the first stage, it reads from the terminal and writes the records to its primary output stream. When it is in any other stage, it reads records from its primary input stream and writes them to the terminal. In Figure 147, TERMINAL is not the first stage, so it writes to the terminal.

```
pipe literal Hello out there | terminal
Hello out there
READY
```

Figure 147. TERMINAL Stage Command Example 1

TERMINAL also writes its input records to the following stage. In the example in Figure 148, the string Hello out there is displayed on the terminal and is also written to the data set TERMINAL.LOG.

```
pipe literal Hello out there | terminal | > terminal.log
Hello out there
READY
```

Figure 148. TERMINAL Stage Command Example 2

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

Figure 149 shows an example in which the records entered at the terminal are written to the data set TYPETO.DATA. The blank line after to a data set. is a null line the user entered to end the TERMINAL stage. In other words, the user pressed the Enter key without first typing anything.

```
pipe terminal | > typeto.data
You can use TERMINAL to type to a data set.
READY
```

Figure 149. TERMINAL Stage Command Example: Typing to a Data Set

Writing Literal Character Strings to a Pipeline (LITERAL)

The LITERAL stage command lets you write a character string to the pipeline. You specify the string as the operand, including any leading or trailing blanks. In Figure 150, LITERAL writes the string Hello, World. to its output stream. TERMINAL, which is the next stage, displays the string on your terminal.

```
pipe literal Hello, World.|terminal
Hello, World.
READY
```

Figure 150. LITERAL Stage Command Example 1

After writing the operand to its output stream, LITERAL copies any records in its input stream to its output stream. Therefore, you can use LITERAL to put a header on your output. For example, Figure 151 shows how to combine TSO and LITERAL stage commands to write a heading for the output of a TSO command.

```
pipe tso listcat | literal My cataloged data sets: | terminal
My cataloged data sets:
IN CATALOG:IFCCAT.HSML04
PREFIX.BROADCAST
PREFIX.ISPF.PROFILE
PREFIX.LOG.MISC
PREFIX.PIPE.EXEC
READY
```

Figure 151. TSO, LITERAL, and TERMINAL Stage Commands Example

The TSO stage command writes the response from the LISTCAT command to its output stream a record at a time. Before LITERAL processes its input, it writes its operand to its output stream. In this example, LITERAL writes My cataloged data

sets: to its output stream. Then it copies its input stream (the results of the LISTCAT command) to its output stream. The TERMINAL stage command writes the records in its input stream (the header followed by the command results) to the screen.

Note that the end of the literal string is the last character before the stage separator (|). In Figure 152 the output on the display has two trailing blank characters (though you normally do not see them on the screen). This is not significant in this case but it can be important when records are modified in the pipeline. The first space after LITERAL is not part of the literal string. Any additional spaces, however, are included at the left of the record written.

```
pipe literal ... world.|literal Hello... | terminal
Hello...
... world.
READY
```

Figure 152. LITERAL Stage Command Example 2

In Figure 152, the second LITERAL stage command writes its string (Hello...) to its output stream before copying the records from its input stream (...world.). Because LITERAL works this way, header records are often added to data near the end of the pipeline, not near the beginning as one might expect.

Reading from and Writing to Data Sets

BatchPipeWorks provides stage commands for working with physical sequential data sets and partitioned data sets:

- < (read data set) — reads a physical sequential data set or a member of a partitioned data set and writes the records to its output stream.
- > (rewrite data set) — reads records from its input stream and writes them to an existing physical sequential data set or member of a partitioned data set (replacing the contents of the data set or member).
- >> (append physical sequential data set) — reads records from its input stream and writes them to an existing physical sequential data set, appending to the contents of the data set. You cannot append to a member of a partitioned data set using the >> stage command.
- QSAM (read or write file) — if QSAM is the first stage in a pipeline, it reads data from the specified file and writes them to its primary output stream. If QSAM is not the first stage in a pipeline, it reads records from its primary input stream and writes them to the specified file.

The < Stage Command

The < (read data set) stage command reads a physical sequential data set or a member of a partitioned data set and writes the records to its output stream. The < stage command must be the first stage of a pipeline. Specify a data set identifier as the operand. The data set identifier can be the name of a physical sequential data set, the name of a partitioned data set and the member that you want to read, or the ddname of an allocated data set.

Figure 153 reads a data set named PREFIX.DAY.LIST and writes the records to its output stream. The TERMINAL stage displays the records.

```
pipe < 'prefix.day.list' | terminal
Morose Monday
Tranquil Tuesday
Wonderful Wednesday
Tumultuous Thursday
Fabulous Friday
Spectacular Saturday
Sedate Sunday
READY
```

Figure 153. < Stage Command Example

Note that the name of the data set is enclosed in single quotation marks. This is because PREFIX.DAY.LIST is the fully qualified name of the data set. You do not need to enclose the name in quotation marks if you are not using the fully qualified name.

When using the < stage command, leave a blank between < and the data set identifier. The < stage command reads both fixed and variable format data sets.

The > Stage Command

The > (rewrite data set) stage command reads records from its input stream and writes those records to a physical sequential data set or member of a partitioned data set. The physical sequential data set must already exist. The member of the partitioned data set may or may not exist. The physical sequential data set is replaced. If the member of the partitioned data set exists, it is replaced. If the member does not exist, it is created. Specify a data set identifier as the operand. The data set identifier can be the name of an existing physical sequential data set, the name of a partitioned data set and the member that you want to replace or create, or the ddname of an allocated data set. A > stage cannot be the first stage of a pipeline.

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

```
pipe tso listcat | > space.data | terminal

pipe tso listcat | terminal | > space.data
```

In the example, the order of the > and TERMINAL stage commands is reversed. TERMINAL displays the LISTCAT response on your terminal, but also writes the records to its output stream. The input stream of the > stage command is connected to the output stream of TERMINAL. So, the > stage reads the response records and writes them to the SPACE.DATA data set, replacing the contents of the data set.

In Figure 154, the < stage command reads the data set DAY.LIST, writing the records to its output stream. Then the > stage command reads those records from its input stream and writes them to the existing data set NEWDAY.LIST.

```
pipe < day.list | > newday.list
READY
```

Figure 154. > Stage Command Example

You can specify operands following the data set identifier to force records to have the specified record length of the data set you are writing to.

- The CHOP operand truncates input records that are longer than the specified record length of the data set.
- The COERCE operand pads input records with blanks or truncate input records to the record length of the data set.
- The PAD operand (with a character) pads records to the record length of the data set.

If you write to a data set with a variable length record format, you do not need to force records to a specified length.

The >> Stage Command

The >> (append physical sequential data set) stage command reads records from its input stream and writes those records to an existing physical sequential data set. The records are appended to the physical sequential data set. A >> stage command cannot be the first stage of a pipeline.

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

```

:
/***** REXX *****/
/* PREFIX.PIPE.EXEC(APPLLOG) */
/*      -- Append a record to the log data set */
/*****
'pipe',
  'literal MYEXEC run on' date() time(),
  '| >> myexec.log'
:

```

You can force records to have the specified record length of the data set you are writing to by specifying the CHOP, COERCE, and PAD operands as you do for the > stage command.

The QSAM Stage Command

When QSAM is the first stage in a pipeline, it reads data from the specified file and writes them to its primary output stream. On QSAM, you specify the file from which records are read. Figure 155 shows the command that allocates the file INFILE and the PIPE command that reads data from INFILE and passes the data to the TERMINAL stage, which displays the records.

```
alloc file(infile) da('prefix.day.list') shr
pipe qsam infileterminal
```

Figure 155. QSAM Example - Reading From a File

When QSAM is not the first stage in a pipeline, it takes records from its primary input stream and writes them to the file specified on the command. Figure 156 shows the commands that allocate the files INFILE and OUTFILE. Then, it shows the PIPE command that reads records from INFILE and passes them to the second QSAM stage, which writes the records to the file OUTFILE.

```
alloc file(infile) da('prefix.day.list') shr
alloc file(outfile) da('prefix.newday.list') like('prefix.day.list')
pipe qsam infileqsam outfile
```

Figure 156. QSAM Example - Writing to a File

Device Drivers Used with BatchPipes (BPREAD, BPWRITE, and BPCOPY)

There are three stage commands that work only in BatchPipes fittings, with pipe data sets that are controlled by BatchPipes:

- BPREAD (BatchPipes read) — reads records from a BatchPipes writer or a BatchPipes pipe.
- BPWRITE (BatchPipes write) — writes records to a BatchPipes reader or a BatchPipes pipe.
- BPCOPY (BatchPipes copy) — takes data from an application or a BatchPipes pipe data set and copies the records by writing them to one or two files specified on the command.

This section describes their use and gives examples. Other examples of these commands are in “Using a BatchPipeWorks Fitting” on page 27.

The BPREAD Stage Command

BPREAD receives data from a BatchPipes pipe data set. As records flow through a pipe data set, BPREAD takes the records and writes them to its primary output stream. Figure 157 on page 85 shows BPREAD taking data from the pipe data set PIPED.DATA and writing them to the BPWRITE stage.

```
//BATPIPE DD DSN=PIPED.DATA,LRECL=80,RECFM=F,  
//  SUBSYS=(BP01,'FIT=BPREAD|BPWRITE')
```

Figure 157. BPREAD and BPWRITE Stage Commands

The BPWRITE Stage Command

BPWRITE returns data to a BatchPipes pipe data set. As records flow through a fitting, BPREAD takes the records from its primary input stream and returns them to the pipe data set. Figure 157 shows BPWRITE taking data from its primary input stream and returning them to the BatchPipes pipe data set PIPED.DATA.

The BPCOPY Stage Command

BPCOPY takes records that flow into a BatchPipes pipe data set and copies them to one or two files specified on the BPCOPY command. In Figure 158, BPCOPY takes records flowing through the pipe data set PIPED.DATA and copies them to the file COPYFILE.

```
//BATPIPE DD DSN=PIPED.DATA,LRECL=80,RECFM=F,  
//  SUBSYS=(BP01,'FIT=BPCOPY COPYFILE')  
//COPYFILE DD DSN=COPY.DATA,DISP=NEW,SPACE=(1,1),RECFM=F,  
//  DSORG=PS,LRECL=80
```

Figure 158. BPCOPY Fitting Example - Making One Copy of the Data

Figure 159 takes records flowing through the BatchPipes pipe data set PIPED.DATA and copies them to two files: COPYA and COPYB.

```
//BATPIPE DD DSN=PIPED.DATA,LRECL=80,RECFM=F,  
//  SUBSYS=(BP01,'FIT=BPCOPY COPYA COPYB')  
//COPYA DD DSN=COPY.A.DATA,DISP=NEW,SPACE=(1,1),RECFM=F,  
//  DSORG=PS,LRECL=80  
//COPYB DD DSN=COPY.B.DATA,DISP=NEW,SPACE=(1,1),RECFM=F,  
//  DSORG=PS,LRECL=80
```

Figure 159. BPCOPY Fitting Example - Making Two Copies of the Data

Note that, unlike BPREAD and BPWRITE, BPCOPY does not work in half-pipe fittings.

Determining if Data Sets Exist (STATE)

Use the STATE stage command when you want to determine if a physical sequential data set or a member of a partitioned data set exists with a particular name. STATE searches for the specified physical sequential data set or member of a partitioned data set and writes the fully qualified name to the primary output stream when the data set or member exists. When the data set or member does not exist, STATE copies its primary input stream to its primary output stream.

You can specify the physical sequential data set name (fully qualified or not), the ddname of an allocated data set, or a member of a partitioned data set and the ddname of that allocated partitioned data set as the input to STATE.

STATE works like the TSO and COMMAND stage commands; it processes any operand first, then process the records from its input streams. One data set identifier should be on each input record. Figure 160 shows an example.

```
pipe literal typeto.data | state day.list | terminal
PREFIX.DAY.LIST
PREFIX.TYPETO.DATA
READY
```

Figure 160. STATE Stage Command Example

In REXX execs, use STATE and VAR to put data set information into stem variables that can be used in the body of the exec. (See Figure 161.)

```
/****** REXX ******/
/* PREFIX.PIPE.EXEC(DSNSTAT) */
/* -- Using STATE to put information into stem variables */
/*******/
state: procedure
parse arg dsn .

'pipe (endchar ?)',
  | state' dsn,          /* Check for Existence */
  | spec /LISTDS ' / 1 1-* n /' n", /* Build a LISTDS command */
  | tso',                /* Run it */
  | spec 1-* 1',         /* Get the DSN */
  | read read 1-* nw',   /* Get RECFM, LRECL, BLKSIZE, & ORG */
  | var STATE'           /* Put it in variable STATE */
if RC/=0 then do
  say RC', Data Set' dsn 'not found.'
  state = ''
end
return state
```

Figure 161. STATE and VAR Stage Commands Example

Packing and Unpacking Data Sets

BatchPipeWorks lets you read data sets that are packed. It also lets you create packed data sets. For more information about how to create packed data sets, see the PACK and UNPACK stage commands in the *BatchPipeWorks Reference*.

Reading and Writing REXX Exec Variables

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

STEM Stage Command

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

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

When STEM is the first stage, it uses the variable *stem.0* to determine how many records to write to its output stream. In Figure 162, STEM looks at *INVAL.0* to determine how many variables to read. Because *INVAL.0* is set to 3, STEM reads the variables *INVAL.1*, *INVAL.2*, and *INVAL.3*. It writes the contents of these variables to its output stream (one record per variable). *TERMINAL* displays the records.

```
/****** REXX *****/
/* PREFIX.PIPE.EXEC(STEMFRST) */
/*      -- STEM as the first stage */
/******/
inval.0 = 3
inval.1 = 'red'
inval.2 = 'white'
inval.3 = 'blue'
inval.4 = 'green'
'pipe',
      'stem INVAL.',
      '| terminal'
exit rc
```

Figure 162. *STEMFRST EXEC: Using STEM Stage Command to Read REXX Variables*

Following is the result of executing *STEMFRST*. Because *INVAL.0* is 3, not 4, green is not displayed on the terminal.

```
stemfrst
red
white
blue
READY
```

When STEM is not first, it reads its input records and writes them to the specified stem variable. STEM also sets *stem.0* to the number of variables set. In Figure 163 on page 88, for example, the variable *OUTVAL.0* is set to 3.

```

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

```

Figure 163. STEMMID EXEC: Using STEM Stage Command to Write REXX Variables

Following is the result of executing STEMMID. STEM writes the three records in its input stream to the stem variable OUTVAL.

```

stemmid
OUTVAL.0 = 3
OUTVAL.1 = red
OUTVAL.2 = white
OUTVAL.3 = blue
READY

```

Use STEM when you want to use the REXX language to do some processing that is not easily done in BatchPipeWorks. You use STEM to temporarily leave BatchPipeWorks. In Figure 164, for example, stage commands read a data set and preprocess the data (using < and LOCATE). STEM puts the preprocessed records in a stemmed variable so that they can be counted. If more than 20 records are counted, STEM is used again to gather the records, which are then written to a data set.

```

/***** REXX *****/
/* PREFIX.PIPES.EXEC(DOUBLE) */
/* -- Find 'the the' */
/***** REXX *****/
'pipe',
  '< mybook.script', /* Read a data set */
  '| locate /the the/', /* Locate records with 'the the' */
  '| stem LINE.' /* Put records in a stem variable */
if line.0 > 20 then /* Now let's use REXX for a while */
  'pipe', /* More than 20 errors... */
  'stem LINE.', /* Get variables */
  '| > error.file' /* Write to data set */
else
  'pipe', /* Less than 20 errors... */
  'stem LINE.', /* Get variables */
  '| terminal' /* Display them */

```

Figure 164. STEM Stage Command Example: Using REXX with BatchPipeWorks

Using STEM, you benefit from the strengths of both BatchPipeWorks and REXX and in a single exec, you can switch between them. Note that REXX determines how to end BatchPipeWorks processing.

VAR Stage Command

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

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

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

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(VARTEST) */
/*      -- Exec fails because DSID is null */
/*****
DSID=''
'pipe',
    'literal WORK.DATA',      /* Put the data set ID in the pipeline */
    'var DSID',               /* Assign it to variable DSID */
    'drop 1',                 /* Drop the record */
    'append <' DSID,          /* Read the data set (FAILS) */
    'terminal'

```

Figure 165. VAR Stage Command Example

Combining Records from Other Device Drivers

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

APPEND Stage Command

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

The stage command that you specify as an APPEND operand can be any device driver or host command interface that can be first in a pipeline. In Figure 166, the output of a TSO LISTALC STATUS command is added to the pipeline after the output of TSO LISTCAT.

```
pipe tso listcat | append tso listalc status | terminal
IN CATALOG:ICFCAT.HSML04
PREFIX.BROADCAST
PREFIX.MVSPIPES.CNTL
PREFIX.ISPF.PROFILE
PREFIX.LOG.MISC
PREFIX.PIPE.EXEC
PREFIX.PIPE.LOAD
PREFIX.PIPE.REXX
--DDNAME---DISP--
SYS1.TSO.CLIST
  SYSPROC  KEEP
SYSTOOLS.CLIB
  KEEP
SYS1.TCPIP.TCPHELP
  KEEP
TERMFILE  SYSPRINT
PREFIX.ISPF.PROFILE
  ISPPROF  KEEP
ISP.PHONE.CLISTS
  SYSEXEC  KEEP
SYS1.ISP.ISPEXEC
  KEEP
PREFIX.PIPE.EXEC
  KEEP
PREFIX.PIPE.REXX
  REXX     KEEP
READY
```

Figure 166. APPEND Stage Command Example

Figure 167 shows how to use APPEND to write the contents of two data sets as a single data set:

```
pipe < monday.data | append < tuesday.data | > week.data
READY
```

Figure 167. Appending Two Data Sets

The < stage command writes the contents of the data set MONDAY.DATA to its output stream. APPEND copies these records from its input stream to its output stream. Then APPEND runs the second < stage command, writing the resultant records to its output stream. Therefore, the records of MONDAY.DATA precede the records of TUESDAY.DATA in the pipeline. Finally, the > stage command reads the records from its input stream and writes them to WEEK.DATA. WEEK.DATA contains the records of MONDAY.DATA followed by those of TUESDAY.DATA.

This example shows another way to combine two literals:

```
pipe literal Hello...|append literal ... world.| terminal
Hello...
... world.
READY
```

PREFACE Stage Command

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

Like APPEND, the stage command that you specify as a PREFACE operand can be any device driver or host command interface that can be first in a pipeline. Figure 168 shows how to use PREFACE to write the contents of two data sets as a single data set:

```
pipe < thursday.data | preface < wednesday.data | >> week.data
READY
```

Figure 168. PREFACE Stage Command Example

The < stage command writes the contents of the data set THURSDAY.DATA to its output stream. PREFACE copies these records from its input stream to its output stream. Then PREFACE runs the second < stage command, writing the resultant records to its output stream, preceding the records already in the output stream. Therefore, the records of WEDNESDAY.DATA precede the records of THURSDAY.DATA in the pipeline. Finally, the >> stage command reads the records from its input stream and writes them to WEEK.DATA, appending any existing data in the data set.

One final observation: do not specify a LITERAL stage command as a PREFACE operand; specifying LITERAL by itself yields the same results.

Chapter 6. Using Stage Commands for Multistream Pipelines

This chapter contains General-Use Programming Interface and Associated Guidance Information.

This chapter builds on the information contained in “Multistream Pipelines” on page 9. It describes stage commands that are frequently used in multistream pipelines, as follows:

- Copy input records to multiple output streams; see “FANOUT Stage Command” on page 93.
- Copy records into more than one output stream; see “FANINANY Stage Command” on page 96
- Combine multiple input streams into a single stream; see “FANIN Stage Command” on page 98.
- Read records from all input streams and merge those records into a single record; see “OVERLAY Stage Command” on page 100.
- Read one record from all input streams for each output record built; see page “SPECS Stage Command” on page 101.
- Count bytes, character strings, or records; see “COUNT Stage Command” on page 103.
- Copy records from all input streams to its primary output stream; see page “MERGE Stage Command” on page 105.
- Match records in its primary input stream with records in its secondary input stream and write matched and unmatched records to different output streams; see “LOOKUP Stage Command” on page 107.

FANOUT Stage Command

Using labels on filter stage commands, you can process data that the stage command would otherwise discard. But, to *copy* the records in the pipeline to more than one output stream, use the FANOUT stage command. FANOUT copies each record that it reads from its input stream to all of its output streams. To use FANOUT, put a label in front of it and refer to that label elsewhere in the pipeline, as you do with other filter stage commands.

For example, suppose you are compiling demographic data. From a master data set you want to extract two sets of records and place them in separate data sets. In one data set, you want the names of all people who were born in 1956. In another data set, you want to list all males. All information concerning a person is on one master data set record. Here are some example records:

Name	Sex	Date of Birth	State
Smith, Robert M.	MALE	12 06 1956	NY
Jones, Morgan E.	FEMALE	05 05 1959	PA
Public, Waldo Q.	MALE	11 13 1960	CA

It is not possible to use the secondary stream of LOCATE to solve the problem. Suppose you use LOCATE to select all records containing 1956 and write them to a data set named DOB1956.DATA. Then you try to use the secondary stream of LOCATE /1956/ to find all males. This will not yield correct results because the secondary stream would not contain any records from 1956. Your output data set would be missing all males born in 1956.

Because you want all records to be processed by both LOCATEs, use FANOUT to copy the stream.

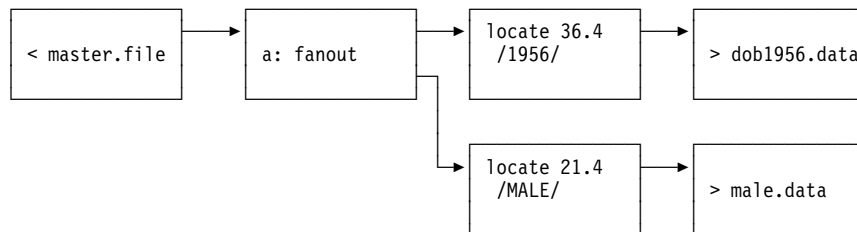


Figure 169. Map of FANOUT Example

Figure 170 shows the PIPE command.

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(FAN2) */
/* -- Example of using FANOUT */
/*****/
'pipe (endchar ?)',
  '< master.file', /* Read master data set */
  '| a: fanout', /* Copy records to all output streams */
  '| locate 36.4 /1956/', /* Locate everyone born in 1956 */
  '| > dob1956.data', /* Write records to data set */
  '?',
  'a:',
  '| locate 21.4 /MALE/', /* Locate all males */
  '| > male.data' /* Write records to data set */

```

Figure 170. FANOUT Example

To create a third data set in which all of the residents of New York state are listed, define yet another output stream (referred to as a *tertiary stream*). FANOUT supports more than two streams. To define a third output stream, refer to the same label again. Figure 171 on page 95 shows a map of the pipeline.

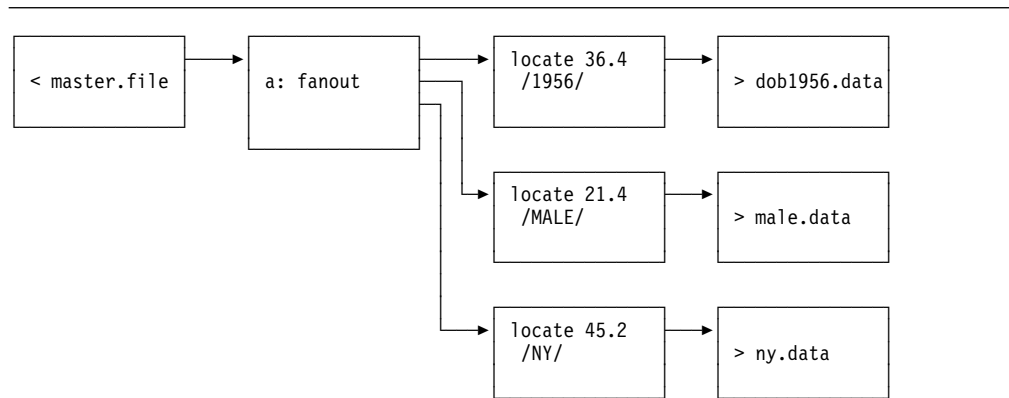


Figure 171. Map of FANOUT Example Using a Tertiary Stream

Figure 172 shows the PIPE command, in which a third pipeline is added. The first stage in the pipeline is a label that refers to the label defined on FANOUT, so an output stream of FANOUT is connected to the input stream of the stage `locate 45.2 /NY/`.

```

/***** REXX *****/
/* PREFIX.PIPES.EXEC(FAN3) */
/* -- Example of using FANOUT */
/*****/
'pipe (endchar ?)',
  '< master.file', /* Read master data set */
  '| a: fanout', /* Copy records to all output streams */
  '| locate 36.4 /1956/', /* Locate everyone born in 1956 */
  '| > dob1956.data', /* Write records to data set */
  '?',
  'a:',
  '| locate 21.4 /MALE/', /* Locate all males */
  '| > male.data', /* Write records to data set */
  '?',
  'a:',
  '| locate 45.2 /NY/', /* Locate all NY residents */
  '| > ny.data' /* Write records to data set */

```

Figure 172. FANOUT Example Using a Tertiary Stream

To determine whether a stage command supports more than two streams, refer to the stage command description in the *BatchPipeWorks Reference*. To use more than two streams on stage commands that support them, use the label repeatedly, as we did in the example.

FANINANY Stage Command

FANINANY reads a record from any input stream that has one and writes the record to its output stream. Use it to combine the records of several pipelines. For example, Figure 173 shows how to use FANINANY with LOCATE to do an OR function. The pipeline writes to a data set a listing of all data sets with EXEC or REXX qualifiers. Any record containing EXEC or REXX as a qualifier is selected. FANINANY reads the selected records from both input streams and writes them to its output stream.

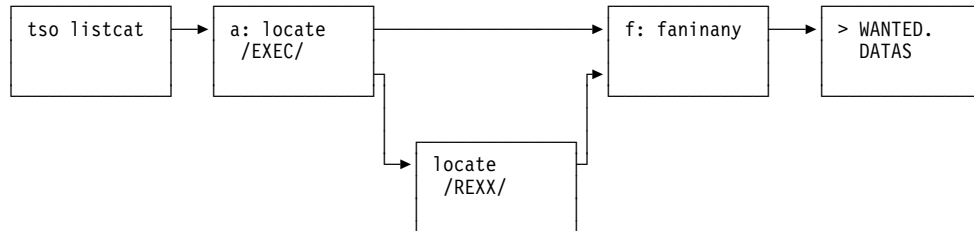


Figure 173. Map of FANINANY Example

Figure 174 shows the PIPE command to do it.

```
/* ***** REXX ***** */
/* PREFIX.PIPE.EXEC(LOCFAN) */
/* -- a FANINANY Example */
/* ***** */
'PIPE (endchar ?)',
  'tso listcat ', /* Put list in pipeline */
  '| a:locate /EXEC/', /* Find names with EXEC qualifier */
  '| f:faninany', /* Combine the streams */
  '| > WANTED.DATAS', /* Write the list in a data set */
  '?', /* non-EXEC data sets go here */
  'a:',
  '| locate /REXX/', /* Find names with REXX qualifier */
  '| f:' /* Route them to FANINANY */
```

Figure 174. LOCFAN EXEC: A FANINANY Example

The TSO stage command issues a LISTCAT command that puts into the pipeline a list of all data sets with your prefix as the first qualifier. The next stage command, LOCATE, finds the data sets having a qualifier of EXEC and writes the names to its primary output stream. These records flow into FANINANY. LOCATE writes records that do not have a qualifier of EXEC to its secondary output stream. A second pipeline, which begins with a stage containing only the label a, processes the records on LOCATE's secondary output stream.

The second pipeline looks for any data sets having a qualifier of REXX, again with a LOCATE stage. Any records that match are sent to the next stage command. The next stage contains the label f. This connects the pipeline to the secondary input of FANINANY.

FANINANY reads records from its primary and secondary inputs in whatever order they arrive. FANINANY writes all records to its primary output. In effect,

FANINANY combines its two input streams. The next stage, a > stage command, writes them to an existing data set.

The records in WANTED.DATAS are in the same relative order as they were in the output from the TSO LISTCAT command. This is because the multistream portion of this pipeline contains only a LOCATE stage and LOCATE does not *delay the records*. More information about delaying the records appears in “How Stages Delay the Records” on page 113.

Identifying Streams

In the operands of some stage commands that we'll be discussing, you'll need to refer to specific streams. So far we've been referring to streams as *primary input streams*, *secondary input streams*, and so on. However, these terms cannot be used as stage command operands. Instead, we use *stream numbers* or *stream names* as stage command operands to identify a particular stream.

Stream Numbers

Each stream number applies to a single stage. Each stage has a primary stream: stream 0. If the stage also has secondary stream, that stream number is 1. A tertiary stream is number 2, and so on. When several streams are fed into a stage or flow out of a stage, the pipelines are associated with streams in the order in which the pipelines are written. For example, the FANOUT stage command in the following example uses four output streams. The primary stream is stream 0. The other pipelines are connected to streams in the order that the pipelines are written:

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(FANOUT4) */
/* -- FANOUT using four output streams */
/*****
'pipe (endchar ?)',
  '< master.file',
  '| a: fanout',          /* Primary stream (stream 0) */
  '| > all.data',
  '?',
  'a:',                  /* This LOCATE stage is connected to */
  '| locate 36.4 /1990/', /* FANOUT's output stream 1 */
  '| > dob1990.data',
  '?',
  'a:',                  /* This LOCATE stage is connected to */
  '| locate 21.4 /MALE/', /* FANOUT's output stream 2 */
  '| > male.data',
  '?',
  'a:',                  /* This LOCATE stage is connected to */
  '| locate 45.2 /NY/',   /* FANOUT's output stream 3 */
  '| > ny.data'

```

A map of the pipeline, with FANOUT's stream numbers, is shown in Figure 175 on page 98.

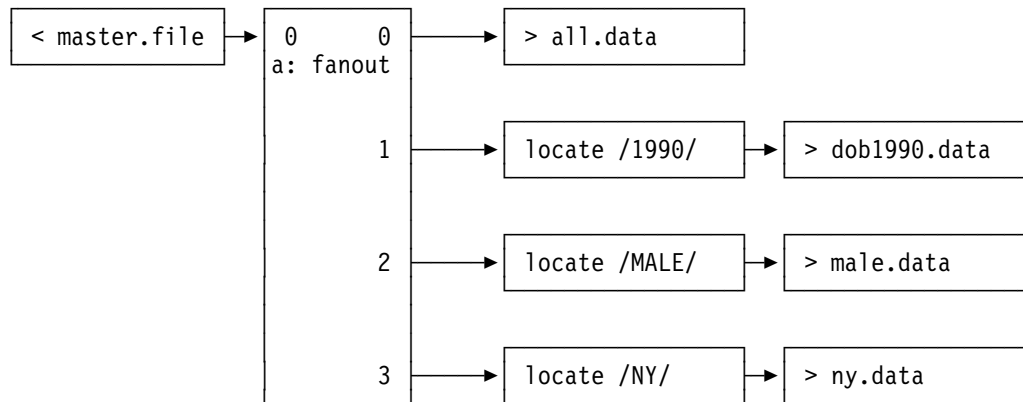


Figure 175. Map of Pipeline Showing Stream Numbers

Notice that all the other stage commands have connections to only their primary input and output streams (stream 0).

Stream Names

A *stream name* is a stream identifier that assigns a symbolic name to a stream. You name a stream to avoid keeping track of its stream number. Instead of using a stream number as an operand on a stage command that combines streams, you can refer to that stream by the stream name you have assigned.

You name a stream on the PIPE command, and on the ADDPIPE and CALLPIPE subcommands like this: add an identifier to the label by writing the label (in this case a), immediately followed by a period (.) and up to four alphabetical characters or a combination of alphabetical characters and digits that includes at least one alphabetical character. A stream identifier must be immediately followed by a colon with no intervening blanks.

Simpler rules apply to the ADDSTREAM subcommand, where you name a stream as described above, but without the surrounding label or colon.

For instance, **a.mstr**: assigns the symbolic name `mstr` to a stream that can be referenced by a stage command. See Figure 179 on page 100 for an example that uses a stream number to identify the primary input stream and a stream name to identify the secondary input stream.

FANIN Stage Command

Like FANINANY, FANIN reads records from its input streams and writes those records to its output stream. But, unlike FANINANY, FANIN reads all the records from one stream before reading records from another. By default, FANIN reads all of its primary stream, then all of its secondary, and so on, until it has processed all input streams. Stream numbers or stream names can be entered as operands to specify a different order, or a subset of all connected streams, or both.

For example, one way to read TEST.DATA and TEST1.DATA into a pipeline is shown in Figure 176 on page 99. Another way is to use APPEND, as Figure 167 on page 90 shows.

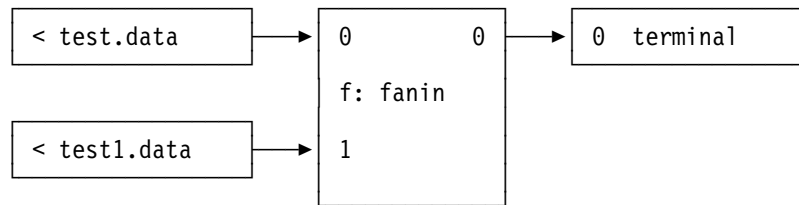


Figure 176. Map of FANIN Example

Figure 177 shows the PIPE command.

```

/***** REXX *****/
/* PREFIX.PIPES.EXEC(FANINEX) */
/* -- a FANIN Example */
/*****/
'pipe (endchar ?)',
  '< test.data',      /* Read first data set */
  '| f: fanin',      /* Gather streams in default order */
  '| terminal',      /* Display data sets */
  '?',
  '< test1.data',     /* Read second data set */
  '| f:'             /* Feed it to FANIN (is FANIN stream 1) */
exit rc

```

Figure 177. FANIN Example Showing Default Stream Order

Figure 177 defines two streams into FANIN. The two input streams are connected to stages that read the desired data sets.

To specify a different order, put stream numbers after the FANIN keyword, as shown in Figure 178. In that example, FANIN reads all of stream 1, then all of stream 0. The order in which the data sets are displayed is reversed. TEST1.DATA is displayed followed by TEST.DATA.

```

/***** REXX *****/
/* PREFIX.PIPES.EXEC(FANINEX2) */
/* -- a FANIN Example showing stream numbers */
/*****/
'pipe (endchar ?)',
  '< test.data',      /* Read first data set */
  '| f: fanin 1 0',  /* Request a specific order on FANIN */
  '| terminal',      /* Display data sets */
  '?',
  '< test1.data',     /* Read second data set */
  '| f:'             /* Feed it to FANIN (is FANIN stream 1) */
exit rc

```

Figure 178. FANIN Example Showing Stream Numbers

Figure 179 on page 100 shows the same example with a stream name identifying the secondary input stream. When you use stream names, you don't need to keep track of the stream numbers.

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(FANINEX3) */
/* -- a FANIN Example showing stream numbers and names */
/***** */
'pipe (endchar ?)',
  '< test.data', /* Read first data set */
  '| f: fanin td1 0', /* Request a specific order on FANIN */
  '| terminal', /* Display data sets */
  '?',
  '< test1.data', /* Read second data set */
  '| f.td1:' /* Feed it to FANIN as stream TD1 */
exit rc

```

Figure 179. FANIN Example Showing Stream Names and Numbers

OVERLAY Stage Command

OVERLAY reads records from all its input streams and merges those records into a single record. Each character of the output is from the record from the highest numbered stream with a nonblank character in the corresponding position. Underscores are treated like any other character.

In the following example, we create lines of text for a direct mail marketing campaign. The data set MASTER.FILE contains a mailing list, one name per record. The data set DIRMAIL.SCRIPT contains a single line of text to be printed on the envelopes. We want to overlay the standard line of text with personal information extracted from the records in MASTER.FILE. The line of text contains space for the personal information.

In Figure 180, the TAKE stage limits the number of MASTER.FILE records being processed; SPECS extracts information from the MASTER.FILE records; DUPLICATE duplicates the record from DIRMAIL.SCRIPT. A standard line of text and the desired personal information flow into OVERLAY. OVERLAY combines the two records and then TERMINAL displays the results.

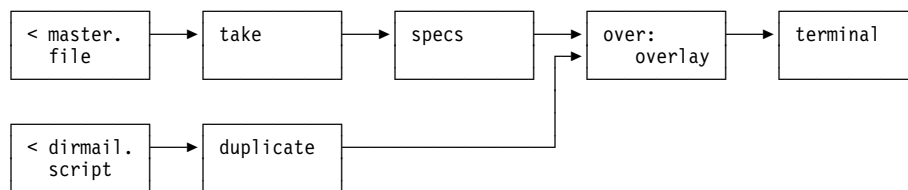


Figure 180. Map of Overlay Example

The complete REXX exec, named OVERDATE, is in Figure 181 on page 101.

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(OVERDATE) */
/* -- Overlay date for direct mail envelopes */
/***** */
numrecs=3 /* Number of master data set records to do */
'pipe (endchar ?)',
  '< master.file', /* Read the MASTER.FILE data set */
  '| take' numrecs, /* Take desired number of records */
  '| specs 36.4 18.4', /* Create output record containing year */
  '| over: overlay', /* Overlay year with direct mail script */
  '| terminal', /* Display records */
  '?',
  '< dirmail.script', /* Read the DIRMAIL.SCRIPT data set */
  '| duplicate' numrecs-1, /* Make correct number of copies */
  '| over:' /* Feed to secondary input of OVERLAY */
exit rc

```

Figure 181. Example of OVERLAY Stage Command: OVERDATE EXEC

The variable numrecs is set to the number of MASTER.FILE records to be processed. This number is used in the TAKE stage and in the DUPLICATE stage to create the correct number of output records. The SPECS stage extracts a field containing the year of birth from the MASTER.FILE input record. It positions the year of birth in the output record such that it will overlay the space reserved in the standard text. Below is an example run. Note the location of the birth dates in the MASTER.FILE records:

```

pipe < master.file | terminal
SMITH, ROBERT M.    MALE    12 06 1956    NY
JONES, MORGAN E.    FEMALE   05 05 1959    PA
PUBLIC, WALDO Q.    MALE     11 13 1960    CA
READY
pipe < dirmail.script | terminal
Were you born in    ? Read the important message inside!
READY
overlay
Were you born in 1956? Read the important message inside!
Were you born in 1959? Read the important message inside!
Were you born in 1960? Read the important message inside!
READY

```

SPECS Stage Command

SPECS, as a filter stage command, is described on “SPECS Stage Command” on page 59. In this chapter, we see SPECS as a multistream stage command. SPECS supports multiple input streams with the keyword SELECT. It reads one record from all its input streams for each output record built. Use the keyword SELECT followed by a stream number or a stream name whenever you wish to refer to input data from a record on a stream other than the primary one (or the one selected previously in the list of operands). Subsequent input fields (up to the next SELECT keyword) refer to the record on the stream specified.

Figure 183 on page 102 shows an example that reads a data set, changes some of its records, and then displays the original and changed records side-by-side.

(The first 35 characters of each record are displayed.) The map for the pipeline is in Figure 182 on page 102.

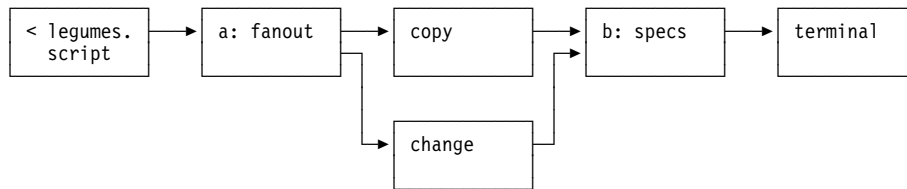


Figure 182. Map of SPECS SELECT Example

An example of SPECS SELECT follows in Figure 183.

```

/***** REXX *****/
/* PREFIX.PIFE.EXEC(SELECT) */
/*      -- Demonstrate SPECS Select operand */
/*****/
'pipe (endchar ?)',
  '< legumes.script',          /* Read data set */
  '| a: fanout',              /* Copy records to secondary output */
  '| copy',                  /* Prevent a pipeline stall */
  '| b: specs 1-35 1',        /* Get data from primary stream (0) */
  'select 1',                /* Select the secondary stream (1) */
  '1-35 41',                 /* Put data in second half of record */
  '| terminal',              /* Display the results */
  '?',
  'a:',                      /* Records from FANOUT */
  '| change /Pole beans/**Crop Failure**/', /* Change the data */
  '| b:'                     /* Send back to SPECS */
exit rc

```

Figure 183. Example of SPECS SELECT Operand: SELECT EXEC

In the example, the < stage reads the data set LEGUMES.SCRIPT and writes the records to its output stream. FANOUT copies the records to its secondary stream, where CHANGE changes some of the data. The SPECS stage puts the original record and the changed record side-by-side on a single output record, which TERMINAL then displays.

The COPY stage command is used between FANOUT and SPECS to avoid an error called a *pipeline stall*. A pipeline stall, which occurs with multistream pipelines, is caused when BatchPipeWorks cannot run any of the stages because every stage is waiting for another stage to perform some action. BatchPipeWorks detects a stall and ends the PIPE command. (Try running the exec after removing the COPY stage command.)

A stall occurs when COPY is omitted because SPECS needs two records at a time, but FANOUT writes one record to each output stream in turn. When FANOUT writes a record to stream 0, it waits for SPECS to read it. SPECS tries to read the record as well as one on its secondary input stream. But, there isn't a record available in the secondary stream—FANOUT cannot write it because it is waiting for its first output to finish. In effect, SPECS and FANOUT are waiting for each other, causing all the other stages to wait. The < stage waits for FANOUT to read

the next record. The TERMINAL stage waits for SPECS to write a record. Thus, we have a stall.

Inserting the COPY stage command avoids the stall. COPY reads the record that FANOUT writes to its primary output stream. FANOUT is then free to write a record to its secondary output stream. When SPECS demands its two records, BatchPipeWorks is able to make the records available, as the following example shows:

```
pipe < legumes.script | terminal
Peas
Bush beans
Pole beans
Lima beans
READY
select
Peas                                Peas
Bush beans                          Bush beans
Pole beans                          **Crop Failure**
Lima beans                          Lima beans
READY
```

SPECS stops when all input streams are empty. If one stream empties before the others, SPECS acts as though that stream contains null records. For information about pipeline stalls, see “Avoiding Pipeline Stalls” on page 20 and “Fixing Pipeline Stalls” on page 110.

COUNT Stage Command

COUNT, as a filter stage command, is described on “Counting Characters, Words, and Records (COUNT)” on page 73. In this chapter, we see COUNT as a multistream stage command that writes different outputs depending on whether its secondary stream is connected. Previous COUNT examples showed how COUNT works when its secondary stream is not connected. It writes a single record containing the requested tallies to its primary output. When the secondary stream is connected, however, this is not the case.

When the secondary stream is connected, COUNT copies its *input stream*, not the tally record, to its primary output. The record containing the tallies is written to the *secondary* output stream instead of the primary.

This makes COUNT far more useful. Look at the WORDUSE EXEC in Figure 184 on page 104. It lists all the words that occur in a data set in alphabetical order. Preceding each word is a number indicating the number of times the word was used. After the list of words, a summary is displayed that tells the number of lines in the data set, the total number of words, and the number of unique words.

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(WORDDUSE) */
/* -- Word Use Analyzer */
/***** */
parse arg dsn
'pipe (endchar ?)',
  '<' dsn, /* Read the data set */
  '| xlate lower 41-7f 40 ' ' ', /* Translate to lowercase, and get */
  '| ', /* rid of punctuation */
  '| a: count words lines', /* Count words and lines */
  '| split', /* Put one word per record */
  '| sort count', /* Sort unique with a count of dups */
  '| b: count lines', /* Count the number of unique words */
  '| specs 1-10 1 11-* 15', /* Format the sorted records */
  '| f: fanin', /* Make sure summary lines at end */
  '| literal Times Used Word', /* Write a header */
  '| terminal', /* Display it */
  '?',
  'a:', /* Process the word and line count */
  '| specs /Number of lines: / 1 words2 next write',
  '| /Total number of words: / 1 words1 next',
  '| f:',
  '?',
  'b:', /* Process the unique word count */
  '| specs /Total number of unique words: / 1 words1 next',
  '| f:'

```

Figure 184. WORDUSE EXEC: Example Exec to Analyze Word Use

After the data set is read, the records are translated to lowercase (XLATE) because we don't want the results to be case-sensitive. XLATE also eliminates punctuation by changing characters in the range X'41' to X'7F' to blanks (X'40'). The pair of apostrophes causes apostrophes to be retained in the records. Otherwise, all contractions (for example, "don't") and some possessives (for example, "Mary's") would be split into two words.

Next the lines and words are counted, but the record containing the count is written to COUNT's secondary stream (label a). In this example, the COUNT stage copies its input to its output stream, for processing by SPLIT.

SPLIT puts each lowercase word on a separate record, so the words can be sorted by the SORT stage command. (SORT sorts records, not items *within* records.) The COUNT operand on SORT causes SORT to eliminate duplicate records. The number of duplicates for each record is placed in the first ten bytes of the output record. The record itself follows, beginning with byte 11. So, SORT COUNT gives us the number of times each word was used. More about SORT is in "Sorting Records (SORT)" on page 74.

Each record flowing out of SORT contains a unique word. To count the total number of unique words, a COUNT LINES stage is added, and the record containing the count is written to the secondary output (label b:). COUNT passes the records themselves to the next stage.

The remainder of the first pipeline formats the records, collects the summary records from the other pipeline, adds a header, and displays everything. The two

other pipelines process records from the two COUNT stages. They add explanatory text to the counts and send the formatted records back to the first pipeline.

Notice that FANIN collects the records from all three streams. FANINANY is not used because we want the list of words to precede the summary records. FANIN guarantees this. Try substituting FANINANY in WORDUSE—the records are not displayed in the desired order.

Figure 185 shows a sample run of WORDUSE. The input data set TEST.DATA contains these lines:

```
=====
```

```
Don't worry about me--
```

worduse test.data

Times Used	Word
------------	------

1	about
---	-------

1	don't
---	-------

1	me
---	----

1	worry
---	-------

Number of lines: 2

Total number of words: 4

Total number of unique words: 4

READY

Figure 185. Counting Several Items

MERGE Stage Command

MERGE copies records from all its input streams to its primary output stream. MERGE is intended to combine sorted lists such that the output from MERGE is in order. When the records in the input streams are not sorted, MERGE still combines the records. In this case, however, the order in which MERGE writes the records is not predictable. MERGE does not verify that the input streams are sorted.

You can specify column ranges on MERGE in the same way as for SORT. When column ranges are specified, MERGE orders the records according to the data in the specified ranges. If you omit column ranges, MERGE uses the whole record to determine the order.

Figure 186 shows the map of a PIPE command that combines the responses from two TSO LISTCAT commands. Because the responses from TSO LISTCAT are sorted, the output from MERGE will be in order.

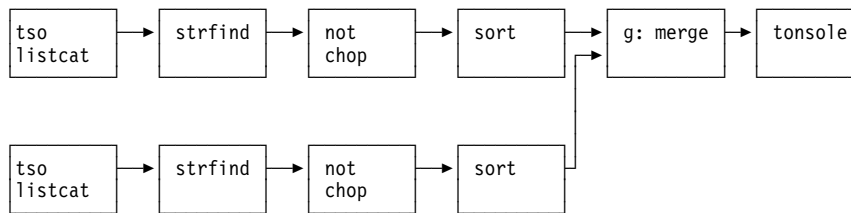


Figure 186. Map of MERGE Example

Figure 187 shows the PIPE command for the above map. The TSO stage commands list the data sets with the prefix of SYS1 and with the prefix of PREFIX. The STRFIND stage command locates all lines that begin with NONVSAM (which are the lines containing the data set names) and then the NOT CHOP stage commands eliminates the first qualifier of each data set name. These lists are then sorted. The output streams of both SORT stage commands are connected to the input streams of MERGE. Because column ranges are omitted from MERGE, it uses the entire record to determine the output order. TERMINAL displays the records from MERGE.

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(LISTMRG) */
/* -- Display merged list of all data sets on two data sets */
/*****/
"pipe (endchar ?)",
  " strfind /NONVSAM/", /* Select the first records */
  " not chop ./", /* Keep the information after the dot */
  " sort", /* Sort the list */
  " g: merge", /* Merge the sorted lists */
  " terminal", /* Display the merged and sorted list */
  "?", /* Start of the second pipeline */
  " strfind /NONVSAM/", /* Select the first records */
  " not chop ./", /* Keep the information after the dot */
  " sort", /* Sort the list */
  " g:" /* Feed to MERGE */
exit

```

Figure 187. MERGE Stage Command Example: LISTMRG EXEC

The following is an example run of LISTMRG EXEC:

```
listmrg
.BN8
.BROADCAST
.DDIR
.DUMP23E
.LOADLIB
.OBJLIB
.TEMP
.TEMP1
.TEMP2
.TESTER
READY
```

LOOKUP Stage Command

LOOKUP matches records in its primary input stream with records in its secondary input stream and writes matched and unmatched records to different output streams. Whole contents of records are matched by default, or the records are matched on the basis of a *key field* (the contents of a specified range of columns in the records).

Before finding records, LOOKUP reads all records on the secondary input stream into a buffer (called the *reference*). These records are called *master records*. LOOKUP discards master records with duplicate keys while loading the buffer.

After building the reference, LOOKUP reads records from its primary input stream and looks for a matching record in the set of reference records. The records read from the primary input stream are referred to as *detail records*. By default, entire records are compared, but you can specify column ranges to look for a key. Upon finding a match, LOOKUP writes the detail record and the matching master record to its primary output stream. Use the operand DETAILS to get just the detail records. If a detail record does not have a matching master record, LOOKUP writes the detail record to its secondary output stream.

After processing all the detail records, LOOKUP writes all unreferenced master records to its tertiary output stream. By unreferenced we mean those not matched by at least one detail record. LOOKUP writes the unreferenced records in ascending order by their keys.

Figure 188 summarizes the streams used by LOOKUP.

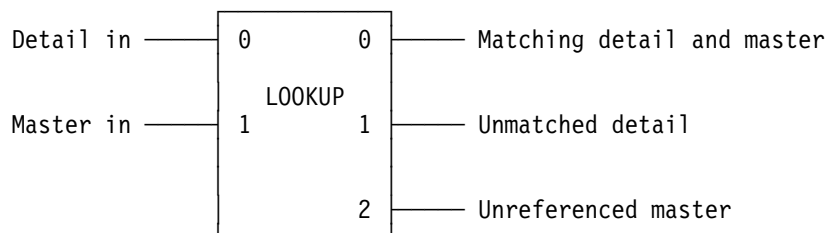


Figure 188. Map of LOOKUP Stage Command

You can see LOOKUP work by executing a REXX exec like the one in Figure 189 on page 108.

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(LOOKSTR) */
/*      -- demonstrate LOOKUP streams */
/*****/

'pipe (endchar ?)',
  'literal car' ||,          /* Record to search for... */
  '| literal snowmobile' ||, /* Another record to search for... */
  '| l: lookup',            /* Look for the records */
  '| specs /Primary: / 1',  /* LOOKUP writes matching detail and */
  '| 1-* next',            /* master records to primary output */
  '| terminal',            /* Display records */
  '?',
  'literal truck' ||,      /* Master record for reference */
  '| literal boat' ||,     /* Master record for reference */
  '| literal car' ||,      /* Master record for reference */
  '| l:',                  /* Connect to secondary input & output */
  '| specs /Secondary: / 1', /* LOOKUP writes unmatched detail to */
  '| 1-* next',            /* its secondary output stream */
  '| terminal',            /* Display unmatched detail records */
  '?',
  '|:',                    /* LOOKUP writes unreferenced master */
  '| specs /Tertiary: / 1', /* records to its tertiary output */
  '| 1-* next',            /* in ascending order */
  '| terminal'             /* Display unreferenced master records */
exit 0

```

Figure 189. LOOKUP Stage Command Example: LOOKSTR EXEC

In the first pipeline, LITERAL stage commands are used to create detail records. LOOKUP writes the matching detail and master records to its primary output stream. SPECS prefixes the records with identifying text and TERMINAL displays the records.

In the second pipeline, the reference is created. LITERAL stage commands are used to create master records. These records flow into the secondary input of LOOKUP. The label l makes the connection. That same label also makes a connection to the secondary output stream of LOOKUP. (See “Defining Multistream Pipelines with Secondary Input and Output Streams” on page 17 for more about this type of connection.) LOOKUP writes the unmatched detail records to its secondary output. SPECS prefixes the records with identifying text, and TERMINAL displays them.

In the third pipeline, the label l connects the tertiary output of LOOKUP. LOOKUP writes unreferenced master records to its tertiary output. Again SPECS and TERMINAL are used to display the records.

The following example shows the response from LOOKSTR:

lookstr

```
Secondary: snowmobile
Primary: car
Primary: car
Tertiary: boat
Tertiary: truck
READY
```

LOOKUP writes snowmobile to its secondary output stream because there is no matching reference record. The record car is written twice: the first record is the detail record, while the second is the matching master record. The records boat and truck are unreferenced master records. LOOKUP writes unreferenced master records to its tertiary output stream.

Another example of LOOKUP, this time with the DETAILS operand, is shown in Figure 191. A map is shown in Figure 190. The example, a user-written stage command name VALIDATE, shows the use of column ranges on LOOKUP to identify a search key. With DETAILS, LOOKUP writes matching detail records to its primary output stream, but not the corresponding master record.

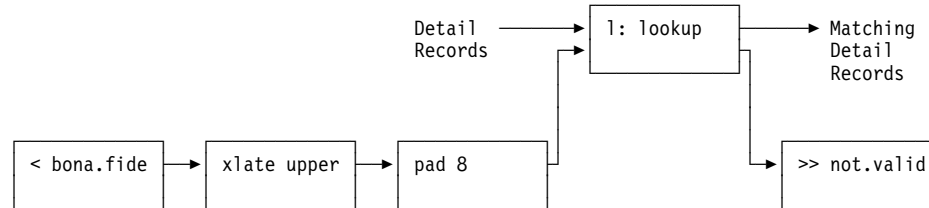


Figure 190. Map of VALIDATE

VALIDATE checks the authorization of the user ID in the first eight columns of the input record against the list of authorized users stored in the data set BONA.FIDE. Unauthorized requests are appended to a log data set.

```

/***** REXX *****/
/* PREFIX.PIPE.REXX(VALIDATE) */
/*      -- Ensure commands are from authorized users      */
/*****
'callpipe (endchar ?)',
  '*: ', /* Read input from calling pipeline */
  '| 1: lookup 1.8 details', /* Search reference */
  '| *: ', /* Pass good ones on */
  '?',
  '< bona.fide', /* Read list of authorized users */
  '| xlate upper', /* Make them uppercase */
  '| pad 8', /* Pad them to 8 characters */
  '| 1:', /* Feed to LOOKUP and get unmatched */
  '| >> not.valid' /* Write unmatched data */
exit rc

```

Figure 191. VALIDATE: Example of LOOKUP

The first pipeline consists of LOOKUP, an input connector, and an output connector. VALIDATE expects its input stream to contain records to be validated. Those that are valid (that is, found in the reference) are written to the output stream. The column range 1.8 is used to define the key to be used in the search. DETAILS is specified to avoid having the master record written to the output stream.

The second pipeline reads the data set BONA.FIDE, prepares the records, and writes those records to LOOKUP's secondary input stream. LOOKUP builds the reference from these records. LOOKUP writes unmatched detail records to its secondary output. These records flow into the >> stage and are appended to the data set NOT.VALID.

In the following example, uppercase user IDs are used because LOOKUP is case-sensitive.

```
pipe < bona.fide | terminal
Bill
Ted
Denise
Mike
READY
pipe literal TED      This should get through. | validate | terminal
TED      This should get through.
READY
pipe literal BOGUS    This should fail. | validate | terminal
READY
pipe < not.valid | terminal
BOGUS    This should fail.
READY
```

Fixing Pipeline Stalls

“Avoiding Pipeline Stalls” on page 20 introduces the term “pipeline stall” and describes circumstances that lead to one. Figure 192 shows one way to fix the stall. A BUFFER stage is added. BUFFER doesn't write any records to its output stream until it has read all the records in its input stream. So, when FANOUT writes a record to its output stream 0, FANIN reads that record. When it writes to stream 1, the BUFFER stage reads the record. After FANOUT writes its last record to stream 1, BUFFER writes the records to its output stream. Because FANIN has, by this time, processed all the records in its input stream 0, it can now process the records supplied by BUFFER on stream 1.

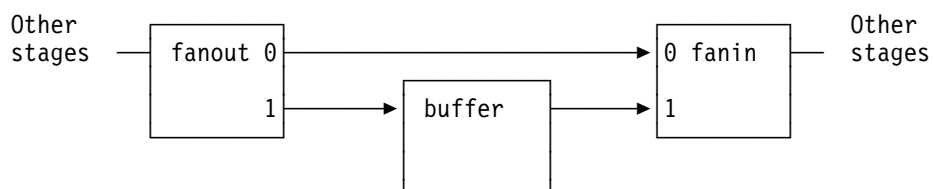


Figure 192. Fixing a Stall with a BUFFER Stage Command

You can also fix the stall by replacing the BUFFER stage in Figure 192 with another stage command called ELASTIC. The ELASTIC stage command works like BUFFER. However, ELASTIC reads only as many records into a buffer as necessary to prevent a stall.

Another solution is to substitute FANINANY for FANIN. FANINANY does not care about the order in which records are delivered to it. Of course, the order of records flowing out of FANINANY is not the same order that FANIN would have provided. So, FANINANY might not be an acceptable solution.

If FANINANY fixes the stall, but delivers the records in the wrong order, you might be able to work around it. You could, in some cases, use SPECS with a RECNO operand to put numbers on the records. After the records are processed and combined by FANINANY, add a SORT stage to put the records in order. Then use another SPECS stage command to remove the record numbers.

Figure 193 shows another common stall. In this case a SPECS stage command contributes to the problem. It expects two records at a time.

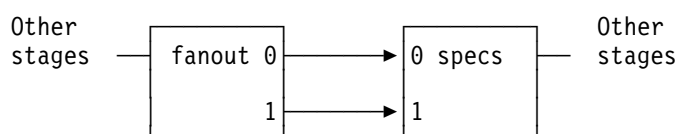


Figure 193. Example Stall Involving a Stage that Needs Multiple Records

In this case, FANOUT writes a record on its output stream 0. It is waiting for its OUTPUT operation to finish. SPECS tries to read records from both of its input streams. Because FANOUT hasn't yet written a record on its output stream 1, SPECS waits. While SPECS waits, FANOUT's OUTPUT operation cannot complete. So, it cannot write a record on its output stream 1. The pipeline is stalled.

Figure 194 shows a solution. The COPY stage command breaks the stall. COPY simply reads a record from its input stream and writes it to its output stream. By doing this, COPY lets FANOUT complete its OUTPUT operation on stream 0. FANOUT can then write a record on its output stream 1. When SPECS starts running, the two records it needs are available.

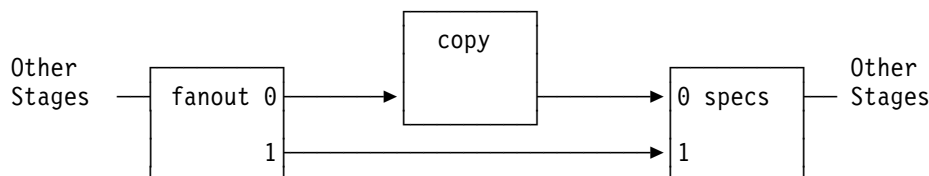


Figure 194. Fixing a Stall with COPY Stage Command

Sometimes, you might be able to fix a stall by using COPY, ELASTIC or a BUFFER stage command. When deciding which one to use, remember that BUFFER holds all the records it reads in storage, ELASTIC holds only as many records in storage as it needs to prevent a stall, while COPY holds only one record. If you want to conserve storage, COPY is the best choice.

BUFFER is not the only stage command that reads all records from its input stream before writing them. SORT, by the nature of its processing, also does this. You may also have some user-written stages at your disposal that buffer records. Assuming you need the functions provided by these stages, they can be used instead of BUFFER to break stalls or to avoid stalls when designing the pipeline.

Although the stalls we have shown involved FANOUT stages, a FANOUT stage is not a prerequisite for a stall. Consider the example in Figure 195. In this case, a LOCATE stage is involved.

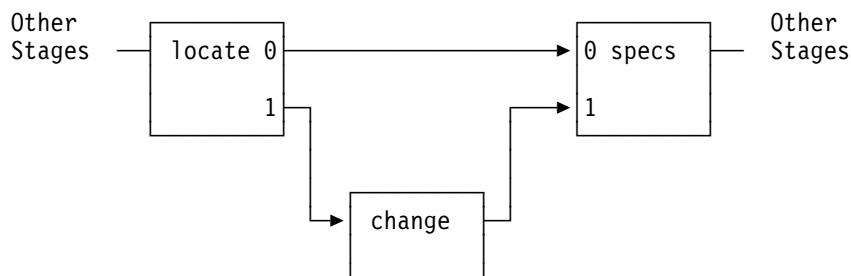


Figure 195. Example Stall Involving a LOCATE Stage

Because of the way LOCATE is connected, only one record at a time is delivered to SPECS. To fix the stall you need to add two BUFFER stages: one between LOCATE and SPECS and the other between CHANGE and SPECS. All records that flow on either stream are held by the BUFFER stages until all the input is read. Then SPECS can read the records from the BUFFER stages two at a time. If one stream has more records than the other, the stream with fewer records will be disconnected from SPECS when all the records are read.

Maintaining the Relative Order of Records

Although the order in which BatchPipeWorks runs stages is unpredictable, sometimes the relative output order of records flowing through multistream pipelines is predictable. This section explains how to write multistream pipelines so the order of the output records is predictable. The following concepts are discussed:

- How each stage of a pipeline runs
- How stages delay the records
- How to predict relative record order.

How Stages of a Pipeline Run

A stage does not run from start to finish once BatchPipeWorks gives it control. Once a stage has written a record to its output stream, that stage is *blocked*, which means it cannot run again until the stage connected to its output stream consumes the record. A stage *consumes a record* when it reads a record from its input stream and removes the record from that input stream. Once a record has been consumed by a stage, it cannot be read again by that stage. A stage is also blocked when it is waiting to read a record, but no records are currently available. This topic is continued in “How BatchPipeWorks Processes a Pipeline Specification” on page 141.

How Stages Delay the Records

Let's consider the following command:

```
PIPE A | B | C
```

If stage A writes a record to its output stream, stage A stops running until the record is consumed by stage B. Therefore, stage B determines when stage A can continue to run. Stage B can process its records in one two ways:

1. It can read a record and remove it from its input stream. This allows stage A to continue running.
2. It can read a record without removing the record from its input stream. This prevents stage A from running.

In other words, there are two methods for writing stages. The first method *delays the records*, meaning a record's progress through the stage is buffered, or held up. This occurs when the output of stage A is consumed by stage B allowing stage A to resume running before stage B writes its output. In the second method, records progress through the stage without being held up or buffered.

To determine whether a stage command delays the records, refer to the usage notes section of the command's description in the *BatchPipeWorks Reference*. Don't confuse the DELAY stage command with delaying the records.

How to Predict Relative Record Order of Records

To maintain the relative order of records in a set of multistream pipeline, the pipeline must:

- Start at one common stage
- Be split into multiple pipelines using only stage commands that do not delay the records
- Contain only stages that do not delay the records
- Be combined into a single stream using a stage that combines multiple streams as records arrive (for example, FANINANY).

By using the process shown in the following examples, you can determine how records flow through a set of pipelines.

Note: The examples assume a specific order of execution of the stages involved. The actual order as determined by BatchPipeWorks may be different. You can trace the PIPE command to see the actual order in which the stages run. Note that a subsequent trace of the same command may show the stages running in a different order.

Example 1 - Not Delaying the Records

Look at the following PIPE command and the map in Figure 196:

```
/* ***** REXX ***** */
/* PREFIX.PIPE.EXEC(NODELAY) */
/* -- Example of an exec that does not delay the records */
/* ***** */
'pipe (endchar ?)',
  '< DELAY.INPUT',          /* Read DELAY.INPUT data set */
  '| l: locate /a/',        /* Find records containing a */
  '| xlate upper',          /* translate records to uppercase */
  '| f: faninany',          /* combine streams back together */
  '| > DELAY.OUTPUT',       /* write result to DELAY.OUTPUT */
  '?',                     /* beginning of second pipeline */
  '|:',                    /* define 2ndary output for locate */
  '| f:',                  /* define 2ndary input for faninany */
```

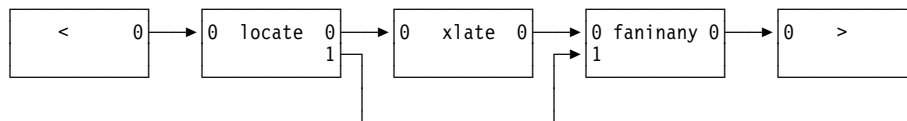


Figure 196. Map of the NODELAY EXEC

Assume that the data set DELAY.INPUT contains the following records:

a1
b2
a3
b4

When you run the NODELAY EXEC, the following takes place:

1. The < stage command reads the first record a1 and writes the record to its output stream. The < stage command is now blocked until its output record is consumed.
2. LOCATE starts running; it finds the record a1 available and processes it. It searches the record for an a and then selects an output stream. The record contains an a, so LOCATE writes the record a1 to its primary output stream. Because LOCATE has not consumed the < stage command's output, < still cannot run. LOCATE is now blocked until its output record is consumed.
3. XLATE starts running; it looks at its input stream and obtains the record a1. XLATE reads the record, translates it to uppercase, and writes the result to its output stream. Because XLATE has not consumed the LOCATE stage's output, LOCATE still cannot run. XLATE is now blocked until its output record is consumed.
4. FANINANY starts running; it finds a record available on its primary input stream. It looks at the record and copies it to its output stream. Because FANINANY has not consumed the XLATE stage's output, XLATE still cannot run. FANINANY is now blocked until its output record is consumed.
5. The > stage command starts running. It looks at its input stream and finds the A1 record. > writes the record to the data set. Because there is nothing attached to its output stream, > writes no output record. > consumes the A1 record from its input stream making FANINANY eligible to run. > then looks for another input record. > cannot run until another record is ready for it.

6. FANINANY resumes, consumes its input, and then looks for another input record. FANINANY cannot run until another record is ready for it.
7. XLATE resumes, consumes its input, and looks for another input record. XLATE cannot run until another record is ready for it.
8. LOCATE resumes, consumes its input, and looks for another input record. LOCATE cannot run until another record is ready for it.
9. < resumes, reads the record b2 , and writes the record to its output stream. The < stage is now blocked until its output record is consumed.
10. LOCATE resumes and it looks at its input stream; it finds the record b2 available and processes it. It searches the record for an a and then selects an output stream. The record does not contain a so LOCATE writes the record to its secondary output stream. LOCATE has not consumed the < stage command's output, so < still cannot run. LOCATE is now blocked until its output record is consumed.
11. FANINANY resumes; it finds a record available on its secondary input stream. FANINANY looks at the record and copies it to its output stream. FANINANY has not consumed LOCATE's output, so LOCATE still cannot run. The FANINANY stage is now blocked until its output record is consumed.
12. > resumes and finds the b2 record. > writes the record to the data set and then to its output stream. > consumes the b2 record from its input stream making FANINANY eligible to run. > then looks for another input record. > cannot run until another record is ready for it.
13. FANINANY resumes, consumes its input, and looks for another input record. FANINANY cannot run until another record is ready for it.
14. LOCATE resumes, consumes its input, and looks for another input record. LOCATE cannot run until another record is ready for it.

At this point, the following records have been written to the output data set:

A1
b2

The process continues for the a3 and b4 records until the output data set contains:

A1
b2
A3
b4

Then the following happens:

1. < attempts to read a record and finds that there are no more records in the data set. < ends causing its output stream to be disconnected. Because this stream is disconnected, LOCATE knows the end of the data set is reached and the stage becomes eligible to run.
2. LOCATE determines that its input stream is disconnected. LOCATE ends causing its output streams to be disconnected. Because these streams are disconnected, the XLATE stage knows the end of the data set is reached and the stage becomes eligible to run.
3. XLATE determines that its input stream is disconnected. XLATE ends causing its output stream to be disconnected. Because this stream is disconnected, the

FANINANY stage knows the end of the data set is reached and the stage becomes eligible to run.

4. FANINANY determines that its input streams are disconnected; it ends causing its output stream to be disconnected. Because this stream is disconnected, the > stage knows the end of the data set is reached and the stage becomes eligible to run.
5. > determines that its input stream is disconnected and ends.

Now the entire set of pipelines has completed processing and the PIPE command ends. Note that every stage command in this example looked at its input record, processed it, wrote it out, and then consumed the input record. None of the stage commands in this example delayed the records.

Once again, remember that the actual order in which the stage commands run is unpredictable, but the order of the resulting output records is the same.

Example 2 - Can Delay the Records

If you have a multistream pipeline that contains a stage command that can delay the records, the order may be unpredictable. For example, look at the following PIPE command and map in Figure 197:

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(CANDELAY) */
/* -- Example of an exec that can delay the records */
/*****
'pipe (endchar ?)',
  '< DELAY.INPUT',          /* read DELAY.INPUT data set */
  '| l: locate /a/',        /* find records containing a */
  '| copy',                 /* delay by one record */
  '| f: faninany',          /* combine records back together */
  '| > DELAY.OUTPUT',       /* write result to DELAY.OUTPUT */
  '?',                      /* start of second pipeline */
  '|:',                     /* define secondary output for locate */
  '| f:'                    /* define secondary input for faninany */

```

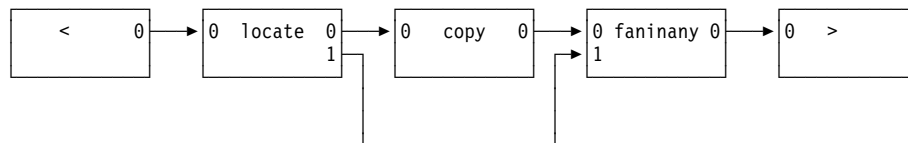


Figure 197. Map of CANDELAY EXEC

Assume that the data set DELAY.INPUT contains the following records:

```

a1
b2
a3
b4

```

When you run the CANDELAY EXEC, the following takes place:

1. The < stage command reads the first record a1 and writes the record to its output stream. The < stage command is now blocked until its output record is consumed.

2. LOCATE starts running; it finds the record a1 available and processes it. It searches the record for an a, and then selects an output stream. The record contains an a, so LOCATE writes the record a1 to its primary output stream. Because LOCATE has not consumed the < stage command's output, < still cannot run. LOCATE is now blocked until its output record is consumed.
3. COPY starts running; it looks at its input stream and obtains the record a1. COPY saves the record a1 and consumes it. At this point LOCATE can run or COPY can continue running.

If LOCATE runs next, it will consume its input enabling < to run. This enables LOCATE to process the b2 record and write the record to its secondary output stream which would cause b2 to arrive at FANINANY before the a1 record.

If COPY continues to run, it will write the a1 record to its primary output stream which causes a1 to be the next record processed by FANINANY.

Because it is unpredictable whether LOCATE or COPY will run, the order of records in the data set DELAY.OUTPUT is also unpredictable.

Processing continues until the set of pipelines completes.

Example 3 - Delaying the Records

In some cases, a multistream pipeline contains stages that delay the records, but the order of the output records can still be predicted. For example, look at the following PIPE command (a map is shown in Figure 198):

```

/***** REXX *****/
/* PREFIX.PIPES.EXEC(DELAY) */
/*      -- Example of an exec that delays the records */
/*****/
'pipe (endchar ?)',
  '< DELAY.INPUT',          /* read DELAY.INPUT data set */
  '| l: locate /a/',        /* find records that contain an a */
  '| join 1',               /* join pairs of records */
  '| f: faninany',          /* combine records back together */
  '| > DELAY.OUTPUT',       /* write result to DELAY.OUTPUT */
  '?',                     /* start of second pipeline */
  'l:',                    /* define secondary output for LOCATE */
  '| f:'                    /* define secondary input for FANINANY */

```

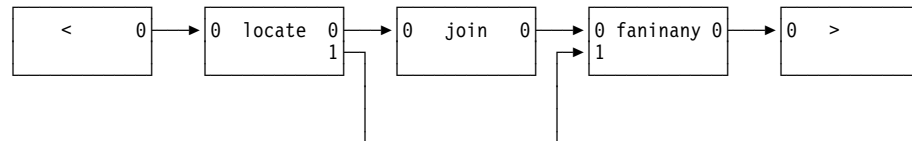


Figure 198. Map of the DELAY EXEC

Assume that the V-format data set DELAY.INPUT contains the following records:

```

a1
b2
a3
b4

```

When you run the DELAY EXEC, the following takes place:

1. < reads the first record a1 and writes the record to its output stream. The < stage command is now blocked until its output record is consumed.
2. LOCATE starts running. LOCATE finds the record a1 available and processes it. It searches the record for an a and then selects an output stream. The record contains an a so LOCATE writes the record a1 to its primary output stream. Because LOCATE has not consumed the < stage's output, < still cannot run. The LOCATE stage is now blocked until its output record is consumed.
3. JOIN starts running. It looks at its input stream and obtains the record a1. Because JOIN needs two input records to build an output record, JOIN saves the contents of the record, consumes the record, and looks for another input record. JOIN can not run until another record is ready for it.
4. LOCATE resumes, consumes its input, and looks for another input record. LOCATE cannot run until another record is ready for it.
5. < resumes, reads the record b2 , and writes the record to its output stream. The < stage is now blocked until its output record is consumed.
6. LOCATE resumes and looks at its input stream. LOCATE finds the record b2 available and processes it. It searches the record for an a and then selects an output stream. The record does not contain a so LOCATE writes the record to its secondary output stream. LOCATE has not consumed the < stage command's output, so < still cannot run. LOCATE is now blocked until its output record is consumed.
7. FANINANY starts running. It finds a record available on its secondary input stream. FANINANY looks at the record and copies it to its output stream. Because FANINANY has not consumed LOCATE's output, LOCATE still cannot run. FANINANY is now blocked until its output record is consumed.
8. The > stage starts running. It looks at its input stream and finds the b2 record. > writes the record to the data set. Because there is nothing attached to its output stream, > writes no output record. > consumes the A1 record from its input stream making FANINANY eligible to run. > then looks for another input record. > cannot run until another record is ready for it.
9. FANINANY resumes, consumes its input, and then looks for another input record. FANINANY cannot run until another record is ready for it.
10. LOCATE resumes, consumes its input, and looks for another input record. LOCATE cannot run until another record is ready for it.
11. < reads the next record a3 and writes the record to its output stream. The < stage command is now blocked until its output record is consumed.
12. LOCATE resumes and it looks at its input stream; it finds the record a3 available and processes it. It searches the record for an a and then selects an output stream. The record contains an a, so LOCATE writes the record to its primary output stream. LOCATE has not consumed the < stage command's output, so < still cannot run. LOCATE is now blocked until its output record is consumed.
13. JOIN resumes running. It finds a record available on its input stream. JOIN combines the record with the a1 record, which it saved, and writes the combined record a1a3 to its output stream. JOIN has not consumed LOCATE's

output, so LOCATE still cannot run. The JOIN stage is now blocked until its output record is consumed.

14. FANINANY resumes. It finds a record available on its primary input stream. FANINANY looks at the record and copies it to its output stream. FANINANY has not consumed JOIN's output, so JOIN still cannot run. FANINANY is now blocked until its output record is consumed.
15. > resumes and finds the a1a3 record. > writes the record to the data set and then to its output stream. > consumes the a1a3 record from its input stream making FANINANY eligible to run. > then looks for another input record. > cannot run until another record is ready for it.
16. FANINANY resumes, consumes its input, and looks for another input record. FANINANY cannot run until another record is ready for it.
17. JOIN resumes, consumes its input, and looks for another input record. JOIN cannot run until another record is ready for it.
18. LOCATE resumes, consumes its input, and looks for another input record. LOCATE cannot run until another record is ready for it.

At this point, the following records have been written to the output data set:

b2
a1a3

The process continues for the b4 record until the output data set contains:

b2
a1a3
b4

Then the following happens:

1. < attempts to read a record and finds that there are no more records in the data set. < ends causing its output stream to be disconnected. Because this stream is disconnected, the LOCATE stage knows the end of the data set is reached and the stage becomes eligible to run.
2. LOCATE determines that its input stream is disconnected. LOCATE ends causing its output streams to be disconnected. Because these streams are disconnected, the JOIN stage knows the end of the data set is reached and the stage becomes eligible to run.
3. JOIN determines that its input stream is disconnected. JOIN ends causing its output stream to be disconnected. Because this stream is disconnected, the FANINANY stage knows the end of the data set is reached and the stage becomes eligible to run.
4. FANINANY determines that its input streams are disconnected. FANINANY ends causing its output stream to be disconnected. Because this stream is disconnected, the > stage knows the end of the data set is reached and the stage becomes eligible to run.
5. > determines that its input stream is disconnected and ends.

Now the entire set of pipelines has finished processing and the PIPE command ends.

Chapter 7. Using BatchPipeWorks in Batch Applications

This chapter contains General-Use Programming Interface and Associated Guidance Information.

You can take advantage of BatchPipeWorks and benefit from its productivity enhancements in a batch environment whether or not the jobs are using BatchPipes. This chapter explains how to include BatchPipeWorks in:

- Batch jobs that are not using BatchPipes; see “Running BatchPipeWorks in a Batch Job”
- Batch jobs that are using BatchPipes; see “Using BatchPipeWorks in Filter Jobs” on page 124.

In a batch environment, you can use any of the stage commands that BatchPipeWorks provides; additionally, you can use the user-written stage commands described in Chapter 9, “Writing Your Own Stage Commands” on page 139.

Running BatchPipeWorks in a Batch Job

To use BatchPipeWorks with batch jobs, include the pipeline specification in an EXEC statement or a DD statement according to the following guidelines:

- If the pipeline is less than 100 characters in length, you can place it in the PARM parameter of the EXEC statement. For details, see “Placing a Pipeline Specification on an EXEC Statement.”
- If the pipeline is longer than 100 characters in length, you must add it to a DD statement. For details, see “Placing a Pipeline Specification on a DD Statement” on page 123.

Note: This book assumes the member containing the BatchPipeWorks module is named PIPE, and that PIPE is in a system library, such as SYS1.SASFPLIB. If your installation named the member that contains the BatchPipeWorks module something other than PIPE, substitute that member name for PIPE on the EXEC statements.

Placing a Pipeline Specification on an EXEC Statement

Once the PIPE member is in a system library, PIPE can be the name of the program on the EXEC statement that the system is to execute. You can pass BatchPipeWorks to the batch environment by executing the PIPE program and passing pipelines through the PARM parameter. The parameter passed to the PIPE program must be in the same format as the string of operands and options on a PIPE command. (The parameter is really a pipeline without the PIPE command). The pipeline starts with the BatchPipeWorks global options and ends with the last stage command. The maximum length of a pipeline submitted on the PARM parameter is 100 characters, including commas.

Figure 199 on page 122 shows an example of providing a pipeline as the PARM parameter on the EXEC statement. This pipeline reads the data set with a ddname of IN, counts the number of records in the data set, and writes the count of the records to the data set with the ddname of OUT:

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM='QSAM IN|COUNT LINES|QSAM OUT'
//IN DD DSN=PSP.PIPES.CNTL(DUP2),DISP=SHR
//OUT DD DSN=PSP.PIPES.CNTL(COPY),DISP=OLD
```

Figure 199. BatchPipeWorks on an EXEC statement (Counting Records)

Figure 200 shows another example of providing a pipeline as a parameter on the EXEC statement. This pipeline reads a data set and writes its contents to two other data sets:

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM='QSAM IN|QSAM OUT2|QSAM OUT1'
//IN DD DSN=PSP.PIPES.CNTL(DUP2),DISP=SHR
//OUT1 DD DSN=ABC.PIPES.CNTL(COPY),DISP=OLD
//OUT2 DD DSN=XYZ.PIPES.BKUP(COPY),DISP=OLD
```

Figure 200. Submitting BatchPipeWorks on an EXEC statement (Making Multiple Copies)

The pipeline is provided as the PARM parameter to the EXEC statement. The pipeline reads the data set with a ddname of IN, and writes the contents of that data set to the data sets with ddnames of OUT1 and OUT2.

Using Multistream Pipelines in a Batch Application

You can also use multistream pipelines on the PARM parameter on the EXEC statement as long as the length of the pipeline is less than 100 characters. For example, if you combine the two previous examples, you can read a data set, count the records, write the number of records to a data set, and copy the records to two other data sets. Figure 201 shows the JCL to do this:

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM='(END ?) QSAM IN|QSAM OUT2| C:COUNT LINES|QSAM OUT1? C:|QSAM OUT'
//IN DD DSN=PSP.PIPES.CNTL(DUP2),DISP=SHR
//OUT DD DSN=ABC.PIPES.CNTL(COUNT),DISP=OLD
//OUT1 DD DSN=ABC.PIPES.CNTL(COPY),DISP=OLD
//OUT2 DD DSN=XYZ.PIPES.BKUP(ORIGINAL),DISP=OLD
```

Figure 201. Multistream BatchPipeWorks on EXEC statement

The pipeline is provided as the PARM parameter to the EXEC statement since it is less than 100 characters in length. This pipeline reads the data set with the ddname of IN and writes its contents to the data set with the ddname of OUT2 and to the COUNT stage command. The COUNT stage command counts the records read, writes the final count to the data set with the ddname of OUT and also writes the records read to the data set with the ddname of OUT1.

Continuing a Line

When the parameters specified on PARM are too long for one line, the line must be continued. Figure 202 shows an example of submitting a pipeline that is longer than one line as a parameter on the EXEC statement, causing the PARM line to be continued:

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM='< dd=DATA|chop str /--/|strip|join * / /|split after string /s
//          ave /|pad 80|chop 80|> dd=PIPEOUT1'
//PIPEOUT1 DD SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//DATA DD *
BatchPipeWorks is fun -- for most people
because you can use pipelines -- as a programming language
to save you time -- for other things
and you can end up the local hero. -- (If that is good for you.)
```

Figure 202. Submitting BatchPipeWorks on EXEC statement - continuing PARM parameter

BatchPipeWorks is fun because you can use pipelines to save you time and you can end up the local hero.

Note that the last character of the pipeline in the PARM parameter ends in column 71 and the pipeline continues in column 16 on the next // statement.

Placing a Pipeline Specification on a DD Statement

When the pipeline is longer than 100 characters, it is not valid with the PARM parameter. The DD statement must read in the pipeline using the RUNPIPE stage command. The RUNPIPE stage command becomes part of the PARM parameter. RUNPIPE reads input records in the same format as the string of operands and options on a PIPE command and issues them as pipelines, capturing any messages they produce. The messages are written to the joblog.

Figure 203 shows a pipeline on a DD statement being executed by a RUNPIPE stage command on the EXEC statement:

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM='< dd=PIPE|CHOP STR /--/|JOIN *|RUNPIPE|joblog'
//OUT1 DD SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//PIPE DD *
    < 'PSP.PIPES.CNTL(TS08)' -- Read records one at a time
    | TAKE 10 -- Keep only the first 10 records
    | SPEC 'top ' 1 -- PUT 'top' first in the output record
    | 1-* NEXT -- Then add on the input record
    | PAD 80 -- Make it at least 80 long for OUTPUT
    | CHOP 80 -- Don't let it be longer than 80
    | JOBLLOG -- Write it to the joblog
    | spec 'bot ' 1 w2-* n -- Place 'bot' in the front of the record
    | > dd=out1 coerce -- Write to OUT1 forcing it to fit
```

Figure 203. BatchPipeWorks on DD statement

Let's examine this example more closely. The pipeline being passed to the PIPE program is defined on the PARM parameter. Since the pipeline is more than 100 characters in length, part of the pipeline is allocated to the DD statement PIPE. When the < dd=PIPE stage is executed, BatchPipeWorks reads in the 9 lines allocated to the ddname PIPE and passes them to the CHOP STR /--/ stage. CHOP STR /--/ eliminates the '--' and any following characters on each line. JOIN * concatenates all lines into a single record. This record, which now consists of < 'PSP.PIPES.CNTL(TS08)' | TAKE 10 | SPEC 'top ' 1 1-* NEXT | PAD 80 | CHOP 80 | JOBLOG | spec 'bot ' 1 w2-* n | > dd=out1 coerce, passes to RUNPIPE. RUNPIPE runs this pipeline and displays the first 116 bytes of any messages in the joblog. Messages longer than 229 characters abend the job.

To write messages longer than the 229 character limit on JOBLOG, write the messages to a data set using the > dd=ddname stage command. If the messages are written to a data set that is fixed length, the COERCE operand will make them fit. Here is an example:

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM='< dd=PIPE|CHOP STR /--/|JOIN *|RUNPIPE|> dd=PIPEMSG coerce'
//PIPEMSG DD SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//OUT1 DD SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//PIPE DD *
    < 'PSP.PIPES.CNTL(TS08)' -- Read records one at a time
    | TAKE 10 -- Keep only the first 10 records
    | SPEC 'top ' 1 -- PUT 'top' first in the output record
    | 1-* NEXT -- Then add on the input record
    | PAD 80 -- Make it at least 80 long for OUTPUT
    | CHOP 80 -- Don't let it be longer than 80
    | JOBLOG -- Write it to the joblog
    | spec 'bot ' 1 w2-* n -- Place 'bot' in the front of the record
    | > dd=out1 coerce -- Write to OUT1 forcing it to fit
```

Figure 204. Submitting BatchPipeWorks on DD statement

The messages written to RUNPIPE's primary output stream are written to the data set allocated to PIPEMSG.

Using BatchPipeWorks in Filter Jobs

Using BatchPipeWorks in a BatchPipes filter job is the same as using BatchPipeWorks in any batch job. You can include the pipeline in an EXEC statement or a DD statement. All of the rules explained in Chapter 7, "Using BatchPipeWorks in Batch Applications" on page 121 apply to using BatchPipeWorks in a filter job. However, the DD statements in your filter jobs obtain data from a BatchPipes pipe or write data to a pipe.

Figure 205 on page 125 shows an existing BatchPipes pipeline. Data flows from one job to the next job.

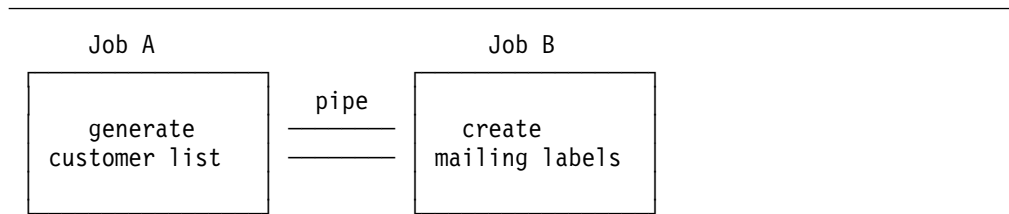


Figure 205. A Typical BatchPipes Pipeline

To intercept the data and perform a selection process, you add a filter job before the mailing labels are created. Figure 206 shows the existing BatchPipes pipeline with the addition of a filter job.

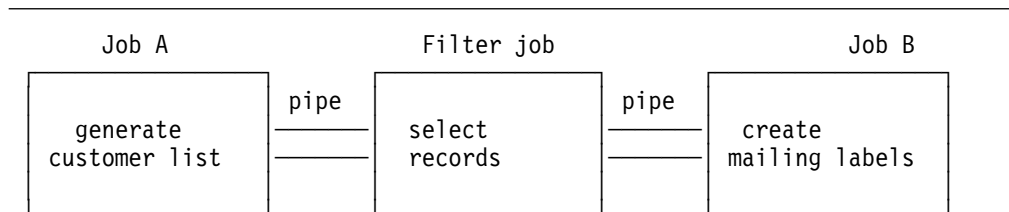


Figure 206. Using BatchPipeWorks in a Filter Job

Example of Selecting Records

Let's say your company maintains a program that generates a customer mailing list. Once a week you run a separate program that generates mailing labels from your customer list. To identify customers who place large orders, you could write a BatchPipeWorks job to select only those records with sales of fifty dollars or more. Figure 207 shows the necessary JCL to test this filter.

```

//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
//  PARM='< dd=PIPE|CHOP STR @/*@|JOIN *|RUNPIPE|> dd=PIPEMSG coerce'
//REXX    DD  DSN=PREFIX.REXX,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//PIPEMSG DD  SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//ORDERS  DD  DSN=PREFIX.ORDER.HISTORY,...
//CUST50  DD  DSN=PREFIX.CUSTOMER.LIST50,...
//PIPE    DD  *
(endchar ? name CUST50  )
|  < dd=orders                /* Get orders information      */
|  orders 50                  /* Select customer if >= $50  */
|  > dd=cust50                /* New list of customers      */
  
```

Figure 207. JCL for BatchPipes BatchPipeWorks Filter

This example uses a user-written stage command called ORDERS. The ORDERS stage obtains information about customer orders such as the total dollar value and selects records with a value of fifty dollars or more. These records are then written to the existing data set CUST50.

A word about two of the JCL statements:

- `//SYSTSPRT DD SYSOUT=*`

This statement is required if the user-written stage or REXX processing needs to issue messages.

- `//REXX DD DSN=PREFIX.REXX,DISP=SHR`

Before you can use a user-written stage command, you must first allocate the data set containing the ORDERS member to a ddname of REXX.

For more information on user-written stage commands see Chapter 9, “Writing Your Own Stage Commands” on page 139.

To send a new catalog to all customers who place large orders of fifty dollars or more, write a filter job to select the orders, match them with the corresponding customer information, and print mailing labels.

Figure 208 uses the previous example and expands it to produce the desired output. The filter reads two input streams (datasets MAILLIST and ORDER.HISTORY). It then finds all orders with a value of \$50 or higher and matches records based on customer number. The merged information is written to the existing data set CATALOG.LIST. The LOOKUP stage writes CATALOG.LIST to its primary output stream.

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM='< dd=PIPE|CHOP STR @/*@|JOIN *|RUNPIPE|> dd=PIPEMSG coerce'
//REXX DD DSN=PREFIX.REXX,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//PIPEMSG DD SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//MAILLIST DD DSN=PREFIX.MAILLIST,...
//ORDERS DD DSN=PREFIX.ORDER.HISTORY,...
//CATALOG DD DSN=PREFIX.CATALOG.LIST,...
//PIPE DD *
(endchar ? name CATALOG ) /* REXX */
| < dd=maillist /* Read in customer list */
| 1:lookup 1.4 detail /* Match customer numbers */
| > dd=catalog /* Write out the equal ones */
? < dd=orders /* Get orders information */
| orders 50 /* Select customer if >= $50 */
| 1: /* Provide to lookup stage */
```

Figure 208. JCL for BatchPipes BatchPipeWorks Filter

Selecting Records Based on Record Content

Suppose you want to test your program with actual data, but your data set is too large to consider for this purpose. Several stage commands let you select records based on the criteria of your choice, as the following examples show.

The first example selects the first five records from the MAIL data set and writes them to the TEST data set. All remaining records are discarded because a secondary output stream of the TAKE stage command is not connected. This allows you to run your existing program with a sample of your real data set without altering the data set in any way. Notice the pipeline is read in from a DD statement

and executed using the RUNPIPE stage command as part of the PARM parameter of the EXEC statement. Also notice the records are taken from the beginning of the input stream. This is because no additional operands are specified. To select records from the end of the input stream, use the LAST operand with the TAKE stage command.

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM='< dd=PIPE|CHOP STR @/*@|JOIN *|RUNPIPE|> dd=PIPEMSG coerce'
//REXX      DD  DSN=PREFIX.REXX,DISP=SHR
//SYSTSPRT  DD  SYSOUT=*
//PIPEMSG   DD  SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//MAILLIST  DD  DSN=PREFIX.MAILLIST,SUBSYS=BP01,...
//TESTLIST  DD  DSN=PREFIX.TEST,SUBSYS=BP01...
//PIPE      DD  *
(endchar ? name CATALOG )
| < dd=maillist                /* Read in paylist data set */
| take 5                      /* Get only the first 5 records */
| > dd=testlist                /* Write out the equal ones */
```

Figure 209. Filter Using TAKE Stage Command

The second example selects data records based on the contents of a particular field or fields. Suppose you need to verify a specific customer in the mailing list. The example uses the LOCATE stage command to create a filter that searches for customer Dr. Karen Frost.

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM='< dd=PIPE|CHOP STR @/*@|JOIN *|RUNPIPE|> dd=PIPEMSG coerce'
//REXX      DD  DSN=PREFIX.REXX,DISP=SHR
//PIPEMSG   DD  SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//MAILLIST  DD  DSN=PREFIX.MAILLIST,SUBSYS=BP01,...
//CUSTNAME  DD  DSN=PREFIX.CUSTNAME.FILE,SUBSYS=BP01...
//PIPE      DD  *
(endchar ? name CATALOG )
| < dd=maillist                /* Read in maillist data set */
| locate (8-22) /Dr. Karen Frost/ /* Search positions 8-22 */
| > dd=custname                /* Write out customer name */
```

Figure 210. Filter Using LOCATE Stage Command

Reformatting or Combining the Contents of Data Records

Suppose the phone company decides to change zip code numbers. The following pipeline changes all existing 45210 zip codes to 45555.

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
//  PARM='< dd=PIPE|CHOP STR @/*@|JOIN *|RUNPIPE|> dd=PIPEMSG coerce'
//REXX      DD DSN=PREFIX.REXX,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//PIPEMSG DD SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//MAILLIST DD DSN=PREFIX.MAILLIST,SUBSYS=BP01,...
//PIPE      DD *
(endchar ? name CATALOG )
| < dd=maillist                /* Read in customer list */
| change /CA 45210/CA 45555/    /* Change the zip code */
| > 'prefix.mail.test'         /* Write out the new list */
```

Figure 211. Changing Data within a Pipeline

The CHANGE stage command reads records from its primary input stream MAILLIST, makes the specified zip code changes, and then writes the records to the existing data set MAIL.TEST.

Notice that a secondary output stream is not defined, therefore, all unchanged records are written to the primary output stream.

Changing the Data Flow within a Job

Using BatchPipeWorks, you can copy your data from one input stream to multiple output streams. Suppose you want to know how many orders each customer placed *while* you are processing the records. Figure 212 uses the FANOUT stage command to copy the incoming records for the SORT COUNT stage to process. The result is written to the PREFIX.TEMP data set and displayed on the terminal.

```
//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
//  PARM='< dd=PIPE|CHOP STR @/*@|JOIN *|RUNPIPE|> dd=PIPEMSG coerce'
//REXX      DD DSN=PREFIX.REXX,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//PIPEMSG DD SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//ORDERS DD DSN=PREFIX.ORDER.HISTORY,DISP=SHR,...
//CATALOG DD DSN=PREFIX.CATALOG.LIST,SUBSYS=BP01,...
//PIPE      DD *
(endchar ? name COUNTER )
| < dd=orders                /* Read in orders */
| f:fanout                  /* Make a copy of the records*/
| > dd=catalog              /* Write out the records */
?f:                        /* Copy of records goes here */
| sort count 1.4            /* Do counting by key field */
| spec /Customer/ 1 11.4 nw /has made/ nw 1.10 nw /order(s)./ nw
| > 'prefix.temp'          /* Save the count in TEMP dsn*/
| joblog                   /* Put count in the listing */
```

Figure 212. JCL for FANOUT Stage Example

```
000001 1234 123456 0023 003 0023.34
000002 1745 123453 0023 001 0025.35
000003 1745 005466 0023 001 0013.45
000004 1679 043678 0023 004 0045.23
000005 1679 034673 0023 003 0089.42
000006 1679 074567 0023 002 0033.44
000007 1892 164324 0023 001 0123.34
```

Figure 213. Input Data Set PREFIX.ORDER.HISTORY

```
000001 Customer 1234 has made      1 order(s).
000002 Customer 1679 has made      3 order(s).
000003 Customer 1745 has made      2 order(s).
000004 Customer 1892 has made      1 order(s).
```

Figure 214. Output Data Set PREFIX.TEMP

Notice FANOUT reads one record at a time from its primary input stream and writes each record to all connected output streams.

Using User-Written Stage Commands in Batch Applications

Before you can use a user-written stage command in a batch environment, you must first allocate the partitioned data set that contains the REXX program you want to run as a user-written stage command. The simplest way to run a user-written stage command in batch is to first allocate the data set containing the user-written stage command with a ddname of REXX. This can be accomplished in two ways. In a batch program you can add the following DD statement, substituting the data set name for *dsname*:

```
//REXX      DD  DSN=dsname,DISP=SHR
```

In TSO, use the ALLOCATE command to allocate the partitioned data set that contains the user-written REXX program that you want to run with a ddname of REXX. Here is an example:

```
ALLOC DDNAME(REXX) DSN('PSP.PIPES.REXX') SHR
```

You can also allocate the partitioned data set that contains the user-written REXX program to a ddname other than REXX. To run this program in batch, you must use the REXX stage command with the ddname as its argument.

Example of Including a User-Written Stage Command

The following example shows a user-written stage command named MCOPY being processed as a stage in a pipeline. For BatchPipeWorks to recognize MCOPY as a user-written stage command, the data set that contains MCOPY as a member was first allocated to REXX on a DD statement. The pipeline allocated to PIPE is read into the PARM parameter and run by the program PIPE.

```

//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM=< dd=PIPE|CHOP STR /--/|JOIN *|RUNPIPE|> dd=pipemsg coerce'
//PIPEMSG DD SYSOUT=*,DCB=(RECFM=V,LRECL=260,BLKSIZE=270)
//REXX DD DSN=PREFIX.PIPES.REXX,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//PIPE DD *
    < dd=INDD -- Read the PDS member
    | MCOPY mcopy args -- Run MCOPY from PREFIX.PIPES.REXX
    | SPEC 1-10 1 -- Keep only the first 10 chars
    | 'XXX' NEXT -- and append 'XXX'
    | PAD 80 -- Make it suitable for F 80 output
    | > dd=OUTDD -- Write it out
    | spec / to temp../ 1 1-* nw -- Add ' to temp..' to each one
    | > 'smith.temp' -- Write it out to SMITH.TEMP
//INDD DD DSN=PREFIX.PIPES.EXEC(SETUP),DISP=SHR
//OUTDD DD SYSOUT=*,DCB=(RECFM=F,LRECL=80,BLKSIZE=80)

```

Figure 215. BatchPipeWorks on EXEC Statement - User-Written Stage Command

Using a User-written Stage Command in a BatchPipes Filter Job

This example is similar to Figure 215, which shows how to run BatchPipeWorks stage commands as a batch job. However, the input and output DD statements change to read the data from a BatchPipes pipe and write the data to a pipe.

```

//JOBNAME JOB parameters
//STEPNAME EXEC PGM=PIPE,
// PARM=< dd=PIPE|CHOP STR /--/|JOIN *|RUNPIPE|> dd=pipemsg coerce'
//PIPEMSG DD SYSOUT=*,DCB=(RECFM=V,LRECL=260,BLKSIZE=270)
//REXX DD DSN=PREFIX.PIPES.REXX,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//PIPE DD *
    < dd=INDD -- Read data from BatchPipes
    | MCOPY mcopy args -- Run MCOPY from PREFIX.PIPES.REXX
    | SPEC 1-10 1 -- Keep only the first 10 characters
    | 'XXX' NEXT -- and append 'XXX'
    | PAD 80 -- Make it suitable for F 80 output
    | > dd=OUTDD -- Write it out to BatchPipes
    | spec / to temp../ 1 1-* nw
    | > 'prefix.temp'
//INDD DD DSN=PREFIX.DATA1,DCB=(...),SUBSYS=BP01
//OUTDD DD DSN=PREFIX.DATA2,DCB=(...),SUBSYS=BP01

```

Figure 216. JCL to a BatchPipes Job with BatchPipeWorks

Chapter 8. Using BatchPipeWorks in TSO/E

This chapter contains General-Use Programming Interface and Associated Guidance Information.

The PIPE command can be entered from the TSO command line or from an exec. PIPE can also be run from a batch job. PIPE accepts one or more pipelines as operands. For example, issue this PIPE command on your terminal:

```
pipe < data.profile | count lines | terminal
```

If you do not have a DATA.PROFILE data set, substitute the name of any existing data set. Be careful to leave a space after <. The number of lines in the data set is displayed. If you make a mistake typing the command, an error message is displayed. Retype the command, correcting the mistake. Figure 217 shows a map of the above pipeline.

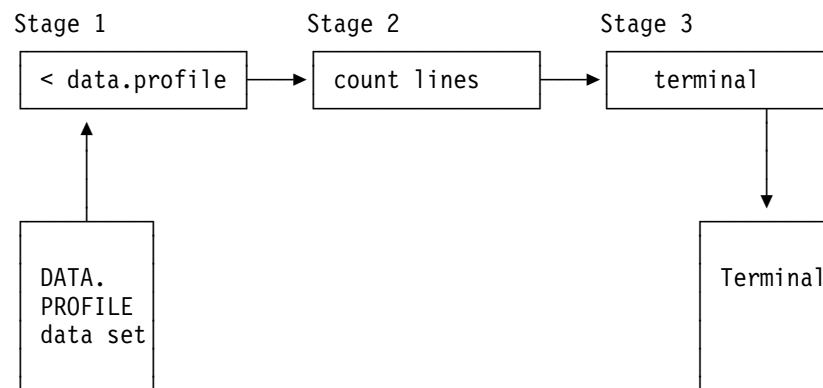


Figure 217. Map of Your Pipeline in TSO

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

< data.profile

Reads DATA.PROFILE and writes each record to its output stream.

count lines

Tallies the count of records in DATA.PROFILE and writes a single record containing the tally to its output stream.

terminal

Either reads from your terminal or writes to it, depending on its position in the pipeline. As used here, TERMINAL reads records from its input stream and displays them on the terminal. Only one record is in its input stream; it contains the count of the lines in DATA.PROFILE.

Using BatchPipeWorks in REXX Execs

The PIPE command can be run from within a REXX exec, just like any other TSO command. This section describes several different uses of BatchPipeWorks within REXX execs.

Saving a BatchPipeWorks Pipeline for Reuse

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

```
/***** REXX *****/
/* PREFIX.PIPE.EXEC(WORDS) */
/* counts the number of words in DATA.PROFILE. */
/*****/
'pipe < data.profile | count words | terminal'
exit
```

The PIPE command reads the data set named DATA.PROFILE, counts the number of words, and displays that number. PIPE is a TSO command, so it should be enclosed in single or double quotation marks as shown. To run the pipeline, run the exec explicitly by using the EXEC command followed by the data set name and the exec keyword operand, or implicitly by entering the member name. You can run an exec implicitly only if the partitioned data set that contains it was allocated to a system file. For more information on running an exec, see *TSO/E REXX/MVS User's Guide*. For example, if the PREFIX.PIPE.EXEC partitioned data set is allocated to file SYSEXEC, you can run the previous exec by specifying: words. You can also run the exec by specifying: exec 'prefix.pipe.exec(words)' exec.

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

```
/***** REXX *****/
/* PREFIX.PIPE.EXEC(WORDS) */
/* counts the number of words. */
/*****/
parse arg datasetid
'pipe <' datasetid ' | count words | terminal'
exit
```

The exec reads an argument and substitutes that argument for the variable datasetid in the PIPE command, just as it would with any other TSO command in an exec. If MY.DATA is specified, the WORDS exec reads records from the MY.DATA dataset. The number of words are counted and then displayed to the terminal.

Continuing BatchPipeWorks on Several Exec Lines

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


```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(WORDS) */
/*      -- counts the number of words */
/*****
parse arg datasetid
'pipe <' datasetid,
  | count words',
  | terminal'

```

The lines in the example are indented to improve readability.

To continue a REXX string, enclose the string in single quotation marks and put the comma after the ending single quote. In the example, `datasetid` is not enclosed in quotation marks because it is a REXX variable (not part of a string). REXX replaces the continuation character with a blank when it interprets the lines. If you do not want REXX to put a blank between the lines when it interprets them, use the REXX concatenation symbol (`||`):

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(CONTIN) */
/*      -- Continuation without an intervening blank */
/*****
'pipe',
  'literal Hello' || ,
  | terminal'

```

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

```
'pipe literal Hello| terminal'
```

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

See the *TSO/E REXX/MVS Reference* for more about the REXX continuation character.

Using BatchPipeWorks As Part of an Exec

BatchPipeWorks can also be used to do tasks within a larger exec. Several device drivers are provided that read and write exec variables from a pipeline. After the data is in the pipeline, you can use filter stage commands to change that data. Most experienced exec writers start using BatchPipeWorks by substituting a PIPE command for the EXECIO command. For example, the following exec fragment uses the EXECIO command to read the data set TEST.DATA into a stem variable (LINES.):

```

:
'ALLOCATE DD(TESTDD) DSN(TEST.DATA) SHR'
'EXECIO * DISKR TESTDD (FINIS STEM LINES.)'
:

```

To do that with a pipeline, use:

```

:
'pipe < test.data | stem LINES.'
:

```

STEM, described on page 87, is a device driver that puts the contents of the pipeline into a stemmed array.

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

```

:
:
EXECIO * DISKR TESTDD (FINIS STEM LINES.'
call mysort                /* Call a subroutine to sort LINES.n    */
:

```

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

```

:
:
'pipe < test.data | sort | stem LINES.'
:

```

The SORT stage command, described in “Sorting Records (SORT)” on page 74, sorts the records. Another popular use of BatchPipeWorks in execs is to simplify the code needed to get command results. For example, the following code fragment lists data sets and invokes the REXX exec MYDSN for each data set in the list:

```

/* Code fragment */
:
address tso
z = outtrap("dsns.")          /* TRAP OUTPUT                */
"LISTCAT"                     /* ISSUE LISTCAT COMMAND      */
z = outtrap("off")            /* UNDO TRAP */
do i = 2 to dsns.0            /* DO EACH LINE OF OUTPUT    */
    "EXEC PIPE.EXEC(MYDSN) '"DSNS.I"' EXEC" /* RUN EXEC WITH DSN */
end

```

If the data set containing the MYDSN member is not concatenated to a system file, you can use a single PIPE command to do the same thing:

```

/* Code fragment using BatchPipeWorks */
:
"pipe",
  "tso listcat",              /* Issue a LISTCAT command */
  "| DROP 1"                  /*                          */
  "| specs /EXEC PIPE.EXEC(MYDSN) '/ 1", /* Prepare the EXEC        */
  "    1-* next",             /* command to be           */
  "    '/ EXEC/ next",        /* run                     */
  "| tso",                    /* Pass it to TSO          */
  "| terminal"                /* Display any responses   */

```

If the data set containing the MYDSN member is concatenated to a system file (SYSEXEC), you can use the following single PIPE command to do the same thing:

```
/* code fragment using pipelines */
"pipe",
  "tso LISTCAT",                      /* Issue LISTCAT command */
  "| specs /MYDSN/ 1",                /* Prepare the EXEC */
  "      1-* nextword",              /*      command */
  "| tso",                            /* Run it */
  "| terminal"                        /* Write result to terminal */
```

Note: To concatenate to SYSEXEC the PREFIX.PIPE.EXEC data set which contains the member MYDSN, use an allocate command similar to the following:

```
/* TO SETUP MY EXEC LIBS */
"ALLOCATE FILE(SYSEXEC) DATASET(",
  "'ISP.PHONE.CLISTS'",              /* system files */
  "'SYS1.ISP.ISPEXEC'",              /*      " */
  "'RELO.QMF.DB2M.DSQEXECE'",        /*      " */
  "'SYS1.EOY.SEOYCLIB'",             /*      " */
  "'PREFIX.PIPE.EXEC'",              /* User EXECs */
  ") SHR REUSE BLKSIZE(9000)"
```

The first stage is a TSO stage command. The TSO stage command passes its operand to TSO for execution. In this example, the operand is a LISTCAT command. TSO writes the response lines from LISTCAT to its output stream. For each data set listed, the SPECS stage command builds a record containing the words EXEC PIPE.EXEC(MYDSN) EXEC or MYDSN, followed by the data set name. SPECS writes these records to its output stream. The second TSO stage command reads the records written by SPECS and passes them to TSO. TERMINAL displays any messages or responses produced by MYDSN.

When writing REXX execs, treat PIPE as you would treat any other TSO command; you can enter many PIPE commands in a single REXX exec. It is also valid to call a REXX exec from a pipeline even if that exec contains other PIPE commands. For example, suppose a REXX exec named TESTER contains a PIPE command. It is valid to call TESTER from another PIPE command, as follows:

```
pipe tso tester | terminal
```

Return Codes in BatchPipeWorks

Some stages of the pipeline give a return code when they end, but BatchPipeWorks returns only one return code — the most serious return code from all the stages in the pipeline. (Any negative return code is more serious than any positive return code.)

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

The following example shows a test for a nonzero return code after the PIPE command:

```
/* Code fragment using batchpipeworks */
:
'pipe',
  'tso LISTCAT',          /* Issue a LISTCAT command */
  '| specs /MYDSN/ 1 1-* nextword', /* Build a MYDSN command */
  '| tso',                /* Pass it to TSO */
  '| terminal'            /* Display any responses */
if rc=0 then              /* Test for return code */
  say 'Return code=' rc 'from PIPE.'
```

Using BatchPipeWorks to run TSO Commands

BatchPipeWorks has several stage commands that interface with TSO. For instance, the TSO command LISTCAT displays all cataloged data sets with your signon as the first qualifier. TSO writes the response to your terminal screen when you issue the command directly to TSO, or the response can be captured and processed in a pipeline with a TSO foreground interface stage command.

There are several TSO foreground interface stage commands in BatchPipeWorks. This section introduces two of them.

TSO and COMMAND are useful stage commands that run TSO commands. The difference between the two stage commands is that the TSO stage command captures the command response and writes it to the pipeline, and the COMMAND stage command displays the command response at the terminal.

TSO Stage Command

The TSO stage command runs TSO commands and writes the response to its output stream. For instance, suppose you want to save the output of a TSO LISTCAT command in an existing data set named CATLG.DATA. You would enter:

```
pipe tso listcat | > catlg.data
```

This pipeline has two stages:

```
tso listcat
```

Runs the LISTCAT command. Instead of displaying the LISTCAT response, TSO writes the response to its output stream.

```
> catlg.data
```

Writes whatever is in its input stream to the data set CATLG.DATA. (The > stage command writes data from the pipeline to an existing data set.)

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

```
pipe tso listcat | > catlg.data | terminal
```

COMMAND Stage Command

To run a TSO command from a pipeline *without* having the response records written to the pipeline, use the COMMAND stage command. When processed by COMMAND, TSO commands run as though they were entered from the TSO environment or with a REXX ADDRESS TSO instruction.

The COMMAND stage command also issues TSO commands. It passes the command to TSO for execution as if the command were invoked from the TSO environment. For instance, to display the output of a TSO LISTCAT command, you would enter:

```
pipe command LISTCAT
```

Putting MVS Command Results in REXX Variables

REXX programmers can use the PIPE command to put TSO command responses directly into stemmed arrays, as shown in Figure 218:

```
/****** REXX ******/
/* PREFIX.PIFE.EXEC(STEMRSP) */
/*      --Put command response into a stemmed array      */
/*******/
'pipe tso listcat',          /* Execute a LISTFILE */
'| stem CATLG.'             /* Put results in FNAME. */
if rc=0 then
  do i=1 to CATLG.0
    /* Other data set processing. */
  end
```

Figure 218. *STEMMED ARRAY: Placing Command Responses in*

The list of data sets is assigned to the CATLG. stem variable. The variable catlg.0 contains the number of lines the STEM stage reads from its input stream. Note that PIPE, described on page 79, is a TSO command, so it should be enclosed by single quotation marks.

Executing Pipeline Records as Commands

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

Figure 219 shows an example in which the TSO stage command executes two commands. One of the commands is specified as an operand (send 'This one first.' user(*)). It executes first. The other command is issued in the pipeline by the LITERAL stage command.

```
pipe literal send 'This is second.' user(*) | tso send 'This one first.' user(*)
This one first. YOURID
This is second. YOURID
READY
```

Figure 219. Executing Multiple Commands

Including BatchPipeWorks in TSO Batch Jobs

You can also process a pipeline as part of a TSO job running in batch. However, in batch, terminal input and full screen functions do not have meaning. To include a pipeline in a TSO batch job, the TSO program, IKJEFT01, must be processed on the EXEC statement. The input to IKJEFT01 is the pipeline that you want to run. The pipeline is allocated to a DD statement with a ddname of SYSTSIN. The pipeline should be in the same format that you would use to run the pipeline in TSO from your terminal. This format is different than the format you use when running a pipeline in a batch job that is not TSO. In TSO batch, the pipeline includes the leading PIPE command. In non-TSO batch the leading PIPE command is omitted. Use a plus sign (+) or minus sign (-) as the TSO line continuation character if you need it.

Figure 220 shows a pipeline, allocated with a ddname of SYSTSIN, being passed to the IKJEFT01 program which will execute this pipeline as a TSO job.

```
//JOBNAME JOB parameters
//STEPTSO EXEC PGM=IKJEFT01,REGION=4000K
//SYSTSIN DD *
PIPE (ENDCHAR ?) +
| literal 1 2 3 4+
| D: FANOUT +
| terminal +
?D: +
| split +
| terminal
/*
//SYSTSPRT DD SYSOUT=*
```

Figure 220. BatchPipeWorks running in TSO batch

The SYSTSPRT DD statement contains the output from the background job. It indicates where the output from the TERMINAL stage in the pipeline is presented.

Chapter 9. Writing Your Own Stage Commands

This chapter contains General-Use Programming Interface and Associated Guidance Information.

When you need to do something that can't be done with the BatchPipeWorks stage commands, you can write your own. While stage commands that you write can do any function you want, most of them will filter pipeline data in some way. You use the user-written stage commands in the same way you use the BatchPipeWorks stage commands.

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

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

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

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

User-Written Stage Command Concepts

Writing a stage command is similar to writing any other REXX-language program. Your use of the REXX language is not restricted; you can use all keyword instructions and all functions from within the stage command. The significant difference between stage commands and other REXX-language programs is that stage commands also interact with BatchPipeWorks. If you use the REXX SAY command to write output to the terminal, add an ALLOCATE statement to your REXX exec:

```
ALLOCATE f(SYSTSPRT) DS(*)
```

Also, to allow input from your exec, add the following statement:

```
ALLOCATE f(SYSTSIN) DS(*)
```

Your stage command interacts with BatchPipeWorks in two ways:

- By executing *pipeline subcommands*
- By passing a return code to BatchPipeWorks on exit.

Pipeline Subcommands: Several subcommands are covered in this chapter:

- READTO — reads a record from an input stream and consumes the record. See page 144.
- OUTPUT — writes a record to an output stream. See page 144.
- PEEKTO — looks at a record in an input stream without removing it from the stream, as READTO does. See page 145.
- SHORT — copies all remaining records in the input stream directly to the output stream. See page 145.
- STAGENUM — returns in variable RC a number indicating the position of your stage command in the pipeline. See page 147.
- SELECT — selects the input and output stream to be processed by subsequent subcommands. See page 149.
- MAXSTREAM — returns the number of the highest stream defined in the PIPE command. See page 150.
- STREAMNUM — validates a stream number. See page 152.
- SEVER — disconnects the current stream. See page 153.

Additionally, the chapter covers two subcommands that allow you to invoke pipelines:

- CALLPIPE — runs a *subroutine pipeline*. See page 153.
- ADDPIPE — adds a pipeline to the set of running pipelines. See page 154.

The PIPE command sets up its own subcommand environment, and subcommands in the stage command interact with BatchPipeWorks. The subcommands give a return code in variable RC. The return codes are the only communications you receive from BatchPipeWorks, so it is important for your stage commands to examine them.

Return Code on Exit: When your stage command completes processing, it gives a return code to BatchPipeWorks. By default, the return code is zero. To set a return code other than zero, specify the desired code on a REXX EXIT instruction, as you do in a regular exec. If your stage command detects an error, it can finish and report the error by using a nonzero return code on an EXIT instruction.

If the pipeline is run from a REXX exec, BatchPipeWorks gathers return codes from all stages, returning only one to REXX. It selects the return code to pass to REXX as follows:

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

Return codes for each subcommand are described in *BatchPipeWorks Reference*.

How BatchPipeWorks Processes a Pipeline Specification

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

How a Pipeline Runs

You can tell how a pipeline runs by examining the interactions between BatchPipeWorks and the stage commands. Suppose you write a stage command that reads records from its input stream, reverses the order of the characters in the records, and writes the changed records to its output stream. (This function is similar to the REVERSE stage command.) Here is the pseudo code to show how it might be written:

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

Figure 221 shows a REXX program stored as the member REVIT in the PREFIX.PIPE.REXX partitioned data set. This REXX program is an implementation of the above pseudo code.

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

Figure 221. REVIT: A Simple User-Written Stage Command

To run the REXX program in REVIT as a user-written stage command, you must first allocate the partitioned data set that contains the REVIT member to a ddname of REXX. In TSO you use the following ALLOCATE command:

```
ALLOCATE DD(REXX) DSN('PREFIX.PIPE.REXX') SHR
```

If you are using the user-written stage command in a batch job, allocate the data set containing REVIT with a DD statement (and allocate the SYSTSPRT file in case REXX needs to issue messages).

```
//REXX DD DSN=PREFIX.PIPE.REXX,DISP=SHR
```

The following example shows how to use REVIT:

```
pipe < test.data | terminal
Test 1
Test 2
Test 3
Ready;
pipe < test.data | revit | terminal
1 tseT
2 tseT
3 tseT
Ready;
```

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

Remember that your stage command may not run from start to finish. Whenever possible, your stage commands should process a record at a time.

How a Pipeline Ends

A pipeline ends when all of its stage commands finish processing. The stage commands can end in any order. A stage command ends when it:

- Completes its function, or
- Detects an error, or
- Detects that one or more of its streams are disconnected, or
- Detects that there is no more data to read from a device (for device drivers only).

Once a stage command ends, the streams that were connected to it become disconnected. This starts a chain reaction that emanates from the ended stage. Soon return codes of 12 spread throughout the pipeline as stages end, and eventually the PIPE command ends. To determine how a certain stage command reacts when its output stream becomes disconnected, see the command description in *BatchPipeWorks Reference*.

An Example of a User-Written Command — HOLD

Figure 222 on page 143 shows a simple stage command that reads records from its input stream and writes them to its output stream without modifying them. The name of the stage command is HOLD, which is stored as a member of the partitioned data set named PREFIX.PIPE.REXX.

```

/***** REXX *****/
/* PREFIX.PIPE.REXX(HOLD) */
/* -- Copy input stream to output stream */
/*****/

/* We get control from BatchPipeWorks */
signal on error /* Set up error handling */
do forever
    'readto record' /* Subcommand to read a record */
    'output' record /* Subcommand to write a record */
end
error:
if rc=12 then rc=0 /* RC=12 is a normal condition */
exit rc /* Return to BatchPipeWorks */

```

Figure 222. *HOLD: A Simple User-Written Stage Command*

The user-written stage command **HOLD** provides the same function as the stage command **COPY**. We are using **HOLD** for illustrative purposes only. If you need a stage command that provides this function, use **COPY**.

Except for the **READTO** and **OUTPUT** commands, **HOLD** is like any other program. **READTO** and **OUTPUT** are subcommands. **READTO** reads one record from its input stream into a variable. The argument of **READTO** is the *name* of a variable. In our example, the record is assigned to the REXX variable *record*. Notice that the variable name is *within* the quotation marks (').

OUTPUT writes a record to the pipeline. Unlike **READTO**, the argument on **OUTPUT** is a string, not the name of the variable. In this example, the contents of *record* are written.

The REXX **SIGNAL** instruction sets up the error handling. It causes REXX to watch for commands that give nonzero return codes. If a command gives a nonzero return code, REXX branches to the **ERROR** label and continues processing.

HOLD loops, reading and writing pipeline data, until a nonzero return code occurs. Then REXX branches to the **ERROR** label. Eventually, the program gets a return code of 12 on **READTO** or **OUTPUT**.

The **IF** instruction following the error label returns a return code. The **IF** instruction checks for a return code of 12. Return code 12 is a normal condition, so the return code of 12 is changed to a 0. The **EXIT** instruction ends the filter and passes the return code back to **BatchPipeWorks**.

Using Your Stage Command

To use a stage command in a pipeline, allocate the data set the command resides in with ddname **REXX**. You can allocate the data set that contains **HOLD** using a **DD** statement or in TSO with the **ALLOCATE** command:

```

//REXX DD DSN=PREFIX.PIPE.REXX,DISP=SHR

```

or

```

ALLOCATE DD(REXX) DSN('PREFIX.PIPE.REXX') SHR

```

Use your user-written stage command in a pipeline the same way you use BatchPipeWorks stage commands. For example, to use HOLD in a pipeline:

```
pipe < test.file | terminal
Test 1
Test 2
Test 3
READY
pipe < test.file | hold | terminal
Test 1
Test 2
Test 3
READY
```

If the name of your stage command is the same as the name of a BatchPipeWorks stage subcommand, use the REXX stage command to run yours. For example, suppose you write a stage command named LOCATE that is not case sensitive. To use your LOCATE instead of the BatchPipeWorks LOCATE:

```
pipe < test.data | rexx locate /Mixed!/ | terminal
```

Pipeline Subcommands

This section describes common subcommands you can use to write your own stage commands. On the subcommands, you can identify input and output streams by keywords and in some instances by stream numbers or names.

READTO Subcommand

READTO reads one record from an input stream. The contents of the record are placed in a variable whose name you specify as the sole READTO operand. A record is read and discarded if you execute READTO without an operand.

The variable RC is set with the return code from READTO. Return code zero means that the function is performed as requested: the variable has a value. Return code 12 means the input stream is disconnected. A likely cause is that all records have already been read. When return code 12 is given, the variable is dropped so you do not inadvertently use obsolete data. Use the REXX SIGNAL ON NOVALUE instruction to detect dropped variables.

Important: The *name* of the variable is the argument to READTO. Be sure to put it inside the quoted string.

OUTPUT Subcommand

OUTPUT writes one record to an output stream. OUTPUT accepts a string as an argument. It writes the string to the output stream. Put the string variable outside of the quoted string. The first blank following the word OUTPUT is not considered to be part of the data. Any other leading blanks are written to the output stream.

The variable RC is set with the return code from OUTPUT. The return code is zero if the line is read by the following stage. Return code 12 is set if the output stream is not connected.

PEEKTO Subcommand

Each time you read a record with READTO, a new record is stored in a REXX variable. Sometimes it is convenient to be able to peek at a record but be able to read it again later with READTO, or leave it in the input stream for a subroutine pipeline to read. (Subroutine pipelines are described later in this chapter.) This is exactly what PEEKTO does. Except for leaving the record in the input stream so it can be read again, PEEKTO behaves the same as READTO.

PEEKTO without a variable name sets the return code to zero when there is a record available to read. Like READTO, return code 12 means the input stream is disconnected.

You can process the record peeked at, and then execute READTO to remove the record from the input stream. PEEKTO is useful in writing stages that do not delay the record.

Important: The *name* of the variable is the argument to PEEKTO. Be sure to put it inside the quoted string.

SHORT Subcommand

Often your stage command processes its input records to end-of-data and then exits. Sometimes you may wish to copy the remaining pipeline records unmodified to the output stream. Although you can write a loop like the one in HOLD, there is an easier way: the SHORT subcommand.

SHORT causes BatchPipeWorks to reconnect streams so that they bypass your stage. Consider the following PIPE command, which uses stage commands named A, B, and C. Your stage command is stage B.

```
pipe A | B | C
```

When your stage command (B) runs the SHORT subcommand, BatchPipeWorks connects the output stream of stage A directly to the input stream of stage C. The remaining records bypass your stage (stage B).

After you execute SHORT, the input and output streams are no longer available to your stage command. Because your stage command has not yet ended, however, BatchPipeWorks will still give control to your stage command (eventually). At that time you can, for instance, do clean-up processing, but you can no longer execute READTO or OUTPUT to process the stream.

If PEEKTO is the last subcommand issued before the SHORT subcommand, the record seen is included in the remainder of the input stream which is passed on.

The HOLD user-written stage command could be more efficiently written as shown in Figure 223.

```
/*          Tight version of HOLD program          */
'short'                                /* Copy input to output */
exit RC                                /* Return                */
```

Figure 223. Modified HOLD: Using SHORT Subcommand

This version of HOLD is as redundant as the one we saw previously, but faster! In BatchPipeWorks, a SHORT is the shortest and fastest path between two stages because the remaining records bypass the stage doing SHORT.

SHORT is also useful when processing headers. Suppose, for example, you are processing input that consists of a header and a body. Your filter needs to modify some lines in the header but not in the body and you want your output to contain both.

Using READTO and OUTPUT subcommands, read and process your header. When you detect the body section (or end of the header) you can execute SHORT. That takes care of copying the body to the output of your filter without modifying the data.

Figure 224 on page 147 shows an example stage command that uses SHORT after processing a header. In this case, the stage command, AUTHOR, changes the header on a SCRIPT data set.

The following is an example of a SCRIPT data set containing a header:

```
.***** Start of Header *****
.*
.* Security classification:  Company Secret
.*
.* Title:  BatchPipeWorks User's Guide
.*
.* Author:  Joe Smith
.*
.* Filename:  MYBOOK
.*
.***** End of Header *****
:h1.BatchPipeWorks Basics
:p.
BatchPipeWorks lets you solve big problems by combining small programs.
It lets you do work that would otherwise require someone to
write a new program.
Often you get the result you need
with a single TSO command: PIPE.
:
```

The header consists of SCRIPT comments, which are records beginning with the string .*. One of the header records contains the keyword Author:. It is this record we must replace. The end of the header is indicated by a comment containing the string End of Header. For simplicity, the AUTHOR user-written stage command assumes that there is an author record.

```

/***** REXX *****/
/* PREFIX.PIPES.REXX(AUTHOR) */
/*      -- Change the author in the header of a SCRIPT data set      */
/*****/
signal on error

authrec='.* Author:  Tim A. Shenka'      /* Set new author record */
do forever
    'readto record'                      /* Read a pipeline record */
    uprec=translate(record)               /* Fold it to uppercase */

    if pos('AUTHOR:',uprec)>0 & left(uprec,2)='.*' then do
        'output' authrec                  /* Write the new author record */
        leave                             /* Leave DO FOREVER */
    end
    else 'output' record                  /* Otherwise, write record to output */
end

'short'                                  /* Copy remaining records to output */

error:
if rc=12 then rc=0
exit rc

```

Figure 224. AUTHOR: Using SHORT Subcommand to Process a Header

AUTHOR reads records and copies them to its output stream until it finds the record containing the author. Then it replaces the author record with the new record and leaves the DO FOREVER loop. Here is an example of how to use AUTHOR in a PIPE command:

```

pipe < edition1.script | author | > edition2.script
READY

```

Note: Remember that you must first allocate the data set that contains AUTHOR to ddname REXX for this pipeline to work.

STAGENUM Subcommand

Occasionally it might be useful for a stage command to know its position in the pipeline, also called the *stage number*. The first stage has stage number 1, the second stage has stage number 2, and so on.

The stage number is made available to a stage command as the return code of the STAGENUM subcommand. STAGENUM has no parameters; after it is executed, the variable RC contains the stage number.

Most stage commands are filters that do the same thing in all positions of the pipeline. This is true for all filter stage commands. Some device drivers do different things when they are first in a pipeline. With STAGENUM, your stage command can do different things depending on its position.

Figure 225 on page 148 shows a stage command that adds numbers. It looks at the first blank-delimited word (or token) on each record. If the word is a valid number, it adds the number to the REXX variable SUM. Otherwise, it ignores the

record. When all input records have been read, ADD writes a record to the pipeline containing the sum.

ADD uses STAGENUM to detect whether it is being used as the first stage. If it is, ADD exits with a return code of 24.

You can easily modify ADD so it doesn't care what stage it is. Simply remove the STAGENUM subcommand and the next line that checks the return code. In this case, using ADD as the first stage causes it to write a record containing a zero to the pipeline. (The first time READTO is executed it gives a return code of 12 because there is nothing connected to the input stream of ADD.)

```

/***** REXX *****/
/* PREFIX.PIPEREXX(ADD) */
/*      -- Add all numbers appearing as first token on input */
/*      records. Ignore any records that do not have a */
/*      valid number as the first token. */
/*****/

'stagenum'          /* What stage are we? */
if rc=1 then exit 24 /* First stage? Exit with RC=24 */

sum=0               /* Initialize SUM */
do forever
  'readto i'        /* Read a record from the pipeline */
  if rc=0 then do   /* Check the return code */
    if rc=12 then 'output 'sum /* No more input? Write sum. */
    leave          /* We're already leaving; ignore RC */
  end              /* In any case, leave the loop. */
  val=word(i,1)    /* Pull off the first token */
  if datatype(val)='NUM' then /* Is it a valid number? */
    sum=sum+val    /* Yes, so add it to SUM */
end

exit 0

```

Figure 225. ADD: Using STAGENUM Subcommand

The following example shows a PIPE command that uses ADD:

```

pipe literal 3 | literal invalid | literal 4 | add | terminal
7
READY

```

Note: Remember that you must first allocate the data set that contains ADD to ddname REXX for this pipeline to work as shown.

SELECT Subcommand

READTO, OUTPUT, and PEEKTO subcommands act on the currently selected stream. By default, BatchPipeWorks uses stream 0 (the primary input and output stream) for READTOs, OUTPUTs, and PEEKTOs. To select a different stream to be used on following subcommands, use the SELECT subcommand.

SELECT selects the stream identified by a keyword (INPUT, OUTPUT, or BOTH) followed by the stream number or name. Figure 226 shows how to use SELECT. It shows a user-written stage command that reads records from its primary input stream and writes those records to a primary and a secondary output stream (in a manner similar to FANOUT).

```

/***** REXX *****/
/* PREFIX.PIFE.REXX(MYFANOUT) */
/*      -- A simplified FANOUT to show SELECT */
/*****/
signal on error

'select input 0' /* Select primary input stream, which is default */

do forever
    'peekto record' /* Copy a record from input stream 0 */
    'select output 0' /* Select output stream 0... */
    'output' record /* ...write the record to output stream 0 */
    'select output 1' /* Select output stream 1... */
    'output' record /* ...write the record to output stream 1 */
    'readto record' /* Read a record from input stream 0 */
end

error:
if rc=12 then rc=0
exit rc
```

Figure 226. Example of the SELECT Subcommand: MYFANOUT

The select input subcommand in MYFANOUT is not really needed because, by default, the primary input stream (stream 0) is selected. We put it in the program so you could see how to write one. To select a secondary input, specify the number 1. The following example tests MYFANOUT through a REXX exec that uses both output streams:

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(MYTEST) */
/* -- Exec to test MYFANOUT */
/*****/
'pipe (endchar ?)',
  'literal Test data',          /* Create record */
  '| a: myfanout',              /* Feed it to our stage command */
  '| specs /Output Stream 0/ 1', /* Put identifier in output record */
  ' 1-* nextword',              /* Put data in output record */
  '| f: faninany',              /* Combine streams */
  '| terminal',                 /* Display results */
  '?',
  'a:',
  '| specs /Output Stream 1/ 1', /* Put identifier in output record */
  ' 1-* nextword',              /* Put data in output record */
  '| f:'
exit rc

```

Figure 227. Exec to Test MYFANOUT user-written stage command

The following is an example run of the REXX exec named MYTEST:

```

mytest
Output Stream 0 Test data
Output Stream 1 Test data
READY

```

If you don't use both output streams, when MYFANOUT tries to select output stream 1, the SELECT subcommand gives a return code of 4. This SELECT return code indicates that the selected stream is not defined. The SIGNAL instruction causes control to pass to the label error, and MYFANOUT ends with a return code of 4.

MAXSTREAM Subcommand

MAXSTREAM returns the number of the highest stream defined. The number is given in the return code from MAXSTREAM. The number is the highest number allowed in a SELECT command for the input or output stream (specify INPUT or OUTPUT as an operand).

Figure 228 on page 151 shows an improved MYFANOUT. MAXSTREAM detects the number of defined output streams. A record is written to each of these streams.

```

/***** REXX *****/
/* PREFIX.PIPE.REXX(MYFANOUT) */
/*      -- Write record to all defined output streams */
/*****/

'maxstream output' /* Get number of defined output streams */
outcount=rc        /* Save that number */

signal on error     /* Now intercept nonzero return codes */

'select input 0' /* Select primary input stream, which is default */

do forever
  'readto record' /* Read a record from input stream 0 */
  do i=0 to outcount /* Write a record to each output stream */
    'select output' i /* Select output stream ... */
    'output' record /* ...write the record to it */
  end
end

error:
if rc=12 then rc=0
exit rc

```

Figure 228. Example of the MAXSTREAM Subcommand: MYFANOUT

MAXSTREAM gives you the number of the highest stream defined, but it does not indicate whether the streams are connected. In the above example, OUTPUT gives a return code of 12 if the selected stream is disconnected. To test the improved MYFANOUT user-written stage command, use a REXX exec like this one:

```

/***** REXX *****/
/* PREFIX.PIPES.EXEC(MYTEST1) */
/*      -- Test the improved MYFANOUT      */
/*****/
'pipe (endchar ?)',
  'literal Test',
  '| a: myfanout',
  '| specs /Output stream 0/ 1',
  '| 1-* nextword',
  '| f: faninany',
  '| terminal',
  '?',
  'a:',
  '| specs /Output stream 1/ 1',
  '| 1-* nextword',
  '| f:',
  '?',
  'a:',
  '| specs /Output stream 2/ 1',
  '| 1-* nextword',
  '| f:',
  '?',
  'a:',
  '| specs /Output stream 3/ 1',
  '| 1-* nextword',
  '| f:'
exit rc

```

Here is an example run of the MYTEST1 exec:

```

mytest1
Output stream 0 Test
Output stream 1 Test
Output stream 2 Test
Output stream 3 Test
READY

```

STREAMNUM Subcommand

The STREAMNUM subcommand tests whether a given stream is defined. If the stream is defined, the stream number is given in the return code. Otherwise, STREAMNUM gives a negative return code. For operands, write a keyword indicating an input or output stream (INPUT or OUTPUT) followed by the stream number, stream name, or an asterisk (*). An asterisk means the currently selected stream. You can also use this subcommand to validate a stream number.

SEVER Subcommand

Use the SEVER subcommand to disconnect the currently selected stream. If the disconnected stream was connected to another stage previously, the connection is restored. SEVER requires one operand (INPUT or OUTPUT) that identifies the stream you want to sever.

SEVER is often used after ADDPIPE. You can use it to restore previously connected streams after you have processed some records on the stream connected with ADDPIPE. SEVER can also be used to sever a stream after you have finished processing it. Some selection stages, for instance TAKE, use SEVER when they switch from the primary output to the secondary output. This reduces the probability of a stall. For example, suppose your stage command finished processing input records. There may be more records in the input stream, but you do not want to process them. Instead of issuing a SEVER, your stage continues with other processing. Because you haven't severed the connection, the stage that was supplying records on your input stream waits indefinitely the next time it writes a record, increasing the probability of a stall.

When you issue a SEVER, however, BatchPipeWorks knows you no longer intend to read records on that stream. It can give a return code of 12 to any stage that was waiting for you to read its output. By freeing the stage behind you, you may free a stage ahead of you, and in a roundabout way, save your own stage from stalling.

When control returns from your stage, BatchPipeWorks severs all streams still connected to that stage.

CALLPIPE Subcommand

You can write subroutine pipelines that use multiple input and output streams. To do so, specify the ENDCHAR option on the CALLPIPE command and separate the pipelines with end characters. (CALLPIPE has the same options as the PIPE command.) When using multiple input streams, tell CALLPIPE which one to use. Use the full format for connectors:

```
*.input.0:    <-- Identifies primary input stream 0
*.output.0:   <-- Identifies primary output stream 0
```

Figure 229 on page 154 shows the REXX program LOCDEPT. LOCDEPT expects personnel records as input. It finds all records beginning with the names of department members and writes those records to its primary output stream. Records that do not begin with names of department members are written to the secondary output stream.

```

/***** REXX *****/
/* PREFIX.PIPES.REXX(LOCDEPT) */
/* -- Locate personnel records for department members */
/***** */
'callpipe (endchar ?)',
  '*input.0:', /* Connect primary input stream */
  '| a: find Smith_||', /* Look for Smith */
  '| d: faninany', /* Combine all department members */
  '| *.output.0:', /* Write department records to output 0 */
  '?',
  'a:',
  '| b: find Jones_||', /* Look for Jones */
  '| d:', /* To FANINANY */
  '?',
  'b:',
  '| c: find Davis_||', /* Look for Davis */
  '| d:', /* To FANINANY */
  '?',
  'c:',
  '| *.output.1:' /* Rejects to output stream 1 */
exit rc

```

Figure 229. Example of Multistream Subroutine Pipeline: LOCDEPT

The following is a sample run. Records of department members are prefixed with an asterisk (*).

```

pipe < salary.data | terminal
Miles          25000
Smith           36500
Jones           22000
Davis           44199
Bush            32072
READY
pipe (end ?) < salary.data | a:locdept | specs /*/ 1 1-* 2 | terminal ? a: | terminal
Miles          25000
*Smith          36500
*Jones           22000
*Davis           44199
Bush            32072
READY

```

Another use of the CALLPIPE subcommand is in “CALLPIPE Subcommand” on page 153.

ADDPPIPE Subcommand

The ADDPIPE subcommand adds a pipeline to the set of executing pipelines. At first glance, this seems similar to what CALLPIPE does, but there are two important differences between ADDPIPE and CALLPIPE.

- First, the stage command issuing ADDPIPE continues to run in parallel with the new pipeline. When CALLPIPE is used, on the other hand, the stage issuing the CALLPIPE waits until the CALLPIPE subcommand ends.

- Second, ADDPIPE expands the kinds of connections you can make between the new pipeline and the existing pipeline. CALLPIPE lets you specify a connector at the beginning or the end of the pipeline. One or both connectors can be omitted. ADDPIPE supports these connections and many others.

By using the connectors effectively, you can have your stage command remap the surrounding pipeline. This lets you use BatchPipeWorks where you would otherwise have to use REXX instructions.

ADDPIPE Format

ADDPIPE accepts a pipeline as an operand. There isn't anything special about ADDPIPE that prevents you from using other subcommands (such as READTO, OUTPUT, or CALLPIPE) in the same stage command. The new pipeline can be joined to the pipeline that called your stage command by using connectors.

Connectors can be used at either end or both ends, or they can be omitted. Unlike CALLPIPE, an input or an output connector can be specified at either end of the pipeline or both ends. As always, though, connectors cannot be in the middle of the pipeline. To avoid confusion, specify the full format of the connector (*.input: and *.output:). A stream number can also be specified (for example, *.input.0:), but it is often omitted. As always, the stream number defaults to 0. Because input and output connectors can be used at either end of the pipeline, there are nine possible combinations:

1. addpipe b | c
2. addpipe b | c | *.input:
3. addpipe *.output: | b | c
4. addpipe *.input: | b | c
5. addpipe b | c | *.output:
6. addpipe *.input: | b | c | *.input:
7. addpipe *.output: | b | c | *.output:
8. addpipe *.input: | b | c | *.output:
9. addpipe *.output: | b | c | *.input:

The spacing in the above examples is for clarity. Fictitious stage names are used to keep the examples simple and to focus your attention on the general capabilities that ADDPIPE provides rather than on specific problems.

Each connection possibility lets you redraw the surrounding map in a different way. In the following sections, we discuss each of these connection variations. Some connection variations are more useful than others. We'll point them out along the way.

Another thing to remember is that in some variations the original connection (that is, the original pipeline map) can be restored by executing a SEVER subcommand. In some cases the original connections cannot be restored, as the following section describes.

ADDPIPE Connections

In our discussion of the connection variations, we'll be referring to the PIPE command shown in Figure 230. That command consists of three fictitious stage commands: A, Z, and D. Stage command Z is actually a user-written stage command from which we'll be executing ADDPIPE subcommands. The other maps in this section show how the original map, in Figure 230, is changed when the ADDPIPE subcommand runs.

pipe A | Z | D

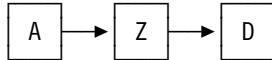


Figure 230. The Original Pipeline

Variation 1: The first variation is an ADDPIPE subcommand that does not have connectors, as in Figure 231. The ADDPIPE subcommand is issued from the Z stage command.

addpipe B | C

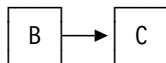
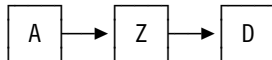


Figure 231. ADDPIPE Map: B | C

In this case, the stage commands in the ADDPIPE subcommand operate independently of the original pipeline. Stage commands B and C start running at the same time A, Z, and D run. The order in which the stage commands start running is unpredictable. The stage command that issued the ADDPIPE command (stage Z) could start before the stages it added with ADDPIPE (stages B and C). In fact, there is no way for stage command Z to tell when stage commands B and C have finished.

The return code from ADDPIPE indicates whether the stages commands have been added successfully to the set of running stages. Nonzero return codes indicate syntax errors in the ADDPIPE subcommand itself. They do not indicate whether the stages ran successfully. (Remember, control may return to your stage command before the added stages are finished.)

If an added stage command ends with a nonzero return code, the return code is reported by the original PIPE command that called your stage command.

Figure 232 on page 157 shows an example of this ADDPIPE variation. The stage command expects data set names in its input records. It uses the data set name in an ADDPIPE command to make an uppercase copy of the data set. Because records are likely to flow through the BACKUP stage command faster than the data sets can be copied, several copy operations may be active at the same time.

```

/***** REXX *****/
/* PREFIX.PIFE.REXX(BACKUP) */
/* -- Make copies of the data sets named in the input records */
/*****

signal on error

do forever
  'readto record'          /* Read record containing a data set name*/
  parse var record dsn .    /* Parse the data set name */
  "addpipe",               /* Add a pipeline... */
  "<" dsn,                 /* ...read the data set */
  "| xlate upper",         /* ...translate to uppercase */
  "| >" dsn".backup"       /* ...write it to a BACKUP data set */
                          /* .....assumes the new DSN <= 44 chars */
  'output' record          /* Write data set name to output stream */
end

error:
if rc=12 then rc=0
exit rc

```

Figure 232. ADDPIPE Example: BACKUP user-written stage command

In the following PIPE command, BACKUP makes uppercase copies of all data sets that have TEMP as a qualifier but do not have BACKUP as a qualifier. The names of the data sets are displayed by TERMINAL.

```

pipe tso listcat|not chop after ./|locate /TEMP/|nlocate /BACKUP/|backup |terminal
TEMP
TEMP1
TEMP2
TEMP3
READY

```

Variation 2: The next connection variation is shown in Figure 233. The original pipeline is altered such that the output from stage command C is connected to the input to stage command Z.

```
addpipe B | C | *.INPUT:
```

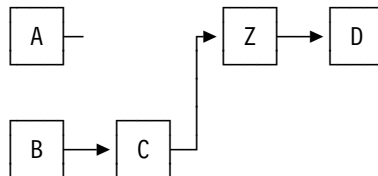


Figure 233. ADDPIPE Map: B | C | *.INPUT:

Figure 234 on page 158 shows an example of this variation. SECPARM writes a record containing a security notice to the output stream. Then it copies all other records in its original input stream to its output stream. SECPARM reads a parameter data set that contains one record to determine what security record should be written. While there are several ways to do this, ADDPIPE is used here.

```

/***** REXX *****/
/* PREFIX.PIPE.REXX(SECPARM) */
/*      -- Add header comment with security notice      */
/*****

/* Add a pipeline to read parameter data set */
'addpipe < parm.data | xlate upper | *.input.0:'

/* Process the record in the parameter data set */
'readto record' /* Read the record from the ADDPIPE stages */

select
when pos('CONFIDENTIAL',record)>0 then
    'output /* This exec is company confidential*/'
otherwise;
    'output /* This exec is unclassified */'
end

'sever input' /* Now sever ADDPIPE connection and restore previous */
'short'      /* Copy records from original connection to output */
exit rc

```

Figure 234. ADDPIPE Example: SECPARM user-written stage command

The stage commands added by ADDPIPE read the data set PARM.DATA, translate the record to uppercase, and then feed the record to SECPARM's input stream. An appropriate security classification is selected and OUTPUT writes the security record to SECPARM's output stream.

How can SECPARM read the record written by XLATE? Remember that the stages added by ADDPIPE run concurrently with SECPARM itself. ADDPIPE alters the map of the pipeline, so the stages start running just as stages run in any other pipeline. For instance, assume that SECPARM starts running before the stages added by ADDPIPE. When SECPARM executes a READTO subcommand, BatchPipeWorks gets control. The map of the pipeline has already been changed by ADDPIPE. So, BatchPipeWorks knows that the output stream of XLATE UPPER is connected to the input stream of SECPARM. BatchPipeWorks also knows it must start running the stages added by ADDPIPE to satisfy the READTO. This is the same as the processing discussed in "How a Pipeline Runs" on page 141.

It's important to remember that ADDPIPE changes the map of the pipeline when it is executed. When reading execs containing ADDPIPE subcommands, draw the map of the pipeline. Use the generic maps in this section as guides.

After the parameter record has been read, we want to copy all of the records in the *original* input stream to SECPARM to the output stream. A SHORT subcommand will do it, but first we have to reconnect the original input stream, depicted in Figure 230 on page 156. BatchPipeWorks does not automatically restore the original connections when the stages added by ADDPIPE end. To restore the original input stream, we execute a SEVER INPUT subcommand.

The SEVER INPUT subcommand disconnects the current input stream. The current input stream happens to be the output from the XLATE stage command that ADDPIPE added. BatchPipeWorks remembers any previous connections. It uses

a *stack* in case several ADDPIPE commands were executed. When a connection is severed, BatchPipeWorks restores the next one on the stack, if any. In the example, the only input stream connection on the stack is the original connection. BatchPipeWorks restores this connection, and our original map is restored.

Finally, a SHORT subcommand is executed, and SECPARM ends. An example run of SECPARM follows:

```
pipe < parm.data | terminal
CONFIDENTIAL
READY
pipe < legumes.script | secparm | terminal
/* This exec is company confidential */
Peas
Bush beans
Pole beans
Lima beans
READY
```

Variation 3: The next variation involves an output connector, as Figure 235 shows.

```
addpipe *.output: | B | C
```

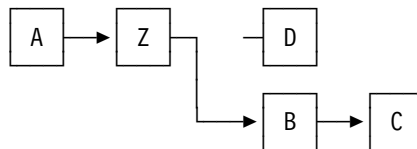


Figure 235. ADDPIPE Map: *.OUTPUT: | B | C

As the map shows, the output stream from stage command Z is connected to the input stream of stage command B. The output is diverted from stage command D. Figure 236 shows an example named TRACER.

TRACER accepts a string of text as an operand. It appends the string of text, along with the date and time, to a data set named TRACER.LOG. Then it copies all the records in its input stream to its original output stream (that is, to stage command D).

```

/***** REXX *****/
/* PREFIX.PIFE.REXX(TRACER) */
/* -- Write text to TRACER.LOG */
/*****/
parse arg text
'addpipe *.output.0: | >> tracer.log'/* Add stages, change connections*/
'output' date() time() text /* Write record to TRACER.LOG */
'sever output' /* Restore original connection */
'short' /* Copy records */
exit rc
```

Figure 236. ADDPIPE Example: TRACER

The ADDPIPE subcommand connects the primary output stream of the TRACER user-written stage command to the primary input stream of the >> stage command. When the OUTPUT subcommand is executed, its record becomes the input to the >> stage command.

After the tracer record is written, TRACER copies the input records to the original output stream. The SEVER OUTPUT subcommand restores the original output connection (which is the only one in the stack of output connections). The map of the pipeline reverts to the map shown in Figure 230 on page 156. The SHORT subcommand copies the original input stream to the original output stream.

Notice that it was not necessary to execute a SEVER subcommand for the input stream. The original input stream was not changed by ADDPIPE.

The following example shows a PIPE command that uses TRACER and a PIPE command that displays the log.

```
pipe < salary.data | tracer Need to verify Smith's salary | terminal
Miles                25000
Smith                 36500
Jones                 22000
Davis                 44199
Bush                  32072
READY
pipe < tracer.log | terminal
20 Jan 1994 16:05:19 Need to verify Smith's salary
READY
```

Variation 4: In some connection variations, it is not possible to reconnect to the original input stream. Consider the next variation, which is shown in Figure 237.

```
addpipe *.input: | B | C
```

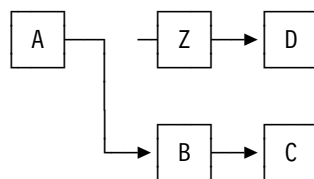


Figure 237. ADDPIPE Map: *.INPUT: | B | C

In this variation, A's output stream is diverted to stages B and C. Once the new connection is established, you cannot sever it and return to the old connection. The stages B and C run concurrently with your stage command (Z). If you were able to sever the connection after ADDPIPE executes, you would have no way of knowing how many records stages B and C processed, if any. You wouldn't even know if the stages still existed. So, severing the input stream in this variation is not permitted. Because Z cannot process its original input stream, this connection variation is one of the least useful.

We've already covered most of the ADDPIPE concepts. The remaining variations truly are variations on the same concepts.

Variation 5: The connection variation in Figure 238 has a problem similar to one in the previous variation. The ADDPIPE subcommand connects the output stream of stage command C to the input stream of stage command D. The output stream of stage command Z is disconnected.

```
addpipe B | C | *.output:
```

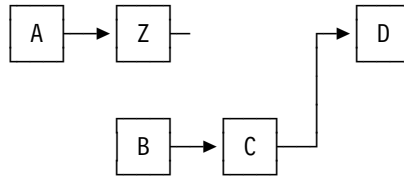


Figure 238. ADDPIPE Map: B | C | *.OUTPUT:

In this variation, the output connection established by ADDPIPE cannot be severed. Stage Z does not know how many records stage C has written, and Z cannot predict when it will start running. If Z could sever the output connection, the processing status of C and D would be unpredictable. Consequently, a sever is not allowed.

Variation 6: This variation is far more useful, as Figure 239 shows, than the previous two. It lets you insert stages before your stage command.

```
addpipe *.input: | B | C | *.input:
```

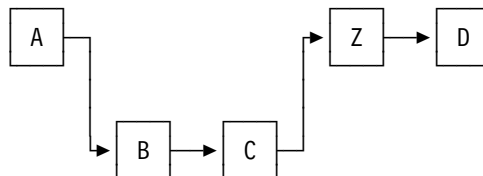


Figure 239. ADDPIPE Map: *.INPUT: | B | C | *.INPUT:

By using ADDPIPE to insert stages, you can do more work in BatchPipeWorks instead of in REXX. For example, the following code fragment selects records containing the string NY and sorts them. Then the preprocessed records are read by READTO:

```

:
:
'addpipe *.input: | locate /NY/ | sort | *.input:'
do forever
  'readto record'
  /* Add REXX instructions here for tasks that BatchPipeWorks cannot do */
  'output' record
end
:

```

Severing the input connection is not allowed. If a sever were allowed, it would not be possible to predict how many records would be preprocessed before the SEVER subcommand is executed.

Variation 7: Another useful variation is shown in Figure 240. It lets you insert stages after your stage command.

```
addpipe *.output: | B | C | *.output:
```

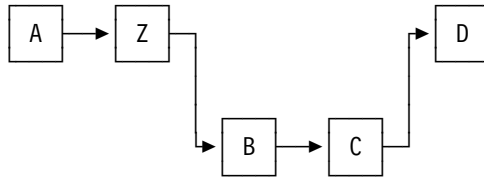


Figure 240. ADDPIPE Map: *.OUTPUT: | B | C | *.OUTPUT:

The stages process the records that your stage command writes to its output stream. Even though the stages act as a post-processor, execute the ADDPIPE subcommand *before* you write records you want to post-process. Otherwise, the records you write before the ADDPIPE command will be processed by stage D (as defined in the original map):

```

:
'output' record      /* Write a record to be processed by stage D    */
'addpipe *.output: | xlate upper | *.output:' /* Change map                */
'output' record      /* Write a record to be processed by XLATE    */
:

```

The connection made in this variation cannot be severed. (You would not be able to tell how many records were processed by the ADDPIPE stages.)

Variation 8: The connections in Figure 241 are similar to those used in CALLPIPE when both connectors are specified. In effect, you substitute the ADDPIPE stages for stage Z. With CALLPIPE, however, the original connections are automatically restored in some cases. Because ADDPIPE stages run concurrently, the original connections cannot be restored. You could process some records with READTO and OUTPUT subcommands, and then let stages B and C process the remainder.

```
addpipe *.input: | B | C | *.output:
```

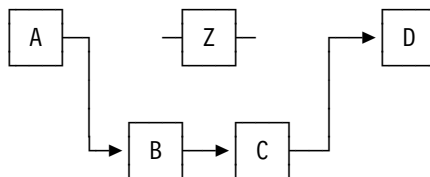


Figure 241. ADDPIPE Map: *.INPUT: | B | C | *.OUTPUT:

This variation provides another way for us to do a SHORT:

```
addpipe *.input: | *.output:
```

Variation 9: The last connection variation is most unusual. The output stream of stage Z is connected to the input stream of stage B. The output of C is connected to the input of Z. We have created a loop—a stall is possible. You can sever the connections made by this ADDPIPE (before a stall, of course).

```
addpipe *.output: | B | C | *.input:
```

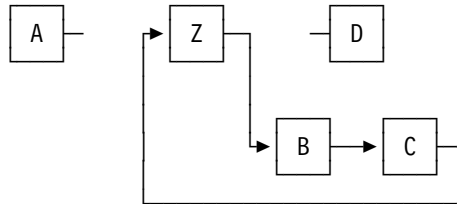


Figure 242. ADDPIPE Map: *.OUTPUT: | B | C | *.INPUT:

Processing Arguments

You can code stage commands to process arguments that are specified when the stage command is invoked. Use the REXX PARSE ARG instruction to get these arguments. Figure 243 on page 164 shows an improved AUTHOR user-written stage command. The original AUTHOR is in Figure 224 on page 147. This version of AUTHOR accepts the name of the author as an argument. The name is used to create the new author record. The new and changed statements are **highlighted**.

```

/***** REXX *****/
/* PREFIX.PIPE.REXX(AUTHOR) */
/*      -- Change the author in the header of a SCRIPT data set      */
/***** */
signal on error
parse arg name                /* Get the name */
if name='' then name='Anonymous' /* Handle missing argument */

authrec='. * Author:  '||name    /* Build new author record */
do forever
  'readto record'              /* Read a pipeline record */
  uprec=translate(record)      /* Fold it to uppercase */

  if pos('AUTHOR:',uprec)>0 & left(uprec,2)='. *' then do
    'output' authrec           /* Write the new author record */
    leave                     /* Leave DO FOREVER */
  end
  else 'output' record         /* Otherwise, write record to output */
end

'short'                        /* Copy remaining records to output */

error:                          /* Error routine and exit */
if rc=12 then rc=0
exit rc

```

Figure 243. AUTHOR: Processing Arguments

To change the author record in data set EDITION1.SCRIPT and have the result written to EDITION2.SCRIPT, you would enter:

```

pipe < edition1.script | author Ward E. Guy | > edition2.script
READY

```

An Example of a Stage Command — TITLE

This section presents an example stage command named TITLE that adds a title before every 20 lines of input. The TITLE filter stage command can be used just as any other filter in a pipeline:

```

pipe < legumes.script | title Legumes: | terminal
Legumes:

Peas
Bush beans
Pole beans
Lima beans
READY

```

TITLE, in Figure 244 on page 165, reads the input given to it from the pipeline. It writes a title (given as an argument to it) followed by 20 lines, and then another title, and so on.

```

/***** REXX *****/
/* PREFIX.PIPE.REXX(TITLE) */
/*      -- Output a title every 20 lines */
/***** */
parse arg title_line          /* Get input arguments */

'readto input_line'           /* Read first record into variable */
do line_count=0 while rc=0    /* Loop while there are records */
  if line_count//20 = 0 then do /* 20-line boundary? */
    'output 'title_line       /* Yes- write the title line... */
    'output '                 /* write a blank line */
  end
  'output 'input_line         /* Write the input record */
  if rc~=0 then leave         /* Check RC from OUTPUT */
  'readto input_line'        /* Read the next pipeline record */
end
if rc=12 then rc=0            /* Ignore 12, which means EOF */
exit rc

```

Figure 244. TITLE: A User-Written Stage Command Example

The WHILE clause on the DO instruction checks the return code after READTO. When all the records are read, READTO gives a return code of 12 and ends the loop. Notice that the return code is checked after the last OUTPUT pipeline subcommand, but not after the first two. The reason is that once one OUTPUT fails, following OUTPUTs will also fail. So, we save some code by checking only the last OUTPUT.

TITLE also shows how to write a blank record to the output stream. Look at the second OUTPUT subcommand. Two blanks follow the OUTPUT keyword before the single quote. The first blank is not part of the string that OUTPUT writes, but the second blank is. Therefore, OUTPUT writes a record containing a single blank to the output stream.

Using CALLPIPE to Write Subroutine Pipelines

The CALLPIPE subcommand lets your stage command run another pipeline to process the records in the input stream. For example, Figure 245 on page 166 shows a stage command that makes all records flowing through it 80 bytes in length.

```

/***** REXX *****/
/* PREFIX.PIFE.REXX(FIXED) */
/*      -- Make all records 80 bytes in length */
/***** */
signal on error      /* Set up error handling */
do forever
  'readto in'        /* Subcommand to read a record */
  'output' left(in,80) /* Pad or chop records as necessary */
end
error:
if rc=12 then rc=0    /* RC=12 is a normal condition */
exit rc              /* Return to BatchPipeWorks */

```

Figure 245. *FIXED: Using READTO and OUTPUT Subcommands*

The CHOP and PAD stage commands offer a simpler way to do the same thing. The CALLPIPE subcommand enables their use, as Figure 246 shows:

```

/***** REXX *****/
/* PREFIX.PIFE.REXX(FIXED) */
/*      -- Make all records 80 bytes in length */
/***** */
'callpipe *: | chop 80 | pad 80 | *:'
exit rc

```

Figure 246. *FIXED: Using CALLPIPE Subcommand*

Rather than use READTO and OUTPUT to process each record, we use a CALLPIPE subcommand to process all of the records. The operand on CALLPIPE is called a *subroutine pipeline*. The pairs of asterisks and colons ([:]) are *connectors*. They must be in stages by themselves as shown. The connector at the beginning of the subroutine pipeline is known as the *input connector*. The connector at the end is known as the *output connector*. A connector cannot be used in the middle of the subroutine pipeline.

The input and output connectors connect the subroutine pipeline to the pipeline that called your stage command. When both connectors are used, CALLPIPE, in effect, inserts a new section of pipeline in the pipeline that called your stage command. By making these connections, all records remaining in the stage command's input stream flow through the subroutine and are written to the stage command's output stream.

In the following PIPE command, for example, the output of the TSO stage is connected to the input of the FIXED stage. The output stream of FIXED is connected to the input stream of the > stage:

```
pipe tso listcat all | fixed | > dataset.list
```

Figure 247 on page 167 shows a diagram of the pipeline before FIXED begins to run.

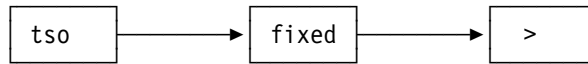


Figure 247. Diagram of Original Pipeline

When the CALLPIPE subcommand runs, however, the output stream of the TSO stage is connected to the input stream of CHOP. And, the output stream of PAD is connected to the input stream of the > stage, as shown in Figure 248.

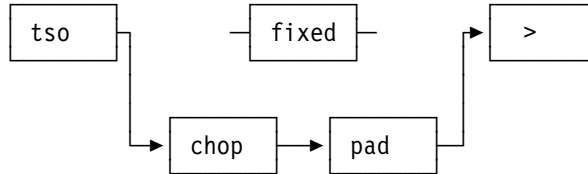


Figure 248. Diagram of Pipeline When CALLPIPE Is Running

CALLPIPE returns when all stages of the new pipeline have ended. The return code from CALLPIPE is the worst of the return codes from the stages in the subroutine pipeline (like the PIPE command). When CALLPIPE finishes, BatchPipeWorks restores the original connections so that the map of the pipeline is as shown in Figure 247. Then FIXED executes a REXX EXIT instruction and ends.

This leads us to an important point about CALLPIPE. When CALLPIPE runs, BatchPipeWorks gets control (as it does for all subcommands) and adds the stages in the CALLPIPE command to the set of stages it is already running and makes the requested connections. Records flow through the stages in the usual manner. But, and this is the important point, your stage command itself does not resume execution until CALLPIPE ends.

When all the stages in the subroutine pipeline have ended, your stage command is eligible to run again. The original connections are restored after CALLPIPE ends, so your stage command can use both its input and output streams as though CALLPIPE's connections never existed.

Storing Sequences of Stage Commands

CALLPIPE makes it easy to store stages or sequences of stages that you often use in PIPE commands. Suppose that you often use this SPECS stage command, which adds sequence numbers to the beginning of a data set:

```
specs recno 1 1-* next
```

You could save it in as a subroutine pipeline within a user-written stage command:

```

/***** REXX *****/
/* PREFIX.PIPE.REXX(ADDSEQ) */
/* -- Adds sequence numbers to the beginning of records */
/*****
'callpipe *: | specs recno 1 1-* next | *:'
exit 0

```

Then use ADDSEQ whenever you want to add sequence numbers:

```
pipe < mybook.script | addseq | > newbook.script
READY
```

More often, you'll be saving sequences of stage commands. Figure 249 shows a user-written stage command named COUNTWDS. It contains a subroutine pipeline that counts the number of words in its primary input stream, adds some text, and writes the result to its primary output stream. It packages two stage commands together for frequent use.

```
/* ***** REXX ***** */
/* PREFIX.PIPE.REXX(COUNTWDS) */
/* -- Count words and add some text */
/* ***** */
'callpipe',
  '*: ', /* connect to output of stage preceding caller */
  '| count words', /* count number of words in input stream */
  '| specs /Number of words is/ 1 1-* nextword', /* prefix record */
  /* with "Number of words is" */
  '| *: ' /* connect to input of stage following caller */
exit
```

Figure 249. COUNTWDS: Subroutine Pipeline Example

The connectors used at the beginning and end of the subcommand allow the subroutine pipeline to take over the input and output streams for the COUNTWDS stage command. Records flow into the subroutine pipeline through its input connector and later flow out through its output connector.

count words writes to the output stream the number of words delimited by a blank in the input stream. specs /Number of words is/ 1 1-* nextword inserts a prefix of Number of words is at the beginning of the record containing the count information. It then writes the resulting record to the output stream.

Here's an example of how to use the user-written stage command to display the number of words in the data set BLACK.BOOK:

```
pipe < black.book | terminal
Anna Karinina
Scarlet OHara
Miss Piggy
pipe < black.book | countwds | terminal
Number of words is 6.
READY
```

Other Formats of Connectors

The connectors we have been using (*:) are actually abbreviations. You'll also see the input and output connectors written like this:

```
*.input:
*.in:
*.output:
*.out:
```

So, we can rewrite FIXED as follows:

```
/***** REXX *****/
/* PREFIX.PIPE.REXX(FIXED) */
/*      -- Make all records 80 bytes in length */
/*****/
'callpipe *.input: | chop 80 | pad 80 | *.output:'
exit rc
```

You may see other connector formats in CALLPIPE subcommands and ADDPIPE subcommands that others have written.

Using Connectors with CALLPIPE

The CALLPIPE subcommands shown in the previous section used connectors to connect the specified pipeline to the input and output streams. In most cases, you'll want to use both connectors. However, you can omit one connector from either end of the pipeline, or even from both ends. Consider the case in which you omit the connector from the end, as Figure 250 shows:

```
/***** REXX *****/
/* PREFIX.PIPE.REXX(LOGIT) */
/*      -- Log records with a time stamp */
/*****/
'callpipe',
  '*:', /* Connect to input stream */
  '| specs /'date() time()'/ 1', /* Tack on date and time */
  '1-* nextword', /* Put input record */
  '| >> logit.file' /* Append records to data set */
exit rc
```

Figure 250. LOGIT: Subroutine without an Output Connector

In LOGIT, a time stamp is added to the records from the input stream. Then the records are appended to the data set LOGIT.FILE. But, because there isn't a connector at the end of CALLPIPE, they do not flow out of the user-written stage command. Here is an example of how you might use LOGIT:

```
pipe literal Returned Bill Smith's call. | logit
READY
pipe literal Worked on Project X today. | logit
READY
```

Figure 251 on page 170 shows a diagram of the previous PIPE commands when CALLPIPE is running.

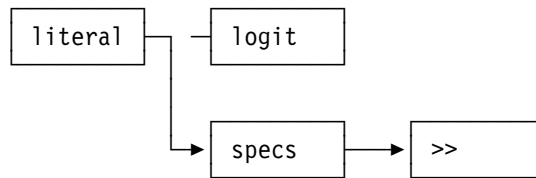


Figure 251. Diagram of Pipeline Using Only an Input Connector

If you put a stage after LOGIT, LOGIT doesn't write to its output stream. Therefore, no records flow to a following stage. Figure 252 shows a CALLPIPE subcommand from which the input connection is omitted. The SEELOG stage command is meant to be used as the first stage of a pipeline.

```

/***** REXX *****/
/* PREFIX.PIPES.REXX(SEELOG) */
/*****/
'callpipe',
  '< logit.file',          /* Read the LOGIT.FILE */
  '| take last 2',        /* Take the last 2 records */
  '| *:'                  /* Write to output stream */
exit rc

```

Figure 252. SEELOG: Subroutine without an Input Connector

Here is an example of the SEELOG subcommand:

```

pipe seelog | terminal
10 Dec 1991 12:38:26 Returned Bill Smith's call.
10 Dec 1991 12:38:30 Worked on Project X today.
READY

```

Figure 253 shows a diagram of the above PIPE command when CALLPIPE is running.

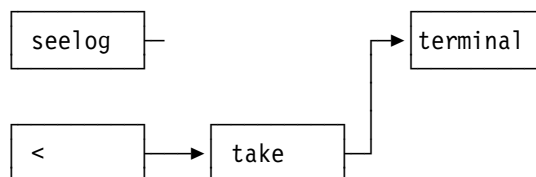


Figure 253. Diagram of Pipeline Using Only an Output Connector

If you don't use SEELOG as the first stage, SEELOG doesn't read its input stream, so the records don't make it past SEELOG. You've seen a similar situation with LOGIT. For example:

```

pipe literal Lost forever | seelog | terminal
10 Dec 1991 12:38:26 Returned Bill Smith's call.
10 Dec 1991 12:38:30 Worked on Project X today.
READY

```

Figure 254 on page 171 shows a diagram of the above PIPE command when CALLPIPE is running. The < stage is not connected to the output stream from LITERAL. Consequently, the record from LITERAL is not processed.

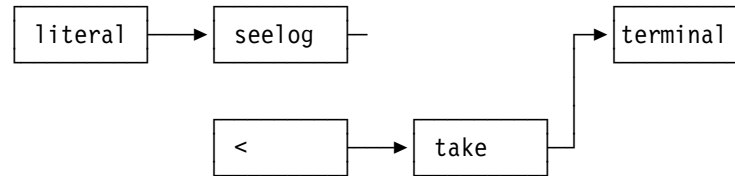


Figure 254. Diagram of Pipeline with Unconnected Streams

Finally, you can omit all connectors from CALLPIPE. In this case, the subroutine pipeline is completely independent from the stage command's input and output streams.

Using CALLPIPE with Other Subcommands

A user-written stage command that contains a CALLPIPE subcommand can also contain other subcommands. There isn't anything special about CALLPIPE that prevents you from also using subcommands like READTO and OUTPUT. You need to be aware of how CALLPIPE is connected to the input or output streams.

Let's look at SEELOG again. Suppose we wanted to process records in the input stream. This is possible because the subroutine pipeline did not read the input stream — the records remain in the input stream. Figure 255 shows a new SEELOG that reads any records in its input stream after the CALLPIPE is processed.

```

/***** REXX *****/
/* PREFIX.PIPE.REXX(SEELOG) */
/*****
signal on error
'callpipe',
  '< logit.file',          /* Read the LOGIT.FILE          */
  '| take last 2',        /* Take the last 2 records      */
  '| *:',                 /* Write to output stream      */
do forever
  'readto in'             /* Read a record from the input */
  'output' date() time() in /* Add time stamp and write it */
end

error:
if rc=12 then rc=0
exit rc

```

Figure 255. SEELOG: Using CALLPIPE, READTO, and OUTPUT Subcommands

We've added the error-handling instructions and a DO loop. The DO loop reads all records in the input stream, prefixes those records with a date and time, and writes them to its output stream. If there aren't any records in the input stream, BatchPipeWorks will give a return code of 12 when READTO is executed.

After CALLPIPE ends, SEELOG reads records from its input stream, processes them, and writes them to its output stream. Thus, you would expect to see these records after those from the data set LOGIT.FILE:

```
pipe literal Lost forever | seelog | terminal
10 Dec 1991 12:38:26 Returned Bill Smith's call.
10 Dec 1991 12:38:30 Worked on Project X today.
10 Dec 1991 13:19:47 Lost forever
READY
```

How would you put the records from the input stream to SEELOG before those in the log data set? You would have to move the DO loop before the CALLPIPE subcommand. You would also have to change the error handling, as shown in Figure 256.

```

/***** REXX *****/
/* PREFIX.PIPES.SEELOG */
/*****
do forever          /* Process the input stream */
  'readto in'        /* Read a record */
  if rc=12 then leave /* Stream not connected? Leave loop */
  'output' date() time() in /* Write output record */
  if rc=12 then exit 0 /* Output not connected? Give up! */
  else if rc<>0 then exit rc /* Pass other nonzero codes to PIPE */
end
'callpipe',
  '< logit.file', /* Read the LOGIT.FILE */
  '| take last 2', /* Take the last 2 records */
  '| *:'          /* Write to output stream */
exit rc

```

Figure 256. SEELOG: Another Variation

As you would expect, the input records are processed first:

```
pipe literal Lost forever | seelog | terminal
10 Dec 1991 13:32:36 Lost forever
10 Dec 1991 12:38:26 Returned Bill Smith's call.
10 Dec 1991 12:38:30 Worked on Project X today.
READY
```

Notice in Figure 256 that we do not end the stage command when a return code of 12 is received on READTO. Instead, we leave the DO loop. In this case, we have some other processing to do. The point is that a return code of 12 does not always mean you must end a stage command. In many situations, a return code of 12 serves as a signal to your stage command to alter its processing flow. The action to be taken is your choice.

In the above SEELOG variants, the CALLPIPE command did not have a connection to the stage's input stream. Suppose CALLPIPE does have a connection to the input stream. In that case, you could still execute READTO subcommands before the CALLPIPE. You might, for example, read and process header records by using READTOs and OUTPUTs, and then process the remaining records with a CALLPIPE.

In the following example, OUTPUT is used to write a header record, then CALLPIPE processes the remaining records:

```
/****** REXX ******/
/* PREFIX.PIFE.REXX(UENG) */
/*      -- Convert data set to uppercase and change header      */
/****** */
signal on error

'output .* This data set contains uppercase records'
'callpipe *: | xlate upper | *:'

error:
if rc=12 then rc=0
exit rc
```

Figure 257. UENG: Using CALLPIPE and OUTPUT

The UENG user-written stage command writes one header record with an OUTPUT subcommand. Then it executes a CALLPIPE subcommand to read the remaining records from the input stream, translate them to uppercase, and write them to the output stream. Here is another example of UENG:

```
pipe < lower.script | terminal
Apples
Bananas
Cherries
Pears
READY
pipe < lower.script | ueng | terminal
.* This data set contains uppercase records
APPLES
BANANAS
CHERRIES
PEARS
READY
```

Another subcommand to use with CALLPIPE is PEEKTO. Suppose you're writing a stage command that processes three different kinds of forms. You need to look at the first record to determine which form it is. But, you don't want to read the record using READTO because that would remove the record from the input stream. Once you have determined which form is being processed, you want to process all pipeline records, including the first, with an appropriate subroutine pipeline.

The PEEKTO subcommand is perfect for this kind of problem, as the following fragment of a stage command shows:

```
/* Stage command fragment to process multiple forms */
signal on error

'peekto in'
select
when pos("Invoice",in) then
    'callpipe *: | ... | *:' /* CALLPIPE that processes invoices */
when pos("Order",in) then
    'callpipe *: | ... | *:' /* CALLPIPE that processes orders */
when pos("Request for Bid",in) then
    'callpipe *: | ... | *:' /* CALLPIPE that processes bids */
otherwise
    exit 99 /* Tell BatchPipeWorks you found an unidentified form */
end /* select */

error:
if rc=12 then rc=0
exit rc
```

To summarize the topic of connections and streams, we provide a REXX program named COMBO, as in Figure 258. COMBO processes three records using READTO/OUTPUT, then three with CALLPIPE, then three more with READTO/OUTPUT and so on until it gets a nonzero return code. The TAKE 3 stage in CALLPIPE forces it to end after processing three records. BatchPipeWorks restores the connections and loops back to READTO/OUTPUT again.

```
/* ***** REXX ***** */
/* PREFIX.PIPE.REXX(COMBO) */
/* -- Take turns handling records */
/* ***** */
signal on error
do forever /* Process a group of six records */
    do i=1 to 3 /* Process only three records */
        'readto record' /* Read a record */
        'output From OUTPUT:' record /* Write it with a tag */
    end
    'callpipe',
    '*: ',
    '| take 3', /* Process only three records */
    '| specs /From CALLPIPE:/ 1', /* Put tag on record */
    '1-* nextword', /* Put input record on it */
    '|*: '
end

error:
if rc=12 then rc=0
exit rc
```

Figure 258. COMBO: Using READTO, OUTPUT, and CALLPIPE Subcommands

Here is an example of using COMBO:

```
pipe < number.list | terminal
one
two
three
four
five
six
seven
eight
nine
ten
READY
pipe < number.list | combo | terminal
From OUTPUT: one
From OUTPUT: two
From OUTPUT: three
From CALLPIPE: four
From CALLPIPE: five
From CALLPIPE: six
From OUTPUT: seven
From OUTPUT: eight
From OUTPUT: nine
From CALLPIPE: ten
READY
```

Additional CALLPIPE Examples

An example you may find useful is a TRACING filter. The argument to TRACING is a string of characters. TRACING prefixes the string to the contents of all records read, displays the records on the terminal, and passes the unmodified record on to the output. The function of the pipeline remains unchanged, but you have added a display of data as it passes through a specific stage. The data is prefixed with a message. Figure 259 on page 176 shows the TRACING subroutine.

```

/***** REXX *****/
/* PREFIX.PIPE.REXX(TRACING) */
/* -- show data in middle of the pipeline, add the tag passed to TRACING, */
/* display each record, and delete the tag before passing the output. */
/***** */
parse arg id /* get argument in mixed case */
if id = '' then do
    'stagenum' /* get the stage number */
    id = 'Stage' rc /* use it as tag */
end
id = id ':'
/* append a colon */

'callpipe',
' *:', /* Read from input */
'| change //'id'/', /* Insert id first */
'| terminal', /* Type */
'| specs' length(id)+1-' * 1', /* Remove id */
'| *:', /* Pass on */
exit rc

```

Figure 259. TRACING: Using CALLPIPE Subcommand

In the CHANGE stage, a null string is specified as the string to be changed. When a null is specified, the string to be substituted (the contents of id, in this case) is inserted at the beginning of each record passing through the stage. The modified records are displayed, and then the SPECS stage removes the identifier.

The following example shows how to run TRACING. (The first PIPE command shows the contents of the TEMP.DATA data set.)

```

pipe < temp.data | terminal
This is the first line.
This is the second line.
READY
pipe < temp.data | tracing first | xlate upper | tracing second | terminal
first :This is the first line.
second :THIS IS THE FIRST LINE.
THIS IS THE FIRST LINE.
first :This is the second line.
second :THIS IS THE SECOND LINE.
THIS IS THE SECOND LINE.
READY

```

Testing Stage Commands

When you develop a new stage command it is easiest to test it by itself. LITERAL and TERMINAL subcommands are especially useful for testing, as are data sets. You can use a device driver for whatever device is convenient.

Two simple ways to test a filter are:

```

pipe literal aa004zz q | myfilter | terminal

pipe terminal | myfilter | terminal

```

Enter the input you want, then look at the output to see if it is correct.

When your input becomes too much to type over again, create a data set with test cases. One way to keep test cases organized is by using stylized data set names. For example, you might use the name of the stage command for the names of data sets containing the test cases (for example, MYFILTER.TEST1, MYFILTER.TEST2).

To test your filter against the data in MYFILTER.TEST1, displaying the result on the terminal:

```
pipe < myfilter.test1 | myfilter | terminal
```

If the output is lengthy, consider using result data sets with stylized data set names:

```
pipe < myfilter.test1 | myfilter | > myfilter.result1
```

These testing techniques are also helpful in the middle of a pipeline to test intermediate results.

Tracing Stage Commands

You can use the REXX TRACE instruction when debugging stage commands. For example, the following statement starts an interactive trace:

```
Trace ?R      /* Starts interactive REXX debug */
```

Each line of your stage command is displayed as it is executed. After each line is displayed, execution pauses until you press the ENTER key or until you end the trace by entering TRACE OFF. Use the SAY instruction to display the contents of variables.

In lengthy stage commands, frame the code that needs to be traced:

```
Trace Results      /* Starts REXX trace                */
    ....           /* code that is traced on the terminal */
Trace Negative     /* Stops REXX trace                */
```

All other trace options can also be used. See the *TSO/E REXX/MVS Reference* for more about TRACE.

If you use several user-written stage commands in a pipeline, you would usually trace only one at a time. When you trace more than one, the displayed traces will be interleaved. Remember that execution of the stages is interleaved and so are the trace displays.

To see how control passes from one stage command to another, trace two of the stages. Note that you do not see the trace record for a subcommand until it ends; another traced stage might start to run before the subcommand ends.

Chapter 10. Blocking and Deblocking

This chapter contains General-Use Programming Interface and Associated Guidance Information.

BatchPipeWorks gives you the ability to convert plain records to *blocked* records and to deblock records. There are several reasons why you might want to block records. In a communications program, for example, you might want to send blocked records instead of individual records for better performance. Or, in a data management program, you might want to block records for efficient storage. Naturally, you would eventually need to deblock the records you have blocked.

You may also have a need to deblock data that another application has blocked. For example, you may need to deblock data sets that originated from another system. If the blocking format matches a format supported by BatchPipeWorks, you can use BatchPipeWorks to deblock the data.

Commonly-used blocking formats range from simple fixed records put one after the other to records wrapped in several layers of protocol. BatchPipeWorks handles several blocking formats. Each of the following formats is discussed in its own section in this chapter:

- Fixed format
- Variable format
- Line-end character format
- Packed format.

The filters that support these blocking formats are BLOCK, DEBLOCK, PACK and UNPACK. At the end of the chapter we also discuss the FBLOCK stage command. FBLOCK creates fixed-length output records from either variable- or fixed-length input records. Unlike the other blocking filters, FBLOCK either blocks or deblocks depending on the relationship of the block size requested to the lengths of the input records.

When blocking records, stage commands put records together in a buffer that usually has room for more than a single record. When the block is created, the filter writes that block to its output stream. Although it is convenient to say that the filter writes a block, the filter actually writes a record (just as other filters do). The record just happens to contain logical records within it. A blocking filter might, for example, read three input records but write only one output record that contains the three records read.

In some formats, a record can *span* blocks. This means that part of the record is in one block and part of it is in the next. In other formats, records cannot span blocks, which means that the entire record must fit in the block.

Another characteristic of blocking formats, in addition to block size and spanning, is the technique used to indicate the end of a record. The end of a record can be defined by a fixed length, a line-end character, or a record descriptor word.

When the end of a record is defined by a fixed length, you know where a record ends because each record is the same known length. When line-end characters are used, a special character (one that is not in the data itself) is used to mark the end of a record. With this scheme, the lengths of the records can vary. When a *record descriptor word* is used, one or more bytes (known as the record descriptor word) are added to the data portion of the record. The record descriptor word describes the record and defines its end.

Blocking operations are *reversible* when it is possible to recover the original format of the data set. This is the case when blocking variable length records, but not when deblocking fixed format records that span blocks.

Fixed Format

In fixed-format blocking, records having fixed lengths are abutted without any control information. The records cannot span blocks, and the block length must be a multiple of the record length. Figure 260 shows a 240-byte block that contains three 80-byte records. Notice that all three records fit neatly into the block.

240-Byte Block

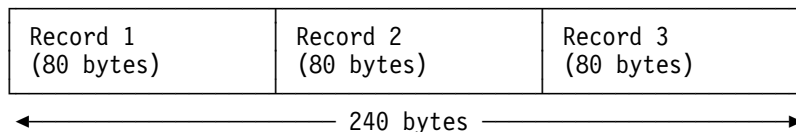


Figure 260. 80-Byte Records in a 240-Byte Block

To block records in fixed format, use the BLOCK stage command. BLOCK requires the block size as an operand. Following the block size, you can optionally specify the FIXED operand. (BLOCK creates fixed-format blocks by default.)

BLOCK FIXED expects fixed-length records in its input stream. The block size you specify must be a multiple of the record length. Conceptually, BLOCK FIXED reads records from its input stream, concatenating them until the block size is reached. Then BLOCK FIXED writes the block (actually a record containing one or more logical records) to its output stream.

In the example in Figure 261 on page 181, an existing data set (NUMBER.LIST) is an F-format data set with a record length of 10. The first PIPE command displays the contents of the data set. Note that each record has some trailing blanks. The last PIPE command shows the use of BLOCK FIXED. It blocks the records from the data set and then displays the blocked records. Notice that the specified block size 30 is a multiple of the record length 10.

```
pipe < number.list | terminal
one
two
three
four
five
READY
pipe < number.list | block 30 fixed | terminal
one      two      three
four     five
READY
```

Figure 261. Blocking Records in Fixed Format

To deblock the records, use the DEBLOCK stage command. Specify the FIXED operand after the DEBLOCK keyword, followed by the length of the original records. FIXED is the default for DEBLOCK, so you can omit the FIXED operand if you wish.

Figure 262 shows two examples of deblocking. In the first example, the records are blocked by BLOCK and are correctly deblocked using a record length of 10. The second example shows what happens if you specify the wrong record size.

```
pipe < number.list | block 30 fixed | deblock fixed 10 | terminal
one
two
three
four
five
READY
pipe < number.list | block 30 fixed | deblock fixed 11 | terminal
one      t
wo       th
ree
four     f
ive
READY
```

Figure 262. Deblocking Fixed-Format Blocks

Variable Format

MVS has four kinds of variable-format records. All four have a 4-byte *block descriptor word* at the beginning of each block that describes the block. The first two bytes contain the length of the complete block including the block descriptor word. The last two bytes are zero. Blocks are limited to 32,760 bytes including the block descriptor word. The four kinds of variable-format records are:

V Deblocked variable records.

In this format, the records are blocked, but with only one record per block. Before each record there is a 4-byte *record descriptor word*. In the first two bytes is the length of the record plus the length of the record descriptor. The second two bytes contain binary zeros. Thus, the longest record possible is 32,756 bytes.

VB Variable blocked records.

In this format the records are blocked with more than one record per block. Before each record there is a 4-byte record descriptor word. Each block contains as many complete records (with their record descriptors) as can fit within the block. Logical records are not spanned across block boundaries.

VBS Variable block spanned records.

In this format, the records are blocked and records can span blocks. Each record or part of a record is a *segment*. Instead of record descriptor words, there are *segment descriptor words*. The 4-byte segment descriptor word contains a two-byte segment length and *segmentation flags*. The segmentation flags define whether the segment is:

- A complete record,
- The first part,
- The last part, or
- An intermediary part of a record that is neither first nor last.

The length of the record is the sum of the lengths of the segments. Because the length is not written explicitly, this format supports logical records longer than 32KB. BatchPipeWorks relieves the programmer from managing the segmentation.

VS Variable spanned records.

In this format, the records are blocked, but with only one record or a part of one record per block. Each new record is placed in a new block, even if there is space available in the previous block. Each record or part of a record is a segment. Instead of record descriptor words, there are segment descriptor words. The 4-byte segment descriptor word contains a two-byte segment length and segmentation flags. The segmentation flags define whether the segment is:

- A complete record,
- The first part,
- The last part, or
- An intermediary part of a record that is neither first nor last.

The length of the record is the sum of the lengths of the segments. Because the length is not written explicitly, this format supports logical records longer than 32KB. BatchPipeWorks relieves the programmer from managing the segmentation.

DEBLOCK V supports all four input formats, but IBM recommends using the V format (specify with the V operand). DEBLOCK V determines the structure of the blocks from record descriptor words and segment descriptor words.

```
pipe < variable.spanned | deblock v | take 20 | terminal
```

When blocking, you must decide the format you want. Code the desired blocking format as a keyword, for instance BLOCK VBS.

Line-End Character Format

In *line-end character* format, the end of each logical record is indicated by a specific character. A line-end character does not follow the last record. The records can span blocks.

Use the BLOCK filter with the LINEND operand to block records in line-end character format. You also need to specify a block size and a line-end character. The character you choose should not occur in the data being blocked (otherwise, it cannot be deblocked properly). Often the best choice is a hexadecimal value that cannot be typed on a terminal.

Figure 263 shows an example in which X'F0' is used as the line-end character. X'F0' happens to be a displayable character: 0. A block size of 80 is used.

```
pipe < legumes.script | terminal
Peas
Bush beans
Pole beans
Lima beans
READY
pipe < legumes.script | block 80 linend f0 | terminal
Peas0Bush beans0Pole beans0Lima beans
READY
```

Figure 263. Blocking Data with Line-End Characters

Notice how the hexadecimal value is specified following `linend`. The hexadecimal value is not enclosed by single quotation marks. Also notice in the output that a line-end character is not placed after the last logical record.

Let's take another example in which the records span blocks. We'll use the same LEGUMES.SCRIPT data set and choose a block size of 5. In the example, the character 0 is specified instead of its hexadecimal value f0; you can use either method to specify the line-end character. Figure 264 shows the result.

```
pipe < legumes.script | block 5 linend 0 | terminal
Peas0
Bush
beans
0Pole
  bean
s0Lim
a bea
ns
READY
```

Figure 264. Spanning Blocks with Line-End Characters

Each displayed line is one block of 5 characters. Since the block size is so small, most of the records span at least one block.

To deblock records in line-end character format, use the DEBLOCK stage command with the LINEND operand. DEBLOCK reads each record from its input stream, and writes a record to its output whenever it finds a line-end character (or when it has read the last record). It does not write the line-end character with the data.

Figure 265 shows PIPE commands that block and deblock data sets. Do not specify the block size as an operand on DEBLOCK. Instead, specify a LINEND operand that identifies the line-end character.

```
pipe < legumes.script | block 80 linend f0 | deblock linend f0 | terminal
Peas
Bush beans
Pole beans
Lima beans
READY
pipe < legumes.script | block 5 linend 0 | deblock linend 0 | terminal
Peas
Bush beans
Pole beans
Lima beans
READY
```

Figure 265. Deblocking Data with Line-End Characters

Packed Format (PACK, UNPACK)

The packed format allows data sets to be saved in less space than is required for unpacked data sets. Packed records have a fixed-length record format and a record length of 1024. They contain logical records in which multiple occurrences of a character are replaced by a count. Logical records can be 64KB or longer.

To pack records, use the PACK stage command. To unpack records, use the UNPACK stage command. First, let's see how to pack records.

When packing records, you must know whether the records have variable lengths or a fixed length. For our first example, let's create packed fixed-length records. Figure 266 shows how. The < stage reads records from an F-format data set named MYCODE.ASSEMBLE. When the input records to PACK have a fixed length, specify the FIXED operand on PACK. PACK reads the fixed-length records from its input stream, compresses the records and writes the compressed records to its output stream. The records are written to an existing data set named MYCODE.PACKED.

```
pipe < mycode.assemble | pack fixed | > mycode.packed
READY
```

Figure 266. Packing Fixed-Length Records

All the records in the input stream to the > stage command have a length of 1024. (PACK fills its last output record with zeros if necessary.)

Now let's pack variable-length records. This time we'll read a data set named MYBOOK.SCRIPT, which is a V-format data set. Assume that the length of the longest record in the data set is 110 bytes. Figure 267 shows how to pack the data set.

```
pipe < mybook.script | pack variable 110 | > packbook.script
READY
```

Figure 267. Packing Variable-Length Records

The < stage reads the data set MYBOOK.SCRIPT and writes those variable-length records to its output stream. PACK reads these records, compresses them, and writes the compressed records to its output stream. The > stage writes the packed records to an existing data set named PACKBOOK.SCRIPT.

Notice that the operands `variable` and `110` are specified on the PACK stage command. What if you don't know the length of the longest record? It is still possible to pack the records, but you must use a multistream pipeline. See the *BatchPipeWorks Reference* for more information about packing variable-length records.

Now that we have packed data sets, how do we unpack them? Use the UNPACK stage command. UNPACK has no operands. It reads records from its input stream and determines whether the records are packed. If they are packed, UNPACK converts them to regular records and writes the converted records to its output stream. If the records are not packed, UNPACK simply copies the records to its output stream.

Figure 268 shows a PIPE command that uses the UNPACK stage command to unpack the data set if it is packed. The command unpacks the data set INPUT.DATA and displays the last 20 lines.

```
pipe < input.data | unpack | take last 20 | terminal
```

Figure 268. Unpacking Records

It's important to keep all packed records intact if you plan to unpack them. Do not use filters that select records, such as TAKE or DROP, before the UNPACK stage. If you do, the remaining records may not look like a packed data set to UNPACK. In that case, UNPACK copies the records to its output stream without converting them. In Figure 268, for example, the TAKE stage is after the UNPACK stage. Reversing the order of the stages would likely cause problems.

Creating Fixed-Format Records with FBLOCK

In “Fixed Format” on page 180, we saw how to block and deblock fixed-length records. One restriction with the BLOCK stage command, however, is that all input records must have the same length. A second restriction is that the block size must be a multiple of the record length; that is, the records cannot span blocks.

The FBLOCK stage command does not have those restrictions. Like the BLOCK stage command, FBLOCK writes output records having fixed lengths, but that is where the similarity ends. FBLOCK accepts both variable- and fixed-length records as input. The block size specified for output records does not have to relate to the size of the input records in any way.

FBLOCK reads records from its input stream and writes output records of the requested length. Whenever FBLOCK's internal buffer is filled, it writes an output record and begins refilling the buffer. Imagine that FBLOCK concatenates all its input records into one long string. Then it chops that long string into pieces of the size you request without regard to the original record boundaries.

Figure 269 on page 187 shows several examples of FBLOCK. The size of the desired output record is specified as an operand.

```

pipe literal bbbb| literal aaa| fblock 3 | terminal
aaa
bbb
b
READY
pipe literal bbbb| literal aaa| fblock 1 | terminal
a
a
a
b
b
b
b
b
READY
pipe literal bbbb| literal aaa| fblock 10 | terminal
aaabbbb
READY

```

Figure 269. Creating Fixed-Length Records with FBLOCK

The preceding examples showed FBLOCK working with input records of variable lengths. FBLOCK also works with input records having a fixed length. In fact, if you specify a block size that is a multiple of the fixed-length input records, FBLOCK behaves the same as BLOCK FIXED. If, however, you specify a block size that is not a multiple of the record length, FBLOCK will span the records as necessary; BLOCK FIXED will display an error message.

It's worth looking at an example of spanned fixed-length records. You might want to deblock data sets in this format. Figure 270 shows an example in which the block size is 200, but the record length is 80. In this case some records (record 3 in the example) begin on one block and end on the next.

200-Byte Blocks

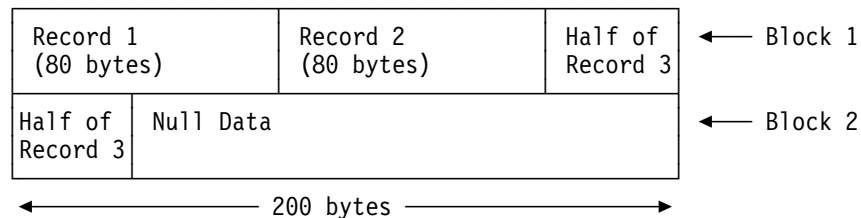


Figure 270. 80-Byte Records in a 200-Byte Block

Figure 271 shows PIPE commands that create fixed blocks having spanned records. The second PIPE command shows how to specify a pad character: type it after the block size. The example uses a hyphen (-), but you can use any character or a hexadecimal value (such as f0).

```
pipe literal 12345| literal 12345| fblock 7 | terminal
1234512
345
READY
pipe literal 12345| literal 12345| fblock 7 - | terminal
1234512
345----
READY
```

Figure 271. Creating Spanned Records with FBLOCK

Let's see what happens when we try to deblock these by adding a second FBLOCK stage. Specify the original record length, which we know to be 5. Figure 272 shows what happens.

```
pipe literal 12345| literal 12345| fblock 7 | fblock 5 | terminal
12345
12345
READY
pipe literal 12345| literal 12345| fblock 7 - | fblock 5 | terminal
12345
12345
----
READY
```

Figure 272. Deblocking Spanned Records with FBLOCK

As you can see, it is not always possible to reverse the blocking operation when fixed, spanned records are involved. Usually you also need to know the number of records that were originally blocked and add a TAKE stage to discard the extras.

Chapter 11. Debugging BatchPipeWorks

This chapter contains Diagnosis, Modification, or Tuning Information.

If you have problems with the BatchPipeWorks programs, fittings, or pipelines, first make sure the syntax of the stage commands is correct. BatchPipeWorks looks for syntax errors before it runs the stage commands. If it finds syntax errors, it displays error messages.

Once the pipeline syntax is correct and the pipeline runs, you may face other problems:

- The pipeline doesn't produce output
- The pipeline produces the wrong output
- You receive an unexpected message on the terminal.

Several tools are available to help determine if the pipeline is coded incorrectly or if there is an error in BatchPipeWorks. The tools are described in this chapter. Most problems can be solved by inspection or by tracing. However, browse through this chapter before debugging to see if one of the other techniques is appropriate.

Tracing BatchPipeWorks

This section uses an example to show how tracing and other tools can be used to solve a problem. The example involves a pipeline that is supposed to read an employee data set and list all employees starting with Jack Brown and ending with Betty Thomas. The employee data set contains:

ALBERT, TOM	40	12904
BROWN, JACK	45	13784
BUTLER, JOE	42	13652
MARKS, SAM	40	17246
THOMAS, BETTY	46	15623
SMITH, SUE	37	16222
WHITE, JOHN	40	14523

We compose a pipeline that uses a BETWEEN stage command to select the records:

```
pipe < employee.data | between Brown Thomas|terminal
BPW000338E Not binary data: Brown
BPW000003I ... Issued from stage 2 of pipeline 1
BPW000001I ... Running "between Brown Thomas"
READY
```

The messages indicate that we have not correctly written stage 2, which contains the BETWEEN stage command. After checking the BETWEEN command syntax, we rewrite the expression with the syntax corrected as follows:

```
pipe < employee.data | between /Brown/ /Thomas/ |terminal
READY
```

This time the pipeline runs, but we do not get any output to the terminal. Because the reason is not obvious, we will trace the pipeline.

To trace a pipeline, specify the TRACE option on the PIPE command or the TRACE operand on the RUNPIPE command. In both cases, the generated trace shows the stage commands as they are executed and the data that is passed from one stage command to the next.

Figure 273 shows an excerpt of a trace specified with the TRACE option on the PIPE command. Notice how the TRACE option is specified.

```
pipe (trace) < employee.data | between /Brown/ /Thomas/ | terminal
BPW000402I Calling Syntax Exit
BPW000003I ... Issued from stage 1 of pipeline 1
BPW000001I ... Running "< employee.data"
BPW000028I Starting stage with save area at X'0AC07AF0 00013B28 00000000'
BPW000003I ... Issued from stage 1 of pipeline 1
BPW000001I ... Running "< employee.data"
BPW000034I "Pipcmd" called
BPW000003I ... Issued from stage 1 of pipeline 1
BPW000001I ... Running "< employee.data"
BPW000039I ... Data: "addpipe (end \) *.output:|deblock 00000000080|i:faninany*.o"
BPW000031I Resuming stage; return code is 0
BPW000003I ... Issued from stage 1 of pipeline 1
BPW000001I ... Running "< employee.data"
BPW000537I Commit level 0
BPW000003I ... Issued from stage 1 of pipeline 1
BPW000001I ... Running "< employee.data"
BPW000028I Starting stage with save area at X'0AC07C30 0AC07230 00000000'
BPW000003I ... Issued from stage 2 of pipeline 1
BPW000001I ... Running "between /Brown/ /Thomas/"
:
```

Figure 273. Example of Trace Output Using Trace Option

You would get the same result with the TRACE operand on RUNPIPE. See Figure 276 on page 192 for an example of using the TRACE operand on RUNPIPE.

Tracing a PIPE command that contains a user-written stage may produce slightly different output. For example, if you use a CALLPIPE or ADDPIPE with the TRACE option in your user-written stage command, the numbering of the stages in the trace output for the CALLPIPE or ADDPIPE begins with one regardless of the stage numbering in the original PIPE command. However, if the CALLPIPE or ADDPIPE begins with a connector, the connector is counted as the first stage. When the CALLPIPE or ADDPIPE has finished processing, the original connection of the stages in the PIPE command are restored and numbering of the stages continues as if the CALLPIPE or ADDPIPE was not issued.

In Figure 274, FIXED is a user-written stage command that contains the following:

```

/***** REXX *****/
/* PREFIX.PIPE.REXX(FIXED) */
/*      -- make all records 80 bytes long. */
/*****
'callpipe *: | CHOP 80 | PAD 80 | *:'

```

The numbers in parentheses indicate the stage numbers that are displayed in the trace output if both the PIPE command and the CALLPIPE subcommand are traced.

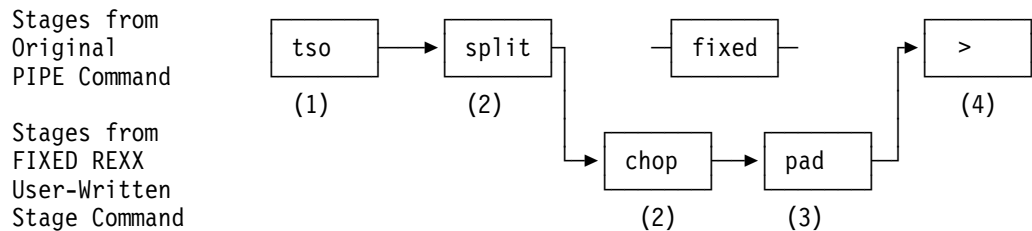


Figure 274. Numbering Stages of a Pipeline When CALLPIPE Is Running

Note that the number of the CHOP stage is 2 because the input connector counts as stage 1.

Usually the trace output is displayed on the terminal. However, because the trace output can be lengthy, it is often better to direct it to a data set.

Tracing to a Data Set

To direct trace output to a data set, use the RUNPIPE stage command. You can specify a trace by using the TRACE option on the PIPE command that is issuing the RUNPIPE stage command, as in Figure 275 or you can use the TRACE operand on RUNPIPE, as in Figure 276 on page 192.

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(TRACER) */
/*      -- Directing Trace Output to a Data Set with RUNPIPE */
/*****
mypipe = '(trace) < employee.data',
          '| between /Brown/ /Thomas/|terminal'
address attach
'PIPE',
  'var mypipe',      /* Write variable MYPIPE to output stream */
  '| runpipe',       /* Run it */
  '| > trace.results' /* Put resultant trace in data set */

```

Figure 275. Directing Trace Output to a Data Set (RUNPIPE with PIPE TRACE Option)

Another way to accomplish the same thing is:

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(TRACE) */
/*      -- Tracing to a data set using TRACE argument of RUNPIPE */
/*****/
mypipe = '< employee.data',
        '| between /Brown/ /Thomas/|terminal'
address attach
'PIPE',
  'var mypipe',      /* Write variable MYPIPE to output stream */
  '| runpipe trace', /* Run it */
  '| > trace.results' /* Put resultant trace in data set */

```

Figure 276. Directing Trace Output to a Data Set (Using RUNPIPE TRACE)

The RUNPIPE stage command reads its input stream and executes the records as pipelines. RUNPIPE writes any terminal output from the execution of the pipelines to its output stream. The stage command following RUNPIPE writes the output to the data set TRACE.RESULTS. Make sure that the data set you are writing to has been previously allocated for variable length records. In our examples, the output from the pipeline consists of trace records.

An excerpt of the trace data set follows. Line numbers are shown for illustration (they do not appear in the actual trace records):

	Lines
:	
BPW000031I Resuming stage; return code is 0	1
BPW000003I ... Issued from stage 1 of pipeline 1	2
BPW000001I ... Running "< employee.data"	3
BPW000035I Output 26 bytes	4
BPW000003I ... Issued from stage 1 of pipeline 1	5
BPW000001I ... Running "< employee.data"	6
BPW000039I ... Data: "BROWN, JACK 45 13784"	7
BPW000031I Resuming stage; return code is 0	8
BPW000003I ... Issued from stage 2 of pipeline 1	9
BPW000001I ... Running "between /Brown/ /Thomas/"	10
BPW000033I Input 0 bytes	11
BPW000003I ... Issued from stage 2 of pipeline 1	12
BPW000001I ... Running "between /Brown/ /Thomas/"	13
BPW000031I Resuming stage; return code is 0	14
BPW000003I ... Issued from stage 2 of pipeline 1	15
BPW000001I ... Running "between /Brown/ /Thomas/"	16
BPW000034I "Locate" called	17
BPW000003I ... Issued from stage 2 of pipeline 1	18
BPW000001I ... Running "between /Brown/ /Thomas/"	19
BPW000031I Resuming stage; return code is 0	20
BPW000003I ... Issued from stage 1 of pipeline 1	21
BPW000001I ... Running "< employee.data"	22
BPW000035I Output 26 bytes	23
BPW000003I ... Issued from stage 1 of pipeline 1	24
BPW000001I ... Running "< employee.data"	25
BPW000039I ... Data: "BUTLER, JOE 42 13652"	26
:	

The excerpt shows the processing for a record we believe should have been selected. To find out why there isn't any output at the terminal, look at the output from stage to stage. The following highlighted excerpt shows what pattern to focus on when scanning this sort of trace:

	Lines
BPW000031I Resuming stage; return code is 0	1
BPW000003I ... Issued from stage 1 of pipeline 1	2
BPW000001I ... Running "< employee.data"	3
BPW000035I Output 26 bytes	4
BPW000003I ... Issued from stage 1 of pipeline 1	5
BPW000001I ... Running "< employee.data"	6
BPW000039I ... Data: "BROWN, JACK 45 13784"	7
BPW000031I Resuming stage; return code is 0	8
BPW000003I ... Issued from stage 2 of pipeline 1	9
BPW000001I ... Running "between /Brown/ /Thomas/"	10
BPW000033I Input 0 bytes	11
BPW000003I ... Issued from stage 2 of pipeline 1	12
BPW000001I ... Running "between /Brown/ /Thomas/"	13
BPW000031I Resuming stage; return code is 0	14
BPW000003I ... Issued from stage 2 of pipeline 1	15
BPW000001I ... Running "between /Brown/ /Thomas/"	16
BPW000034I "Locate" called	17
BPW000003I ... Issued from stage 2 of pipeline 1	18
BPW000001I ... Running "between /Brown/ /Thomas/"	19
BPW000031I Resuming stage; return code is 0	20
BPW000003I ... Issued from stage 1 of pipeline 1	21
BPW000001I ... Running "< employee.data"	22
BPW000035I Output 26 bytes	23
BPW000003I ... Issued from stage 1 of pipeline 1	24
BPW000001I ... Running "< employee.data"	25
BPW000039I ... Data: "BUTLER, JOE 42 13652"	26

Stage 1, which reads EMPLOYEE.DATA, has OUTPUT messages, but stage 2 does not. Stage 2, which contains the BETWEEN stage command, is the culprit.

We see on line 7 that the output from stage 1 is BROWN, JACK 45 13784. The output from stage 1 is the input to stage 2. So, why doesn't BETWEEN select the record?

A hint is on line 10. In the statement between /Brown/ /Thomas/, mixed case is used, but the data is in uppercase. Remembering that BatchPipeWorks is case sensitive, we solve the problem. Rerunning the command with uppercase names gets the output we want.

```
pipe < employee.data | between /BROWN/ /THOMAS/ | terminal
BROWN, JACK      45  13784
BUTLER, JOE      42  13652
MARKS, SAM       40  17246
THOMAS, BETTY   46  15623
READY
```

The following is part of the trace done when the correct pipeline expression was entered:

	Lines
BPW000031I Resuming stage; return code is 0	1
BPW000003I ... Issued from stage 1 of pipeline 1	2
BPW000001I ... Running "< employee.data"	3
BPW000035I Output 26 bytes	4
BPW000003I ... Issued from stage 1 of pipeline 1	5
BPW000001I ... Running "< employee.data"	6
BPW000039I ... Data: "BROWN, JACK 45 13784"	7
BPW000031I Resuming stage; return code is 0	8
BPW000003I ... Issued from stage 2 of pipeline 1	9
BPW000001I ... Running "between /BROWN/ /THOMAS/"	10
BPW000035I Output 26 bytes	11
BPW000003I ... Issued from stage 2 of pipeline 1	12
BPW000001I ... Running "between /BROWN/ /THOMAS/"	13
BPW000039I ... Data: "BROWN, JACK 45 13784"	14

Notice how all the stages now have output for a record that satisfies the BETWEEN criteria.

Tracing Individual Stages

Because tracing generates very large trace data sets, you may want to trace just a few stages. Put the TRACE option at the beginning of the stage you wish to trace, as shown in Figure 277.

```
pipe literal test | (trace) literal trace selectivity | terminal
BPW000028I Starting stage with save area at X'0AC07C38 0AC07310 00000000
BPW000003I ... Issued from stage 2 of pipeline 1
BPW000001I ... Running "literal trace selectivity"
BPW000035I Output 18 bytes
BPW000003I ... Issued from stage 2 of pipeline 1
BPW000001I ... Running "literal trace selectivity"
BPW000039I ... Data: "trace selectivity"
trace selectivity
BPW000031I Resuming stage; return code is 0
BPW000003I ... Issued from stage 2 of pipeline 1
BPW000001I ... Running "literal trace selectivity"
BPW000034I "SHORT" called
BPW000003I ... Issued from stage 2 of pipeline 1
BPW000001I ... Running "literal trace selectivity"
BPW000031I Resuming stage; return code is 0
BPW000003I ... Issued from stage 2 of pipeline 1
BPW000001I ... Running "literal trace selectivity"
BPW000020I Stage returns with code 0
BPW000003I ... Issued from stage 2 of pipeline 1
BPW000001I ... Running "literal trace selectivity"
test
READY
```

Figure 277. Tracing Individual Stages (TRACE Option)

You cannot trace an individual stage using the TRACE operand on the RUNPIPE stage command.

Controlling Trace Messages

You can suppress some trace messages by specifying the NOMSGLEVEL option. NOMSGLEVEL lets you selectively suppress certain levels of messages. Figure 278 shows an example in which some messages are suppressed for the stage command being traced.

```
pipe literal test | (trace nomsglevel 7) literal trace selectivity | terminal
BPW000028I Starting stage with save area at X'0AC07C38 0AC07310 00000000
BPW000001I ... Running "literal trace selectivity"
BPW000035I Output 18 bytes
BPW000039I ... Data: "trace selectivity"
trace selectivity
BPW000031I Resuming stage; return code is 0
BPW000034I "SHORT" called
BPW000031I Resuming stage; return code is 0
BPW000020I Stage returns with code 0
test
READY
```

Figure 278. Using NOMSGLEVEL to Suppress Messages

You can also control trace messages for the entire pipeline by using either the TRACE NOMSGLEVEL option on the PIPE command or by using the MSGLEVEL operand and the TRACE operand on RUNPIPE. See the *BatchPipeWorks Reference* for more about NOMSGLEVEL and MSGLEVEL.

Taking Snapshots of Data

As an alternative to tracing consider taking snapshots of your data as it passes through stages. Tracing lets you follow the data a record at a time, but snapshots let you see all the data at once. For example, suppose you want to see how the data changes after each stage in this PIPE command:

```
/***** REXX *****/
/* PREFIX.PIPE.EXEC(TEMP) */
/*      -- Display list of TEMP data sets sorted in descending order */
/*****/

'pipe',
  'tso listcat',          /* Execute LISTCAT command */
  '| locate /TEMP/',      /* Find data sets with TEMP qualifier */
  '| sort descending',    /* Sort them */
  '| terminal'
```

Stages that write to data sets are added to the pipeline. Figure 279 on page 197 shows how to take snapshots from a REXX exec.

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(TEMP) */
/* -- Display list of TEMP data sets sorted in descending order */
/*****
'pipe',
  'tso listcat',          /* Execute LISTCAT command */
  '  > test1.snapshot',  /* ...take snapshot */
  '  locate /TEMP/',      /* Find those with SCRIPT data set type */
  '  > test2.snapshot',  /* ...take snapshot */
  '  sort descending',    /* Sort them */
  '  terminal'

```

Figure 279. Taking Snapshots

When you use this technique to take snapshots, the data sets that you write to must be previously allocated for variable length records.

Naming BatchPipeWorks (NAME Option)

When writing execs that call other execs that issue PIPE commands, it can be hard to find a failing PIPE command. The error messages reveal the problem, but you can't tell which pipeline issued the messages. In these situations, use the NAME option on your PIPE commands. NAME lets you give a pipeline a name. The name is displayed in any error messages caused by the pipeline.

The following example shows how you might keep track of data sets that have been allocated. The ALC EXEC writes a set of records to a log data set:

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(ALC) */
/* -- to display LISTALC STATUS command in an easy to read format. */
/*****
'pipe (name alc endchar ?)', /* Name the pipeline ALC */
  '  tso listalc status',     /* Get the information */
  '  d:drop 1',              /* Remove the heading for later */
  '  spec 1-* 30 read 1.29 1', /* Put the lines together */
  '  frtarget locate /PREFIX/', /* Display from here */
  '  f:faninany',            /* Put heading back on top */
  '  >> 'prefix.alc.log'',    /* Append the results in ALC.LOG */
  '?',                      /* Start of the second pipeline */
  '  d:',                   /* Get the heading line */
  '  f:'                    /* Put the heading back */

```

ALC EXEC is called by the MYEXEC EXEC:

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(MYEXEC) */
/* -- An example of logging the currently allocated data set */
/*****
  'ALC' /* Log the allocated data sets */
/*****
/* BODY OF THE EXEC */
/* : */
/*****

```

This is a sample of what will be placed in the ALC.LOG data set.

```
--DDNAME---DISP--
  ISPPROF  KEEP          PREFIX.ISPF.PROFILE
           KEEP          SYS1.HELP.TCPHELP
           KEEP          SYS1.REXX.FANHENU
           KEEP          SYS1.REXX.EAGHENU
  SYSEXEC  KEEP          ISP.PHONE.CLISTS
           KEEP          SYS1.ISP.ISPEXEC
           KEEP          PREFIX.PIPE.EXEC
  REXX     KEEP          PREFIX.PIPES.REXX
           KEEP          SYS1.PIPES.REXX
  PIPE     KEEP          SYS1.PIPES.LOAD
READY
```

The only problem is that the PREFIX.ALC.LOG data set does not exist. This causes a PIPE error. Because the NAME option is used, the error messages will include the name of the failing pipeline.

Displaying All Nonzero Return Codes (LISTERR Option)

When you issue a PIPE command, each stage command in the pipeline returns a return code. BatchPipeWorks then returns the most serious return code from all its stages. (Any negative return code is worse than any positive return code.) If you want to see the nonzero return codes returned by the stages in your PIPE command, use the LISTERR option. For example, the following PIPE command produces two errors:

```
pipe (listerr) tso listcat entry() |append tso listct |terminal
BPW000020I Stage returns with code 12
BPW000003I ... Issued from stage 1 of pipeline 1
BPW000001I ... Running "tso listcat entry()"
BPW000020I Stage returns with code -3
BPW000003I ... Issued from stage 2 of pipeline 1
BPW000001I ... Running "append tso listct"
READY
```

Because the entry parameter is incorrectly specified, the TSO LISTCAT stage gives a nonzero return code. The second nonzero return code is caused by a typing error in the APPEND stage. Instead of listcat, the string listct is entered.

With the LISTERR option, messages are displayed for both stages. The messages contain the return codes. Without the LISTERR option, you would not know which stages gave a nonzero return code. But, you know that the PIPE command returns the most serious return code from all its stages. Therefore, the return code for this PIPE command is -3. To see the resulting return code for the PIPE command, run the PIPE command from a REXX exec similar to the one in Figure 280:

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(SHOWERR) */
/*      -- Displaying the resulting Return Code of a PIPE command */
/*****
'PIPE (LISTERR) TSO LISTCAT ENTRY() |APPEND TSO LISTCT |TERMINAL'
SAY 'RETURN CODE IS:' RC

```

Figure 280. Displaying Resulting Return Code of PIPE Command

The resulting return code for the PIPE command is:

```

showerr
BPW000020I Stage returns with code 12
BPW000003I ... Issued from stage 1 of pipeline 1
BPW000001I ... Running "TSO LISTCAT ENTRY()"
BPW000020I Stage returns with code -3
BPW000003I ... Issued from stage 2 of pipeline 1
BPW000001I ... Running "APPEND TSO LISTCT"
      2 *- 'PIPE (LISTERR) TSO LISTCAT ENTRY() |APPEND TSO LISTCT |TERMINAL
      +++ RC(-3) +++
RETURN CODE IS: -3
READY

```

Appendix A. Stage Commands and Pipeline Subcommands

This Appendix lists all the BatchPipeWorks stage commands, by category. When a stage command has the characteristics of two categories, it is listed in both tables. Categories are as follows:

- Stage commands that device drivers; see page 201.
- Stage commands that change the contents of records; see page 202.
- Stage commands that select records; see page 203.
- Stage commands that support multiple streams; see page 204.
- Stage commands that block and unblock records; see page 205.
- Stage commands that work with REXX execs; see page 205.
- Stage commands that are event-driven; see page 205.
- Miscellaneous stage commands; see page 206.

Additionally, in “Pipeline Subcommands” on page 207, this appendix lists the subcommands BatchPipeWorks provides for you to write your own stage commands.

Stage Commands that are Device Drivers

<i>Table 2 (Page 1 of 2). Summary of Device Driver Stage Commands</i>	
Stage Command	Task Performed
<	Reads the contents of a physical sequential data set or member of a partitioned data set.
>	Rewrites a physical sequential data set or member of a partitioned data set.
>>	Appends data from the pipeline to a physical sequential data set.
APPEND	Writes primary input stream records to the primary output stream followed by records from a specified stage command or subroutine pipeline.
BPCOPY	Used within BatchPipeWorks fittings, makes copies of data flowing through the BatchPipes pipe.
BPREAD	Used within BatchPipeWorks fittings, reads records from a BatchPipes writer or a BatchPipes pipe.
BPWRITE	Used within BatchPipeWorks fittings, writes records to a BatchPipes reader or a BatchPipes pipe.
COMMAND	Issues TSO commands. TSO then writes the responses from the commands to the terminal.
GETFILES	Reads a list of data sets.
JOBLOG	Writes to the joblog.
LISTISPF	Reads the directory of a partitioned data set and writes a record for each member of the partitioned data set, including any information that ISPF has added to the directory.
LISTPDS	Reads the directory of a partitioned data set and writes a record for each member.
LITERAL	Writes the specified data to the primary output stream and then copies primary input stream records to the primary output stream.

Table 2 (Page 2 of 2). Summary of Device Driver Stage Commands

Stage Command	Task Performed
PREFACE	Writes records from a specified stage command or subroutine pipeline to the primary output stream followed by primary input stream records.
QSAM	Gets records from or puts records to a physical sequential data set using queued sequential processing.
READPDS	Extracts members from a partitioned data set.
STRLITERAL	Writes the specified data to the primary output stream and then copies primary input stream records to the primary output stream.
STATE	Determines whether the specified data set or data sets exist.
SUBCOM	Passes specified commands to a specified subcommand environment.
TERMINAL	Reads from or writes to the terminal in line mode.
TSO	Issues TSO commands and writes the responses from the commands to the pipeline rather than the terminal.
WRITEPDS	Stores members into a partitioned data set.
XRANGE	
Creates one record containing a specified range of characters.	

Stage Commands that Change the Contents of Records

This table includes the subset of filter stage commands that change the contents of records. The filter stage commands that select records are listed in “Stage Commands that Select Records” on page 203.

Table 3 (Page 1 of 2). Summary Stage Commands that Change the Contents of Records

Stage Command	Task Performed
ASATOMC	Converts ASA carriage control to machine carriage control
CHANGE	Replaces a string of characters with another string of characters.
CHOP	Selectively truncates records.
COMBINE	Combines several records into one record according to a specified logical operator.
JOIN	Concatenates groups of records.
JOINCONT	Joins records marked with a continuation string.
MCTOASA	Converts machine carriage control to ASA carriage control.
PAD	Extends records with one or more occurrences of a specified character.
REVERSE	Reverses the contents of records on a character-by-character basis.
SNAKE	Builds a multicolumn page layout.
SPECS	Rearranges the contents of records.
SPLIT	Splits records into multiple records.
STRIP	Removes leading or trailing characters from records.
TOKENIZE	Parses records according to a specified token.
VCHAR	Recodes characters to a different length.

<i>Table 3 (Page 2 of 2). Summary Stage Commands that Change the Contents of Records</i>	
Stage Command	Task Performed
UNTAB	Expands tab characters (X'05') to blanks for lining up data into columns.
XLATE	Translates characters based on a specified translation table.

Stage Commands that Select Records

This table includes the subset of filter stage commands that change records. The filter stage commands that change the contents of records are listed in “Stage Commands that Change the Contents of Records” on page 202.

<i>Table 4 (Page 1 of 2). Summary of Stage Commands that Select Records</i>	
Stage Command	Task Performed
ALL	Selects records containing a specified string or specified strings defined by an expression comprising of character strings and logical operators.
BETWEEN	Selects records between two specified targets including the records containing the target. The specified targets must begin in the first column of a record.
DROP	Discards one or more records.
FIND	Selects records that begin with a specified text.
FRLABEL	Selects records that follow a specified target including the target record. The specified target must begin in the first column of a record.
FRTARGET	Selects all records starting with the first record selected by a specified stage command.
HOLE	Discards records.
INSIDE	Selects records between two specified targets not including the records containing the target. The specified targets must begin in the first column of a record.
LOCATE	Selects records that contain a specified string of characters. The characters can appear at any position within the record.
NFIND	Selects records that do not begin with a specified text.
NINSIDE	Selects records not located between two specified targets. The records containing the targets are also selected. The specified targets must begin in the first column of a record.
NLOCATE	Selects records that do not contain a specified string of characters.
PICK	Compares a field in the primary input stream record to a specified string or a second field in the record, and selects the record if the comparison satisfies the specified relation.
OUTSIDE	Selects records not located between two specified targets. The records containing the targets are not selected. The specified targets must begin in the first column of a record.
STRFIND	Selects records that begin with a specified string of characters.
STRFRLABEL	Selects records that follow a specified target including the target record. The specified string must begin in the first column of a record.
STRNFIND	Selects records that do not begin with a specified string of characters.
STRTOLABEL	Selects records that precede a specified target, not including the target record. The specified target must begin in the first column of a record.

Table 4 (Page 2 of 2). Summary of Stage Commands that Select Records

Stage Command	Task Performed
STRWHILELABEL	Selects consecutive records that begin with a specified string. The records must be at the beginning of the input stream. The specified string must begin in the first column of a record.
TAKE	Selects one or more records from the beginning or end of the primary input stream.
TOLABEL	Selects records that precede a specified target not including the target record. The specified target must begin in the first column of a record.
TOTARGET	Selects all records up to but not including the first record selected by a specified stage command.
UNIQUE	Compares the contents of adjacent records and discards or retains the duplicate records.
WHILELABEL	Selects consecutive records that begin with a specified string. The records must be at the beginning of the input stream. The specified target must begin in the first column of a record.

Stage Commands that Support Multiple Streams

Table 5. Summary of Stage Commands that Support Multiple Streams

Stage Command	Task Performed
COLLATE	Matches records from two input streams and writes matched and unmatched records to different output streams.
DEAL	Writes a primary input stream record to one of its connected output streams in either sequential order starting with the primary output stream, or some other order specified on the secondary input stream.
FANIN	Combines multiple input streams into a single stream in a specified order.
FANINANY	Combines multiple input streams into a single stream. FANINANY reads an input record from any input stream that has a record available.
FANOUT	Copies primary input stream records to multiple output streams.
GATE	Causes portions of a pipeline to end.
GATHER	Reads records from its connected input streams in either sequential or some other specified order, and writes them to its primary output stream.
JUXTAPOSE	Prefaces records in the secondary input stream with records from the primary input stream.
LOOKUP	Finds records in a reference.
MERGE	Combines records from all input streams in ascending or descending order.
OVERLAY	Reads a record from each input stream and merges the records read into a single record.
PREDSELECT	Copies a record from its primary input stream to either its primary or secondary output stream depending on the order of arrival of input records on its other input streams.
SYNCHRONIZE	Reads records from each of its input streams while each stream has a record available.
UPDATE	Modifies the primary input stream based on the contents of the secondary input stream.

Stage Commands that Block and Deblock Records

<i>Table 6. Summary of Stage Commands that Block and Deblock</i>	
Stage Command	Task Performed
BLOCK	Blocks records.
DEBLOCK	Converts blocked records back into their original format or creates logical records from an external data format.
FBLOCK	Reformats the primary input stream records to blocks of a specified size.
IEBCOPY	Processes an MVS unloaded data set.
JOIN	Concatenates groups of records.
JOINCONT	Joins records marked with a continuation string.
PACK	Compacts data.
SNAKE	Builds a multicolumn page layout.
SPLIT	Splits records into multiple records.
UNPACK	Converts primary input stream records compressed by PACK back to their original format.

Stage Commands that Work With REXX Execs

<i>Table 7. Summary of Stage Commands that Work With REXX Execs</i>	
Stage Command	Task Performed
REXX	Invokes a REXX program as a stage command.
REXXVARS	Gives information about REXX variables.
SCM	Lines up comments and completes unclosed comments in REXX and C programs.
STACK	Reads from or writes to the data stack.
STEM	Gets or sets REXX variables with the specified stem.
VAR	Gets or sets a REXX variable.
VARLOAD	Sets a REXX variable.

Stage Commands that are Event-Driven

<i>Table 8. Summary of Event-Driven Stage Commands</i>	
Stage Command	Task Performed
DELAY	Waits until a particular time of day or until a specified interval of time has passed to copy the record.
PIPESTOP	Terminates stages waiting for an external event.

Miscellaneous Stage Commands

Table 9. Summary of Miscellaneous Stage Commands

Stage Command	Task Performed
BUFFER	Accumulates all records in a single stage not passing any on until all have been received.
CASEI	Selects records relative to a target character string regardless of the case representation of the character string.
COPY	Delays by one record the passing of records from the input stream to the output stream to prevent a pipeline stall.
COUNT	Counts bytes, blank-delimited character strings, or records.
DUPLICATE	Writes each input record in addition to the specified number of copies of each input record.
ELASTIC	Puts a sufficient number of input records into a buffer to prevent a pipeline stall.
ESCAPE	Inserts escape characters in records.
INSTORE	Reads records from its input stream into storage and writes a single record containing only the pointers to the records in storage.
MACLIB	Generates a macro library from COPY data set members.
NOT	Reverses the primary and secondary output streams of a specified stage command.
OUTSTORE	Unloads a data set loaded into storage by INSTORE.
PAUSE	Sends a signal from the pipeline containing the PAUSE stage to the pipeline containing the RUNPIPE stage to receive a type X'11' PAUSE event record.
PIPCMD	Issues primary input stream records as subcommands.
QUERY	Displays one of the following: the version of BatchPipeWorks, the message level, the list of messages that have been issued, or the level of BatchPipeWorks.
RUNPIPE	Issues input stream records as pipelines.
SORT	Arranges records in ascending or descending order.
TIMESTAMP	Determines when a record was read.
ZONE	Defines locations of the input data in records from which records are selected when using a specified stage command.

Pipeline Subcommands

Table 10. Summary of Subcommands

Subcommand	Task Performed
ADDPIPE	Adds one or more pipelines to the set of running pipelines.
ADDSTREAM	Defines an unconnected input or output stream.
BEGOUTPUT	Enters an implied output mode where anything directed to the subcommand environment is written to the currently selected output stream.
CALLPIPE	Invokes a subroutine pipeline.
COMMIT	Commits a stage to a different commit level.
MAXSTREAM	Gets the number of the highest numbered input or output stream.
MESSAGE	Displays a message at the terminal.
NOCOMMIT	Disables automatic commits performed by subcommands.
OUTPUT	Writes a record to the currently selected output stream.
PEEKTO	Reads a record from the currently selected input stream without removing the record from the stream.
READTO	Reads a record from the currently selected input stream.
RESOLVE	Determines if a stage command is a built-in stage command.
REXX	Invokes a REXX program as a stage command.
SELECT	Selects a stream.
SETRC	Sets a return code.
SEVER	Disconnects from the currently selected stream and restores the previous connection, if any.
SHORT	Connects the currently selected input stream to the currently selected output stream.
STAGENUM	Gets the relative position of a specified stage in a pipeline of the primary stream.
STREAMNUM	Gets the stream number of a specified stream.
STREAMSTATE	Gets the state of a specified stream.

Appendix B. Additional Examples

This appendix contains additional examples that show how stage commands can be combined effectively.

Displaying Block Comments

The REXX exec DISPBLKC (Figure 281) displays the first block comment in a REXX exec. The example specifies the REXX exec to be processed as an argument to DISPBLKC. DISPBLKC uses the FRLABEL and TOLABEL stage commands to select the lines required. It drops the first line matched to avoid selecting a null range.

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(DISPBLKC) */
/*      -- Get block comment from an exec */
/*****/
address attach
arg dsn .
'PIPE',
  '<' dsn,          /* Read the requested data set */
  '| frlabel /*****' ||, /* Copy from start of block */
  '| drop 1',        /* But drop the first comment */
  '| tolabel /*****' ||, /* Stop at end of block */
  '| change -/*--',   /* Discard comment delimiters */
  '| change -/*--',
  '| terminal'        /* Display the comment text */
exit rc
```

Figure 281. Displaying Block Comments: DISPBLKC EXEC

Here is an example:

```

dispblkc pipe.exec(dispblkc)
PREFIX.PIPE.EXEC(DISPBLKC)
      -- Get block comment from an exec
READY
```

Adding Sequence Numbers to a Data Set

Figure 282 on page 210 shows a sample REXX exec that creates a data set having sequence numbers with leading zeros in columns 1 through 8. The first record number is 10. The sequence numbers are incremented by 10.

Note that the SPECS RECNO option generates a 10-character field, but you want only 8 of them. An explicit field length takes care of this. You could also generate a 10-byte sequence field and then discard the first two bytes in a subsequent stage.

```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(CRTSCR) */
/* -- Create data set with sequence numbers */
/*****/

address attach
'PIPE',
  'literal :egdoc.'||,          /* Some data lines */
  '| literal here is a line in the document'||,
  '| literal :body.'||,
  '| literal :gdoc.'||,
  'specs',
  'recno 1.7 right',           /* Add sequence numbers */
  '1-* 9',                    /* Load record into column 9 onward */
  '| xlate 1.8 40 0',          /* Make leading zeros */
  '| > sample.script'
exit rc

```

Figure 282. Adding Sequence Numbers: CRTSCR

Because the RIGHT operand is used to position the record number (RECNO), the number is padded on the left with blanks. Notice that column 8 is also a blank. The XLATE stage changes the leading blanks and the blank in column 8 to zeros. This yields leading zeros and an increment by 10, as the following example shows:

```

crtscr
READY
pipe < sample.script|terminal
00000010:gdoc
00000020:body.
00000030 here is a line in the document
00000040:egdoc.
READY

```

Reversing the Order of Records

The REXX exec named REVLINES in Figure 283 on page 211 reverses the order of records flowing through it. It accepts a number as an operand. The operand indicates the number of records that are to be reversed. After reversing the specified number of records, REVLINES copies any remaining records in its input stream to its output stream.

To reverse the records, REVLINES stores them in REXX variables and writes them in reverse order (notice the DO instruction). If the requested number is greater than the number of records in the input stream, REVLINES reverses all the available records.

```

/***** REXX *****/
/* PREFIX.PIFE.REXX(REVLINES) */
/* -- Reverse the order of the first N lines */
/***** */
parse arg number . /* Read number of lines to reverse */
if ~datatype(number, 'Whole') then exit 999 /* Valid number? */

'callpipe',
  '*:', /* Connect input stream */
  '| take' number, /* Take number of records requested */
  '| stem IN.' /* Put them in a stemmed array */

do i=in.0 by -1 to 1 /* Write them in reverse order */
  'output' in.i
end

'short' /* Copy the rest */
exit rc

```

Figure 283. Reversing the Order of Records: REVLINES exec

Examples are as follows:

```

pipe literal def|literal abc|terminal
abc
def
READY
pipe literal def|literal abc|revlines 3|terminal
def
abc
READY
pipe literal ghi|literal def|literal abc|revlines 2|terminal
def
abc
ghi
READY

```

Isolating Words

The following REXX exec, ISWORDS, (Figure 284) isolates words of text by writing each word in its input records to a separate output record. It retains quotation marks within words (for example, Fred's), but not quotation marks at the beginning or end (for example, users').

```

/***** REXX *****/
/* PREFIX.PIPES.REXX(ISOWORD) */
/*      -- Isolate words */
/*****/
'callpipe',
  '*:',
  "xlata 40-7f 40 ' ' 0-9 40", /* Connect input stream */
  'split', /* No punctuation and numbers */
  'specs 1-* next / / next', /* Isolate words */
  "change 1 '/'", /* Pad a blank on end */
  "change '/' //", /* Remove leading quote */
  'strip', /* ...and trailing one */
  '*:' /* Just words, no blanks */
/* Write to output stream */
exit rc
```

Figure 284. Isolating Words: ISOWORD exec

The XLATE stage removes all other punctuation and numbers, but leaves single quotation marks. The record is then split into words. The SPECS stage command puts a blank at the end of each word so that a following CHANGE stage command can remove trailing quotation marks. After leading and trailing quotation marks are removed, STRIP removes any leading or trailing blanks. Note that case is respected in the output records. Use XLATE LOWER if you want all output records in lowercase.

Writing the First Comment Lines of Data Sets

If your PDS members have meaningful comments on their first lines, you may wish to have an index of first lines. The user-written stage command CMLINES (Figure 285 on page 213) is an example that produces such an index.

CMLINES does not have an operand. Instead, it expects a list of PDS members in its input stream.

For each data set listed in the input stream, CMLINES writes output records containing the member name of the data set and the first line from it. CMLINES eliminates comment delimiters from the lines.


```

/***** REXX *****/
/* PREFIX.PIPE.EXEC(CMLINES) */
/* - Display the first comment lines from each member */
/***** REXX *****/
arg pds .
'pipe (endchar ? name getfiles)',
  '| listispf' pds, /* get member names */
  '| spec w1 1', /* get only the member name */
  '| f:fanout', /* send member name to getfiles */
  '| pad 10', /* add blanks to the member name */
  '| j:juxtapose', /* put the member name in front */
  '| terminal', /* the content before displaying */
  '| ?f:', /* second pipeline */
  '| specs /"pds"(/ 1 w1 n //) n', /* ad the member name to the PDS */
  '| l:locate /'/', /* is the name fully qualified? */
  '| change //'', /* if so remove the quotes */
  '| specs /' 1 1-* n /' n', /* and put them on the ends */
  '| i:faninany', /* join names into one stream */
  '| specs %callpipe% 1',
  '| % <% n 1-* nw', /* get the contents of each member*/
  '| %||strwhilelabel &/%&% n', /* keep starting comment lines */
  '| %||*:% n', /* return to the original pipeline*/
  '| pipcmd', /* run the callpipe */
  '| j:', /* put the contents with name */
  '| ?l:', /* bypass for DSN's with out */
  '| i:' /* quotes */
Exit

/* Output.....
ADDIN /* ADDIN EXEC */
ALC /* a LISTALC STATUS command that you can read */
APLDEC /* APLDEC EXEC */
ASKNAME /* ASKNAME EXEC */
CMLINES /***** REXX *****/
CMLINES /* PREFIX.PIPE.EXEC(CMLINES) */
CMLINES /* - Display the first comment lines from each member*/
CMLINES /***** REXX *****/
DISPBLKC /***** REXX *****/
DISPBLKC /* PREFIX.PIPE.EXEC(DISPBLKC) */
DISPBLKC /* -- Get block comment from an exec */
DISPBLKC /***** REXX *****/
MYALLOC /***** REXX *****/
MYALLOC /* PREFIX.PIPE.ACCESS(MYALLOC) */
MYALLOC /* - Check the return code from the ALLOCATE command. */
MYALLOC /***** REXX *****/
MYDSN /* TO DO SOMETHING WITH A DSN */
MYEXEC /* MYEXEC WILL COUNT THE NUMBER OF WORDS IN THE ARGUMENT */
RUNMIN /***** REXX *****/
RUNMIN /* PIPE.EXEC(runmin) */
RUNMIN /* To run a specified command every minute until 5:00pm*/
RUNMIN /***** REXX *****/
SETUP /* TO SETUP MY EXEC LIBS */

```

Figure 285. Writing the First Comment Line of Data Sets: CMLINES

Marking Selected Lines

Figure 286 shows a REXX exec named MARKLINE. MARKLINE scans its input records for a specified string. Records containing the string are prefixed with a pointer. Records not containing the target get a blank prefix. MARKLINE writes the prefixed records to its output stream.

```

/***** REXX *****/
/* PREFIX.PIFE.REXX(MARKLINE) */
/*      -- Mark lines with a locate target. */
/*****/
parse arg target
if target='' then target='/ /'

'callpipe (endchar ?)',
  '*:', /* Connect to input stream */
  '| l: locate' target, /* Look for target */
  '| specs /---> / 1', /* Put mark at beginning of output record */
  '1-* next', /* Put all of input record after the mark */
  '| f: faninany', /* Join with other lines */
  '| *:', /* Send to output */
  '?',
  '|:', /* Lines without the target flow here */
  '| specs / / 1', /* Put blanks at the beginning of output */
  '1-* next', /* Put all of input record after the blanks */
  '| f:' /* Send to FANINANY to be merged with others */
exit rc
```

Figure 286. Marking Selected Lines: MARKLINE

An example of the output is:

```

pipe < fruits.script | terminal
apple
banana
cranberry
date
elderberry
READY
pipe < fruits.script | markline /berry/ | terminal
  apple
  banana
---> cranberry
  date
---> elderberry
READY
```

Reversing the First and Last Records

The example in Figure 287 writes the first record of the input stream after it has written all the other records to the output stream:

```
/* ***** REXX ***** */
/* PREFIX.PIPEREXX(FIRLAST) */
/*      -- Write the first input record last */
/* ***** */
'callpipe (end ?)',
  '*:',
  '| d: drop 1', /* Write first record to secondary output stream */
  '| f: fanin', /* Read primary input, then secondary */
  '*:',
  '?',
  'd:',
  '| buffer', /* Hold the first record until all are read */
  '| f:'
exit rc
```

Figure 287. Writing the First Record Last: FIRLAST

The example in Figure 288 puts the last record of the input stream first in the output stream data set.

```
/* ***** REXX ***** */
/* PREFIX.PIPEREXX(LASTFIR) */
/*      -- Write the last input record first */
/* ***** */
'callpipe (end ?)',
  '*:',
  '| d: take last', /* Write all but last record to secondary output */
  '| f: fanin', /* Combine streams */
  '*:',
  '?',
  'd:',
  '| buffer', /* Hold records until the last is read */
  '| f:'
exit rc
```

Figure 288. Writing the Last Record First: LASTFIR

In the first example you need to buffer the first record of the data set; in the second example you must buffer all except the last record of the data set. Example runs of FIRLAST and LASTFIR follow:

```
pipe < fruits.script | firlast | terminal
```

```
banana  
cranberry  
date  
elderberry  
apple  
READY
```

```
pipe < fruits.script | lastfir | terminal
```

```
elderberry  
apple  
banana  
cranberry  
date  
READY
```

Appendix C. BatchPipeWorks Messages

BPW00000E No message text for message *number*

Explanation: BatchPipeWorks has discovered an internal error. A *BatchPipeWorks* module requests the message with the number shown, but there is no action defined for the message.

System Action: Depends on where the message is issued.

User Response: Ensure the message level is odd (it is unless you have changed it). Note the string substituted in message BPW00001I, if one follows. Contact your systems support staff.

System Programmer Response: If message BPW00001I is issued and it indicates a REXX program, the program may have issued the ISSUMSG subcommand; ensure that it uses a correct message number. If message BPW00001I is not issued, the unknown message is issued in the pipeline specification parser.

BPW00001I ... Running "*string*"

Explanation: This message is issued after any other message when a stage is currently running and the message level is odd. The first 60 characters of the specification of the stage are substituted in the message.

System Action: None.

User Response: The message level is set by the command PIPCMD MSGlevel *number* and by the MSGLEVEL.

BPW00002I ... Processing "*command*"

Explanation: This message is issued after messages issued by the subcommand processor if the bit for 2 is on in the message level. The first 60 characters of the subcommand issued are substituted in the message.

System Action: Message BPW00001I is issued if the message level is odd. Processing continues.

BPW00003I ... Issued from stage *number* of pipeline *number*

Explanation: This message is issued to identify which stage is the cause of the previously issued message when the NAME is not used in the pipeline specification.

System Action: Message BPW00001I is issued if the message level is odd. Processing continues.

BPW00004I ... Issued from stage : PIPA7MSG

Explanation: This message is issued to identify which stage is the cause of the previously issued message when the NAME is used in the pipeline specification.

System Action: Message BPW00001I is issued if the message level is odd. Processing continues.

BPW00011E Null or blank parameter list found

Explanation: A null parameter list is found by PIPE, the subcommand processor, or a stage needing parameters.

System Action: PIPE, the subcommand processor, or the stage returns with return code -11.

BPW00012E Null pipeline

Explanation: A pipeline specification begins or ends with an end-character; two consecutive end characters are met; or global options are present (in parentheses) with no more data or followed by an end-character.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -12.

BPW00013E No ending *right parenthesis* for global options

Explanation: A leading left parenthesis is found, indicating global options, but there is no closing parenthesis.

System Action: Pipeline scan terminates with return code 13.

User Response: Terminate global options with a right parenthesis.

BPW00014E Option "*word*" not recognized

Explanation: The word substituted is not recognized as one of the global options supported.

System Action: Pipeline scan terminates with return code 14.

User Response: Defined global options are: NAME LISTRC LISTERR LISTCMD STAGESEP ENDCHAR ESCAPE MSGLEVEL.

BPW00015E Value missing for keyword "*keyword*"

Explanation: An operand is specified that requires a value (for instance, NAME), but the following non-blank character is the right parenthesis that ends the global options, or the operand is the last word of the argument string to a stage. This message is issued when an option list ends prematurely, and by stages that use values with operands.

System Action: Pipeline scan terminates with return code 15. When issued by a stage, the stage returns with return code 15.

BPW00016E Last character is escape character

Explanation: The escape character (declared by the ESCAPE) is the last character of a pipeline specification. This is an error because there is nothing to escape.

System Action: Pipeline scan terminates with return code -16.

BPW00017E Null stage found

Explanation: There is a stage separator at the beginning or the end of a pipeline specification; a stage separator is adjacent to an end-character; or there are two stage separators with only blank characters between them.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -17.

User Response: Ensure that the pipeline specification is complete. Check if a comma is missing to indicate REXX continuation to the next stage on the following line.

BPW00018E Pipeline incorrectly generated for "character"

Explanation: BatchPipeWorks has discovered an internal error. *BatchPipeWorks* is generated with unacceptable characters for one of the delimiter characters (stage separator, left parenthesis, right parenthesis, period, or colon).

System Action: Pipeline scan terminates with return code -18.

User Response: Contact your systems support staff.

BPW00019W Label "word" truncated to eight characters

Explanation: The first word of a stage ends in a colon; there are more than eight characters before the colon or the first period in the label.

System Action: The label is truncated on the right. Processing continues.

BPW00020I Stage returns with code *number*

Explanation: Pipeline dispatcher trace is active; LISTRC or LISTERR is in effect. The stage has completed processing.

System Action: The pipeline dispatcher continues with other work. Control returns to the caller when all stages are complete.

BPW00021E Unable to find EXECCOMM for REXX

Explanation: BatchPipeWorks has discovered an internal error. The REXX interpreter did not set up a subcommand environment for EXECCOMM before issuing a command to the default command environment.

System Action: The REXX interface returns with code -21.

User Response: Contact your systems support staff.

System Programmer Response: Ensure that the pipeline module is generated correctly. This message indicates a change in the implementation of REXX. Investigate if corrective service is available.

BPW00023E Impossible record (*number bytes from X'address'*)

Explanation: A stage writes a record (or tries to read into a buffer) that is completely or partially beyond the size of an address space.

The contents of general registers zero and one are substituted.

System Action: Control returns to the stage with return code -23. The call is ignored.

User Response: Check the input file. If the contents of register zero is shown as negative, there may be an error in *BatchPipeWorks*. Contact IBM for service.

BPW00024W Program list in "command" is not doubleword aligned and is ignored

Explanation: A PIPREXX macro is issued with a non-zero value in the three low-order bits of general register 2.

System Action: The in-storage program is ignored.

User Response: Contact your systems support staff.

System Programmer Response: Make sure registers 2 and 3 are zero before issuing the macro PIPREXX to run a REXX program as a pipeline stage.

BPW00027E Entry point *word* not found

Explanation: The named entry point is not a built-in program; it is not found in any declared local directory; and there is no file with file name *word* and REXX.

System Action: Message BPW00001I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -27. The stage terminates with return code -27 when this message is issued by a look-up routine.

User Response: Verify the spelling of the name of the program to run.

BPW00028I Starting stage with save area at X'*address*'

Explanation: Pipeline dispatcher trace is active; the STOP is specified. The address substituted designates the save area that contains the initial register set for the stage.

System Action: None.

BPW00029E Pipeline stalled

Explanation: A set of pipelines is deadlocked.

System Action: The state of each stage is listed in subsequent message BPW00030Is. All stages that have not completed have their input and output connections severed before being dispatched with return code -4095.

User Response: It may help to use FANINANY instead of FANIN. Ensure that there is buffering in all loops when this does not help.

BPW00030I Stage is *state*

Explanation: The pipeline is stalled. The state of each stage is listed. The following states are defined:

ready	The stage is ready to run.
wait loc	Waiting for data in locate-mode.
wait in	Waiting for data in move-mode.
wait out	Waiting for a stage to read its output.
wait ecb	Waiting for an event control block to be posted.
unavail	The stage has been redefined by CALLPIPE; it waits for the subroutine pipeline to complete.
wait any	Waiting for data on any input stream.
returned	The stage has completed execution.
wait com	Waiting for other stages to commit.

System Action: None.

BPW00031I Resuming stage; return code is *number*

Explanation: Pipeline dispatcher trace is active. The stage is being resumed. The return code from the call to the pipeline dispatcher is shown.

System Action: None.

BPW00032I Storage *address length*

Explanation: When a pipeline specification is issued from RUNPIPE trace, this message is issued before message 39 is issued to describe a data record and before message 34 is issued to indicate that a CALLPIPE or ADDPIPE subcommand is being processed. The message text can be used as a pipeline stage to obtain the complete record or command.

BPW00033I Input *number* bytes

Explanation: Pipeline dispatcher trace is active. A PIPINPUT macro or a READTO subcommand is issued. The contents of register 0 are substituted for *number*.

System Action: None.

BPW00034I "entry point" called

Explanation: Pipeline dispatcher trace is active. The entry point shown is called.

System Action: Message BPW00039I may follow with the subcommand being issued.

BPW00035I Output *number* bytes

Explanation: Pipeline dispatcher trace is active. A PIPOUTP macro or a OUTPUT subcommand is issued. The contents of register 0 are substituted for *number*.

System Action: Message BPW00039I follows.

BPW00036I Select *side stream number*

Explanation: Pipeline dispatcher trace is active. A PIPSEL macro or a SELECT subcommand is issued.

System Action: None.

BPW00037I Streamnum *side intersection number*

Explanation: Pipeline dispatcher trace is active. A PIPSTRNO macro or a STREAMNUM subcommand is issued.

System Action: None.

BPW00038I Setting dispatcher exit to X'address'

Explanation: Pipeline dispatcher trace is active. A PIPEXIT macro is issued.

System Action: None.

BPW00039I ... Data: "data"

Explanation: Pipeline dispatcher trace is active. The first 60 bytes of the record are shown.

System Action: None.

User Response: To see all data passing between two stages in the pipeline, insert a stage that copies the data to a file; then look at it later. Or write a REXX program to "say" the data.

BPW00040E REXX program *name* not found

Explanation: The REXX interface cannot find a file for the program you request. (Return code 8 on EXECSTAT.)

System Action: The stage terminates with return code 40.

BPW00042E Entry point missing

Explanation: The RESOLVE subcommand is issued with no operands.

System Action: Return code 42 or -42 is set.

BPW00043E Null label

Explanation: The first non-blank character of a stage definition is a colon.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -43.

User Response: To invoke a REXX program that has a colon in its name:

- Write REXX to specify that the program is in REXX, or
- Define an escape character; put the escape character before the colon, or
- Write a null label (.) as a place-holder.

BPW00044E "*string*" is not an acceptable label

Explanation: *string* does not conform to the syntax for a label. For instance, there may be two or more periods in it.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -44.

BPW00045W Stream identifier "*name*" truncated to four characters

Explanation: There are more than four characters between the period beginning a stream identifier and the colon ending the label.

System Action: The stream identifier is truncated. Processing continues.

BPW00046E Label "*label*" declared without a stage

Explanation: No specification for a stage is found the first time the label is used. The first usage of a label defines the stage to run, and any operands it may have. Subsequent references are to the label by itself.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -46.

User Response: Ensure that the label is spelled correctly. If this is the case, inspect the pipeline specification to see if a stage separator is erroneously put between the label and the verb for the stage.

BPW00047E Label "*label*" is already declared

Explanation: A reference is made to a label that is already defined. The label reference should be followed by a stage separator or an end-character to indicate reference rather than definition.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -47.

User Response: Ensure that the label is spelled correctly. If this is the case, add a stage separator after the label to indicate that this is a reference to a stream other than the primary one. Note that all references to a label refer to the invocation of the stage that is defined with the first usage of the label.

BPW00048E Encoding of "*keyword*" not acceptable

Explanation: The character used for the operand is already a special character in the pipeline specification parser.

The following characters are reserved for other uses in a pipeline specification: left and right parentheses "()," colon ":", period ".", blank, asterisk "*." Other special characters are defined by the global options SEPARATOR (by default "|"), ENDCHAR, and ESCAPE.

System Action: Scan terminates with return code 48.

User Response: Use another character for the function.

BPW00049E Value for keyword "*keyword*" is not acceptable

Explanation: The value must be a single character or two-digit hexadecimal representation of the character to be used for the function indicated by the operand.

System Action: Scan terminates with return code 49.

BPW00050E "*word*" not a character or two hexadecimal digits

Explanation: *word* is not a character or a two-digit hexadecimal representation of a character.

System Action: Return code 50 is set. If issued from the scanner, scan terminates with return code 50. If issued from a stage, the stage terminates with return code 50.

BPW00051E Translation specification missing

Explanation: A column range or a list enclosed in parentheses is specified, but no further operands are present.

System Action: The stage terminates with return code 51.

User Response: Use a place-holder range "*" if the translation specification is correct.

BPW00052E Unknown translate table "*word*"

Explanation: The table is not INPUT, OUTPUT, LOWER, UPPER, A2E, or E2A; nor is it one of the operands TO or FROM (which designate a codepage number).

word is the first word specified after column ranges, if any. It is neither a translation specification nor one of the operands designating a translate table.

System Action: The stage terminates with return code 52.

BPW00053E Odd number of translate pairs

Explanation: The argument string ends prematurely.

System Action: The stage terminates with return code 53.

User Response: The most likely cause of this error is that the first operand is interpreted as a column range instead of a translation specification. For instance, "xlate 40 a" gives this message instead of translating blank characters to lowercase "a." "xlate space a" or "xlate 1-* 40 a" performs the intended function.

BPW00054E Range "*numbers*" not valid

Explanation: A list of column ranges is opened or a keyword is specified that indicates a list of words or fields. A word in it does not conform to a column range syntax. If a valid decimal range is specified, the beginning column is zero or the end of the range is before the beginning.

System Action: The stage terminates with return code 54.

User Response: Use a place-holder "***-**" for the column range if you wish to translate the left parenthesis.

BPW00055E No column range found

Explanation: A left parenthesis is found with a right parenthesis as the next non-blank character.

System Action: The stage terminates with return code 55.

User Response: Use a place-holder "***-**" for the column range if you intend to translate the left parenthesis to the right one, like this: "xlate **-** ()".

BPW00056E More than 10 column specifications

Explanation: There are more than 10 words in the list of column ranges.

System Action: The stage terminates with return code 56.

User Response: Use a cascade of XLATE if you need to translate more than 10 ranges. Alternatively, use a subroutine pipeline with a SPEC to put the fields to be translated adjacent to each other; perform the transliteration desired; then use another SPEC to put them back where they were in the input record.

BPW00057E Column range not ended

Explanation: A left parenthesis is found, meaning a range of columns is specified, but no closing right parenthesis is found.

System Action: The stage terminates with return code 57.

User Response: Use a place-holder range (for instance, *1-**) if you wish to translate the left parenthesis.

BPW00058E Decimal number expected, but "*word*" was found

Explanation: The word has a nonnumeric character.

System Action: The stage terminates with return code 58.

BPW00059E "*number*" is not an acceptable record length

Explanation: The number is zero or negative.

System Action: The stage terminates with return code 59.

BPW00060E Delimiter missing after string "*string*"

Explanation: No closing delimiter is found for a delimited string.

System Action: The stage terminates with return code 60.

User Response: Most likely you never intended to specify a delimited string, but a mistake in a column range caused the specification error.

BPW00061E Output specification missing

Explanation: The output column is not specified for the last item.

System Action: The stage terminates with return code 61.

User Response: A likely cause is that an earlier specification is interpreted as a delimited string instead of what it was intended to be.

BPW00063E Output column "word" not acceptable

Explanation: The word specifies where to put a field in the output record; it is not a positive number or a column range.

System Action: The stage terminates with return code 63.

User Response: A mistake in a conversion or placement option can trigger this message. Another likely cause is that an earlier input specification has been scanned as a delimited string where it should have been a column.

BPW00064E Hexadecimal data missing after *prefix*

Explanation: A prefix is found, indicating that a hexadecimal constant should follow, but the next character is blank or the end of the argument string.

System Action: The stage terminates with return code 64.

User Response: Do not use letters as delimiters for a delimited string.

BPW00065E "string" is not hexadecimal

Explanation: An h, H, x, or X is found in the first character of a specification item to specify a hexadecimal literal, but the remainder of the word is not composed of hexadecimal digits.

System Action: The stage terminates with return code 65.

User Response: Do not use letters as delimiters for a delimited string.

BPW00066E Unacceptable number *number*

Explanation: The number is not appropriate in the context where it is used.

System Action: The stage terminates with return code 66.

BPW00067E The number is incompatible with "option"

Explanation: A number is found first in the argument string, indicating a modifier, but the AT option implies that the target is removed when splitting. SPLIT cannot split before or after this target.

System Action: The stage terminates with return code 67.

BPW00068E Incorrect OS Block Descriptor Word X'hex'

Explanation: The first four bytes of an input record are substituted. The last two bytes of the block descriptor word are not zero.

System Action: The stage terminates with return code 68.

User Response: Check the input file. The most likely cause is that the file you are trying to deblock is not a variable format OS file with block and record descriptor words.

BPW00069E Blocksize mismatch; *number* bytes read, but Block Descriptor Word is *number* bytes

Explanation: The blocksize in the block descriptor word does not agree with the amount of data read.

System Action: The stage terminates with return code 69.

User Response: Check the input file. The most likely cause is that the file you are trying to deblock is not a variable format OS file with block and record descriptor words.

The following REXX program pads or truncates a record to the length indicated in the block descriptor word.

```
/* FIXBDW REXX -- make as long as BDW says */
signal on error
do forever
  'readto in'
  'output' left(in,c2d(left(in,2)))
end
error: exit RC*(RC<>12)
```

BPW00070E Incorrect OS Record Descriptor Word X'*hex*'

Explanation: The last byte of the record descriptor word is not zero. If less than 4 bytes of hexadecimal data are substituted, there may be spurious data at the end of a block.

System Action: The stage terminates with return code 70.

User Response: Check the input file. The most likely cause is that the file you are trying to deblock is not a variable format OS file with block and record descriptor words.

BPW00071E Column number "*number*" must be positive

Explanation: The number is not positive.

System Action: The stage terminates with return code 71.

BPW00072E Last record not complete

Explanation: Premature end-of-file is received by DEBLOCK or UNPACK. That is, end-of-file is received in the middle of a logical record.

System Action: The stage terminates with return code 72.

User Response: Check the input file. The most likely cause is that the input is not blocked in packed format.

BPW00073E Segmentation flags not compatible *previous current*

Explanation: The segmentation flags are incompatible in the sense that the end of a record is not followed by the beginning of another one, or a segment that is not the end of a record is followed by a segment indicating the beginning of a record.

System Action: The stage terminates with return code 73.

User Response: Check the input file. The most probable cause of this error is that the data set is not in the specified format. When using DEBLOCK netdata. on a reader file, be sure to:

- Select only records with X'41' in the first column.
- Delete the first column.
- Pad the record to 80 bytes.

BPW00074E Fixed records not same length; last bytes followed by current bytes

Explanation: Input records to BLOCK fixed. are not all the same length.

System Action: The stage terminates with return code 74.

User Response: Check the input file. Maybe you wanted the function performed by FBLOCK rather than BLOCK; FBLOCK accepts records of any length. Use PAD to increase the length of short records, CHOP to truncate records.

BPW00075E Blocksize not integral multiple of record length; remainder is *number*

Explanation: The blocksize specified is not an integral multiple of the length of the first record read.

System Action: The stage terminates with return code 75.

User Response: Use FBLOCK if you wish to combine records irrespective of their lengths.

BPW00076I Waiting on ECB at X'address': *hex*

Explanation: Pipeline dispatcher trace is active. The stage issues the macro PIPWECB. The address of the ECB and its contents are shown. Bit 1 of the ECB (X'40') indicates that it is posted.

System Action: Processing continues.

BPW00077I Return code *number*

Explanation: The return code from a subcommand is not zero. The LISTERR is active.

System Action: None.

BPW00078E Record length *number* is too much

Explanation: The input record is too long for the device driver or blocking filter in question.

System Action: The stage terminates with return code 78.

User Response: Check the input file. BLOCK, PRINTMC and PUNCH only accept up to 65535 bytes of data.

BLOCK V and BLOCK VB do not support input records longer than 32752 bytes (which is equivalent to the OS restriction of 32756 including the record descriptor word). Use BLOCK VBS to process records of any length.

BPW00081E Incomplete conversion triplet

Explanation: The number of operand words is not divisible by three.

System Action: The stage terminates with return code 81.

**BPW00086I BatchPipes/MVS BatchPipeWorks, 5655-065/SCPSP 1.0109
(Version.Release/Mod) - Generated *date at time***

Explanation: This is the response to the PIPE Query command. Date and time represent the time the module was generated.

System Action: Return code 86 is set.

BPW00087E This stage must be the first stage of a pipeline

Explanation: A program that cannot process input records is not in the first position of the pipeline.

System Action: The stage terminates with return code 87.

BPW00092E More than ten key fields

Explanation: More than the maximum ten key fields are specified for SORT or MERGE.

System Action: The stage terminates with return code 92.

User Response: Use SPEC to rearrange the records to make the fields contiguous so that they can be coalesced.

BPW00097E No pipeline header allocated

Explanation: BatchPipeWorks has discovered an internal error. No pipeline header is allocated.

System Action: Processing terminates with return code 97.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in *BatchPipeWorks*.

BPW00098E Connector not by itself

Explanation: A label is found that has an asterisk as the first component, but a stage definition follows.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -98.

User Response: Connectors must be at the beginning or the end of a pipeline. Most likely an end-character or a stage separator is missing.

BPW00099E Connector not at the beginning or the end of a pipeline

Explanation: A connector is in the middle of a pipeline.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -99.

User Response: Connectors specify how to couple streams between the active pipeline and the one being added to the pipeline set. Connectors must be at the beginning or the end of a pipeline. Most likely an end-character or a stage separator is missing.

BPW00100E Direction "word" not input or output

Explanation: The second component of a connector is not a recognized operand.

System Action: Pipeline scan terminates with return code 100.

BPW00101E "connector" can only be connected with ADDPIPE or CALLPIPE subcommand

Explanation: A connector is found, but the pipeline was not issued with an ADDPIPE or CALLPIPE.

System Action: Pipeline scan terminates with return code 101.

BPW00102E Stream *number* does not exist

Explanation: A connector requests the stream with the number shown, but the calling stage does not have that many streams defined.

System Action: Pipeline scan terminates with return code 102.

User Response: Note that the primary stream has number 0; the secondary stream is number 1.

BPW00103E Stream "*identifier*" not found

Explanation: The calling stage does not have a stream with the identifier specified in the third component of the connector.

System Action: Pipeline scan terminates with return code 103.

BPW00109E "*word*" not a recognized data set organization

Explanation: The operand is not Fixed, Variable, C, and so on.

System Action: The stage terminates with return code 109.

User Response: DEBLOCK V supports all OS variable record formats, blocked or spanned, or both.

BPW00110E Unsupported record in IEBCOPY unloaded data

Explanation: The top three bits of the first record are not all zero.

System Action: The stage terminates with return code 110.

User Response: Check the input file. If the data is indeed an IEBCOPY unloaded PDS, then there seems to be a note list. Remove it in a DROP or NFIND stage.

BPW00111E Operand "*word*" not recognized

Explanation: A keyword operand is expected, but the word does not match any keyword that is valid in the context.

System Action: The stage terminates with return code 111.

BPW00114E Blocksize missing

Explanation: BLOCK is issued without an operand.

System Action: The stage terminates with return code 114.

User Response: Specify the blocksize for a default of Fixed.

BPW00115E Blocksize too small; *number* is minimum for this type

Explanation: The blocksize is too small to hold a one-byte record or segment.

System Action: The stage terminates with return code 115.

BPW00118E Return code *number* renaming the file

Explanation: An erase and write operation is requested for a file. The file exists, so a utility file is written and renamed. The RENAME function fails with the return code shown.

System Action: The stage terminates with return code 118.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in *BatchPipeWorks*. Investigate if corrective service is available.

BPW00122E Insufficient free storage

Explanation: A stage requesting storage has received a non-zero return code.

System Action: The stage terminates with return code -122. There may be too little storage left even to issue this message. In that case, the message is suppressed, but the pipeline return code is likely to be -122.

BPW00124E File system error: Record Descriptor Word length *number*; file LRECL *number*

Explanation: A V-format file is being read through the full block interface. A record is met with a length field indicating a length longer than the logical record length in the file status table entry for the file.

System Action: The stage terminates with return code 124.

User Response: Contact your systems support staff.

System Programmer Response: This may be a real error in the file.

BPW00127E This stage cannot be first in a pipeline

Explanation: A device driver that requires an input stream is first in a pipeline, where there can be no input to read.

System Action: The stage terminates with return code 127.

BPW00128E Record format not existing file format *letter*

Explanation: A file is to be appended to. The explicit record format specified is not the same as the one for the existing file.

System Action: The stage terminates with return code 128.

User Response: Specify the correct record format; use > to replace a file; or erase the existing file before issuing the pipeline.

BPW00129E Null record in file

Explanation: A V-format file is being read through the full block interface. The end-of-file record is not expected.

System Action: The stage terminates with return code 129.

BPW00131E The file exists with record length *number*

Explanation: A fixed-format file is to be appended to. The record length specified is not the same as the one for an existing file.

System Action: The stage terminates with return code 131.

User Response: Specify the correct record length or erase the existing file.

BPW00132E Stream "*word*" already replaced

Explanation: The stream is requested to be replaced in two or more connectors. For instance, two or more connectors refer to *.input: at the beginning of a pipeline.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -132.

BPW00133E Stream "*word*" already prefixed

Explanation: The stream is referenced in two or more connectors that specify a prefix-type connection. For instance, two or more connectors refer to *.input: at the end of a pipeline.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -133.

BPW00134E Record is *number* bytes, but format F file record length is *number*

Explanation: While a fixed-format file is being written, an input record does not have the correct length.

System Action: The stage terminates with return code 134.

User Response: Check the input file. Use PAD to extend records; CHOP to truncate.

BPW00138E Short-circuit not from input to output in *connector*

Explanation: Two connectors are in a pipeline of their own with no stage between them. In this case, the first one must be for input and the second one must be for output.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -138.

BPW00139E No connection available to redefine for *connector*

Explanation: A redefine operation is attempted with a ADDPIPE or CALLPIPE subcommand, but the connection is severed, and thus there is no connection to redefine.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -139.

BPW00140E Record longer than buffer (*bytes bytes input*)

Explanation: An input record does not fit in the buffer when creating format V or VB records or a record is longer than an explicit length on PACK variable.. The length of the record is substituted.

System Action: The stage terminates with return code 140.

User Response: Check the input file. Increase the blocksize to accommodate the required length if you are indeed blocking the data you intend to block.

BPW00151E "*string*" not range of characters or delimitedString

Explanation: The operand is neither a range of characters nor a delimited string of enumerated characters.

System Action: The stage terminates with return code 151.

BPW00152E Blocksize *number* too large; *number* is the maximum

Explanation: The blocksize for BLOCK is larger than the size supported for the blocking format in question. For V and the three other variable formats, the maximum is 32760.

User Response: Choose a smaller blocksize.

BPW00154E Operating environment not supported by stage

Explanation: A stage is requested which does not run on the operating system at hand.

System Action: The stage terminates with return code 154.

BPW00156E String missing

Explanation: A operand (for instance, ANYOF) is found, indicating that a string should follow, but there are no more operands.

System Action: The stage terminates with return code 156.

BPW00157E Null string found

Explanation: There are two consecutive delimiter characters.

System Action: The stage terminates with return code 157.

BPW00161E 64K or more inbound data

Explanation: A 3270 generates 64K bytes or more of input data.

System Action: The stage terminates with return code 161.

User Response: If your terminal is a personal computer, the terminal simulator may have generated an incorrect inbound transmission.

BPW00163E Direction missing

Explanation: SELECT and SEVER must have an operand.

System Action: Processing terminates with return code 163.

BPW00164E Direction "word" not valid or not supported

Explanation: A stage issues a subcommand where the first operand is the word shown. This combination is not supported.

System Action: Processing terminates with return code 164.

BPW00165E Stream identifier "word" not recognized

Explanation: A stage issues a subcommand where *word* is expected to be a stream identifier. The combination shown is not supported.

System Action: Processing terminates with return code 165.

BPW00169E Stream identifier missing

Explanation: SELECT has no operands.

System Action: Processing terminates with return code 169.

BPW00170E Prefix or suffix type connector not allowed

Explanation: A pipeline specification that is issued with CALLPIPE contains an output connector at the beginning of a pipeline or an input connector at the end of a pipeline.

System Action: Processing terminates with return code 170.

User Response: Use the ADDPIPE subcommand to process alternative input or redirect output.

BPW00173E No stage found to run

Explanation: BatchPipeWorks has discovered an internal error. The pipeline is stalled, but error recovery finds no stage that is forced ready to run.

System Action: Processing terminates with return code 173.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in *BatchPipeWorks*.

BPW00174E Stream "*identifier*" already exists

Explanation: The second component of the label refers to a stream that is already defined for the stage.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -174.

User Response: Choose another stream identifier for the label reference.

BPW00175E Language table not generated

Explanation: The language table describing message texts for multiple languages has not been generated.

System Action: Processing terminates with return code 175.

BPW00176E Language "*word*" not found

Explanation: Messages for the requested language were not generated with *BatchPipeWorks*.

System Action: Processing terminates with return code 176.

BPW00177I Spent *number milliseconds* in routine

Explanation: This message is issued when the message level includes the bit for 8K. A message is issued for each stage as it completes. Further messages are issued to list time spent in system services.

System Action: None.

BPW00178E Stream "*identifier*" not found

Explanation: FANIN is used with operands to designate a specific order of streams to be read, but the one shown cannot be selected. SELECT on SPEC requests a stream that is not defined.

System Action: The stage terminates with return code 178.

User Response: This error can be caused by a missing stage separator after FANIN.

BPW00179E Character "*char*" is not an ASA carriage control character

Explanation: The file is not in the correct format.

System Action: The stage terminates with return code 179.

User Response: Check the input file.

BPW00180E Character *X'hex'* is not a machine carriage control character

Explanation: The file is not in the correct format.

System Action: The stage terminates with return code 180.

User Response: Check the input file.

BPW00182W String "*string*" ignored in READTO or PEEKTO command

Explanation: An input operation is performed through the REXX interface. The subcommand has more than two words.

System Action: Remaining words are ignored.

User Response: Ensure the subcommand is issued correctly.

BPW00183E Output buffer overflow; *number* required

Explanation: While unpacking a file, a logical record is met that is longer than the maximum record length declared for the file in the first record.

System Action: The stage terminates with return code 183.

BPW00184E Storage at *address* not released; R12 *hex* R14 *hex*

Explanation: A stage obtained storage. The area of storage was not released through the proper interface. The contents of general registers 12 and 14, at the time storage was allocated, are substituted in the message.

System Action: None.

User Response: Ensure the message level is odd (it is unless you have changed it) and that message BPW00001I is issued to show the stage command in error. Contact your systems support staff.

BPW00185E Entry point *name* is not executable

Explanation: The entry point for the stage contains the operation code zero. Executing it would lead to a program check.

System Action: Message BPW00001I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -185.

User Response: Ensure the message level is odd (it is unless you have changed it) and that message BPW00001I is issued to show the stage command in error.

BPW00186I PIPMOD MSGLEVEL *number*

Explanation: This is the response to "pipe query msglevel".

System Action: Return code 186 is reflected.

BPW00187E "*word*" must be LIFO or FIFO

Explanation: The operand to STACK is not valid.

System Action: The stage terminates with return code 187.

BPW00190E The character cannot begin a stage

Explanation: The first character of the definition of a stage is a special character.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -190.

BPW00191E Second character of connector not a period

Explanation: The first character is an asterisk, indicating a connector, but the second character is not a period.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -191.

BPW00192I ... Scan at position *number*; previous data "*string*"

Explanation: The number substituted is the number of characters from the beginning of the pipeline specification (including global options) to the current scan pointer. The last 20 characters before the scan pointer are substituted for *string*.

System Action: None.

User Response: The error is at or before the character indicated by the scan pointer.

BPW00193E Colon missing in connector

Explanation: The definition of a stage begins with an asterisk, but a blank character or a parenthesis is met before a colon.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -193.

BPW00194E Parenthesis not supported in connector

Explanation: A parenthesis is met in a connector.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -194.

BPW00195E Connector alone in a pipeline

Explanation: A connector at the beginning of a pipeline ends the operand string, or it is followed by an end-character.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -195.

BPW00196E Overlapping or non-sequential ranges

Explanation: CHANGE finds overlapping ranges or ranges that are not left-to-right.

System Action: The stage terminates with return code 196.

BPW00197E Range shorter than first string

Explanation: The “from” string is longer than a range.

System Action: The stage terminates with return code 197.

BPW00198E Count must be one when first string is null

Explanation: A null “from” string is present; the only acceptable count is one.

System Action: The stage terminates with return code 198.

BPW00200E Missing ending parenthesis in expression

Explanation: More left parentheses are met than can be paired with right parentheses in the expression.

System Action: The stage terminates with return code 200.

BPW00204E Too many ending parentheses in expression

Explanation: A right parenthesis is met for which there is no open left parenthesis.

System Action: The stage terminates with return code 204.

BPW00206E Expression missing

Explanation: An opening parenthesis is followed by a closing one or a comma; a comma is followed by a comma, or a comma is followed by a closing parenthesis.

System Action: The stage terminates with return code 206.

BPW00209E Segment length *number* not 2 or more

Explanation: The length byte in front of a segment is zero or one. This is not valid for data blocked in the netdata format.

System Action: The stage terminates with return code 209.

User Response: Check the input file. Ensure that the input stream is indeed in the netdata format and that records are padded to 80 bytes.

BPW00211E Second target missing

Explanation: A delimited string is found for the first target, but the second target is not present.

System Action: The stage terminates with return code 211.

BPW00221E Invalid character "<character>" in expression

Explanation: The character shown is not valid in an expression.

System Action: The stage terminates with return code 221.

BPW00222E Secondary stream not defined

Explanation: Only the primary stream is defined. UPDATE requires two input and output streams. LOOKUP requires at least two streams.

System Action: The stage terminates with return code 222.

User Response: UPDATE reads the master file from the primary input and writes the updated file to the primary output. Transactions (update control cards and replaced/inserted records) are read from the secondary input. The update log is written to the secondary output.

```
/* Sample update */
'pipe (end ?)',
  '< input.data|u:update|> output.data',
  '?< update.data|u:      |> log.data'
```

BPW00223E Sequence error in output file: *previous* to *new*

Explanation: A sequence error is introduced in the output file.

System Action: This message is written to the update log stream. Processing continues. Return code 8 is set unless other errors force a higher return code.

BPW00224E Premature EOF on input; sequence *number* not found

Explanation: An update control record references the number shown, but it is not found before the input stream is exhausted.

System Action: This message is written to the update log stream. Processing continues. Return code 12 is set unless other errors force a higher return code.

BPW00225E Sequence *number* not found

Explanation: An update control record references the number shown, but it is not found. A line with a higher serial number is encountered.

System Action: This message is written to the update log stream. Processing continues. Return code 12 is set unless other errors force a higher return code.

BPW00226E Sequence field length *length* too long; 15 is maximum

Explanation: The length of the sequence field is larger than the maximum supported.

System Action: The stage terminates with return code 226.

BPW00227E Sequence field not present in record; *number* bytes read

Explanation: An input record is too short to contain the sequence field. All master input records are checked for this; detail records being inserted are checked if the control record indicates that the sequence field in the record is to be retained.

System Action: The stage terminates with return code 227.

User Response: Check the input file. Ensure that all records have a sequence field. Move the sequence field to the beginning of variable length records.

BPW00229E Sequence error in input file: *previous* to *new*

Explanation: The input master file has a sequence error.

System Action: This message is written to the update log stream. Processing continues. Return code 8 is set unless other errors force a higher return code.

BPW00230E Unsupported format "*type*"

Explanation: The record format for the packed file is neither fixed nor variable.

System Action: The stage terminates with return code 230.

BPW00231E Null variable name

Explanation: The first two characters of an input record are the same.

System Action: The stage terminates with return code 231.

User Response: Check the input file. Ensure that a single delimiter is used to delimit the variable name from the data to load. The name must begin in the second column of the input record.

A blank or an asterisk (*) in column one indicates a comment line for which no variable is set.

BPW00232E Stem or variable name is *number* characters

Explanation: The variable name is too long. VAR supports at most 250 bytes for the variable name. STEM supports at most 240 bytes in the name of the stem to allow for a 10-character sequence number.

System Action: The stage terminates with return code 232.

User Response: Choose a shorter name for the variable or stem.

BPW00233E No active EXEC environment found

Explanation: A stage refers to the EXEC environment, but no such environment is found.

System Action: The stage terminates with return code 233.

User Response: Ensure that the pipeline is started from an EXEC when using filters referencing EXEC or REXX variables.

BPW00234E Caller not REXX

Explanation: REXXVARS is unable to obtain the interpreter private data.

System Action: The stage terminates with return code 234.

User Response: Ensure that REXXVARS is only called from REXX programs. Such programs begin with a REXX comment (/ * ... */).

BPW00235E Invalid variable name: *word*

Explanation: The variable name is unacceptable to the EXECCOMM interface. The variable may be longer than 250 characters or it may contain a character that is not valid in a variable name.

System Action: The stage terminates with return code 235.

User Response: Ensure that the stem or variable name is spelled correctly. Do not put an ampersand (&) at the beginning of it. VARLOAD requires that the stem part of a variable name must be in uppercase; a simple variable must be completely in uppercase.

BPW00236E Too much data for variable *name*

Explanation: Too much data is to be set. This message is also issued when there is insufficient storage for EXECCOMM processing to complete.

System Action: The stage terminates with return code 236.

User Response: Use CHOP to truncate records.

BPW00237E Unrecognized error status X'*hex*' from EXECCOMM; RC=*number*

Explanation: *BatchPipeWorks* is not prepared for the return code it receives from EXECCOMM.

System Action: Message BPW00552I displays the EXECCOMM parameter list. The stage terminates with return code 237.

BPW00238E Record count "*word*" not zero or positive

Explanation: The contents of the variable that specifies the number of variables in a stemmed array cannot be converted to a non-negative number. The name of this variable is formed by appending a zero (X'F0') to the operand to STEM. The contents of the variable are substituted.

System Action: The stage terminates with return code 238.

User Response: Remember to set the variable to an integer value before calling a subroutine pipeline using STEM to read a stemmed array.

Ensure that the variable set is the one referenced.

BPW00240E Function *name* not supported

Explanation: An expression has an identifier followed by a left parenthesis, indicating a function call, but the function requested does not exist.

System Action: The stage terminates with return code 240.

BPW00241E Unsupported record format

Explanation: MEMBERS is used with a file that is not fixed with 80 byte records; QSAM is used for a file with a record format for which it is not designed.

System Action: The stage terminates with return code 241.

User Response: For MEMBERS, correct the way the library is generated. For QSAM, use a utility to change the record format to fixed or variable.

BPW00250E Syntax error in expression

Explanation: A malformed expression is met. This includes adjacent operators, empty parentheses, and strings that are not separated by an operator.

System Action: The stage terminates with return code 250.

BPW00253E Data not a NETDATA control record

Explanation: A record is met that does not conform to the transmission data format. The beginning of the record is not X'E0' followed by 'INMR0' followed by a number 1 through 4, 6, or 7.

System Action: The stage terminates with return code 253.

User Response: Check the input file. Most likely, the input data is not from a file in the netdata format or data records have not been removed.

BPW00256I Enter a pipeline specification

Explanation: The PIPE command is issued without operands.

System Action: The return code is 256.

User Response: Provide a pipeline specification with the PIPE command.

BPW00257E Subcommand environment *word* not found

Explanation: A device driver interfacing to a subcommand environment is unable to locate the requested environment.

System Action: The stage terminates with return code 257.

BPW00261E Unable to open *ddname*

Explanation: The third bit of DCBOFLGS stays zero.

System Action: The stage terminates with return code 261.

BPW00264E Too many streams

Explanation: Too many streams are defined for MERGE; a selection stage has more than two streams; a secondary stream is defined for a stage that does not use it.

System Action: The stage terminates with return code 264.

User Response: Cascade MERGE stages to merge the required number of streams. For other stages, this message usually indicates trouble with the multistream topology. For instance, this is a subroutine pipeline to select lines with A, B, or C:

```
'callpipe (end ? name ALLMSGs)',
'|*:',
'|a:locate string /A/',
'|f:faninany',
'|*:',
'|?a:',
'|b:locate string /B/',
'|f:',
'|?b:',
'| locate string /C/',
'|f:'
```

BPW00281W Mixed-case command verb "word"

Explanation: COMMAND finds the first word to be different from its translation to uppercase.

System Action: The tokenized parameter list is translated to uppercase.

BPW00282E Stage cannot be used with ADDPIPE

Explanation: One of the device drivers referring to REXX or EXEC variables is requested in a pipeline specification issued with ADDPIPE. Since the two programs would run in parallel, it is not possible to ensure that the EXECCOMM environment will remain for the duration of the new pipeline.

System Action: The stage terminates with return code 282.

User Response: Use CALLPIPE to load or store variables in a REXX filter.

BPW00287E Number *number* cannot be negative

Explanation: A negative number is specified for an operand to a stage that only supports zero or positive numbers.

System Action: The stage terminates with return code 287.

BPW00303E Error *number* on *function*

Explanation: The return code shown is received when performing the ISPF or CMS function shown.

System Action: The stage terminates with return code 303.

BPW00335E Odd number of characters in hex data: *string*

Explanation: A prefix indicating a hexadecimal constant is found, but the remainder of the word does not contain an even number of characters.

System Action: The stage terminates with return code 335.

BPW00336E String length not divisible by 8: *string*

Explanation: A prefix indicating a binary constant is found, but the number of characters in the remainder of the word is not divisible by eight.

System Action: The stage terminates with return code 336.

BPW00337E Binary data missing after *prefix*

Explanation: A prefix indicating a binary constant is found, but there are no more characters in the argument string or the next character is blank.

System Action: The stage terminates with return code 337.

BPW00338E Not binary data: *string*

Explanation: A prefix indicating a binary constant is found, but the remainder of the word contains a character that is neither 0 nor 1.

System Action: The stage terminates with return code 338.

BPW00339E PIPSDDEL return code *number*

Explanation: An unexpected return code is received on a conversion operation.

User Response: Contact your systems support staff.

System Programmer Response: This is an error in *BatchPipeWorks*. Contact IBM for service.

BPW00350E Primary key longer than secondary

Explanation: The primary key is longer than the secondary key.

System Action: The stage terminates with return code 350.

BPW00380E Left parenthesis missing

Explanation: A left parenthesis is expected for a list of items, but one is not found.

System Action: The stage terminates with return code 380.

BPW00381E Right parenthesis missing

Explanation: A left parenthesis for a list of items has been met, but no right parenthesis is found.

System Action: The PIPE command or stage terminates with return code 00381.

BPW00382E Nothing specified within parentheses

Explanation: An opening parenthesis is found with only blank characters before the closing parenthesis.

System Action: The stage terminates with return code 382.

BPW00391E Unsupported conversion *type*

Explanation: The type shown is syntactically correct to request a conversion of a field, but the conversion is not available. An example of such conversion is B2F.

System Action: The stage terminates with return code 391.

User Response: Use two SPEC stages to perform the conversion via an intermediary format; for instance, character.

BPW00392E Error *reason in type record number data: "string"*

Explanation: The string shown has a value that is not valid for the conversion requested.

System Action: The stage terminates with return code 392.

User Response: Check the input file.

The naming conventions for the conversion routines are adopted from REXX. The formats of the input and output types are defined by the characters surrounding the number '2':

- C A character string with the internal representation of the data type. In C2D, for instance, the input character string should be four characters corresponding to a fullword integer in twos-complement notation.

- X A zoned hexadecimal string containing the digits 0 through 9 and the letters a through f (in uppercase or lowercase, or a mixture). There must be an even number of hexadecimal digits in the string; blanks are only allowed at byte boundaries.
- D A zoned decimal integer made up from the digits 0 through 9, possibly with a leading sign. Leading and trailing blanks are allowed, as are blanks between the sign and the number.
- B A string of zeros and ones. The length must be a multiple of eight.
- F A floating point number. Examples of floating point numbers are -5, .03, 2.7e-76.
- V A varying-length character field with a halfword (two bytes) length prefix.
- P A zoned decimal number made up from the digits 0 through 9 with an optional leading sign and an optional decimal point. Leading and trailing blanks are allowed, as are blanks between the sign and the number.
- I A date. If the field contains six characters, it is taken to be year, month, and date (two digits each). When the field is eight characters or longer, it consists of a two-character century followed by a six-character date followed by an optional timestamp which can contain up to six digits. The timestamp contains three two-digit fields for hours, minutes, and seconds.

The reason code describes what went wrong:

- 4 Missing character in number or exponent.
- 8 A character in number or exponent is not valid.
- 12 Exponent overflow or underflow.
- 16 Invalid character in integer or number too large for fullword representation.
- 20 The input field for C2D is longer than 4 bytes and the sign is not propagated.
- 24 The first or last character of a hexadecimal field is blank.
- 28 Odd number of characters in a hexadecimal field.
- 32 Invalid character in a hexadecimal field.
- 36 The number of characters in a bit field is not divisible by 8.
- 40 Character in a bit field is neither 0 nor 1.
- 44 Floating point number is shorter than 2 bytes or longer than 8 bytes.
- 48 A field to be converted to varying character is 64K or longer; the length cannot be expressed as a halfword integer.
- 52 The length of a varying character field is longer than the input field available.
- 56 A number consists of blanks or a sign. That is, it contains no digits.
- 60 Invalid character (not a decimal digit) in number to be packed.
- 64 A packed decimal field contains an invalid sign.
- 68 A packed decimal field contains an invalid digit.
- 72 A packed decimal field is null.
- 76 A Julian date is shorter than six characters, it has an odd number of characters, or it is longer than fourteen characters.
- 80 A Julian date that is eight characters or longer begins with two digits that are less than 19.
- 84 Invalid digit in Julian date (not decimal).
- 88 Month or day is zero or too large.
- 92 A field to be converted to Julian is shorter than three bytes or longer than seven bytes.
- 96 A field to be converted to Julian does not contain X'F' in the low-order nibble of the third or fourth byte.
- 100 A field to be converted to Julian contains an invalid digit (not decimal).
- 104 A field to be converted to Julian contains an invalid value (century field over X'80', which means beyond year 9999; day larger than 365/366; hours, minutes, seconds out of range).
- 108 Hours are larger than 23; minutes or seconds are larger than 59.

BPW00393E Output field too short to contain field length

Explanation: V2C conversion is requested with an explicit output field length. The length is less than 3, which means that no characters can be loaded in the field.

System Action: The stage terminates with return code 393.

BPW00400E Delay word is not acceptable

Explanation: The first word of an input record is not three (or fewer) decimal numbers separated by colons.

System Action: The stage terminates with return code 400.

BPW00401E Input record too short (*number bytes*)

Explanation: For JOIN keylength., the input record was shorter than the specified key length.

System Action: The stage terminates with return code 401.

User Response: Check the input file.

BPW00402I Calling Syntax Exit

Explanation: Pipeline dispatcher trace is active. The stage is defined with a syntax exit which is called.

System Action: None.

BPW00409E Assert failure *code at address*

Explanation: BatchPipeWorks has discovered an internal error. A program check operation exception is forced to indicate a condition which should not occur.

System Action: Message BPW00411I is issued if the information is available.

User Response: Ensure the message level is odd (it is unless you have changed it) and that message BPW00001I is issued to show the stage command in error. Make a note of the code and the following message. Contact your systems support staff.

System Programmer Response: Investigate if corrective service is available.

BPW00413I ... Store *hex: hex*

Explanation: The contents of storage at the point of failure are substituted.

BPW00498E Output descriptor *name* is not valid

Explanation: Reason code X'035C8002' was received when dynamically allocating a SYSOUT data set. The output descriptor contains an invalid character.

System Action: The stage terminates with return code 498.

BPW00499E Output descriptor *name* is not defined

Explanation: Reason code X'04CC8002' was received when dynamically allocating a SYSOUT data set.

System Action: The stage terminates with return code 499.

BPW00500E Dataset *dsname* is partitioned

Explanation: The requested data set is partitioned but no second operand is provided to indicate a specific member.

System Action: The stage terminates with return code 500.

User Response: Select a specific member when allocating the data set.

BPW00501E No dataset is allocated for *ddname*

Explanation: There is no data set allocated for the data definition name shown. The return code 4 is received on the RDJFCB macro.

System Action: The stage terminates with return code 501.

BPW00502E Member *name* already selected by allocation

Explanation: A second operand is found to indicate that a member of a partitioned data set is to be read or written, but the specific member name substituted is specified in the allocation of the data set.

System Action: The stage terminates with return code 502.

User Response: Allocate the complete partitioned data set when referring to members.

BPW00503E Error *number* obtaining dataset control block

Explanation: The return code from OBTAIN is greater than 8. Return code 12 indicates an error reading the volume table of contents. Return code 16 indicates a programming error in *BatchPipeWorks*.

System Action: The stage terminates with return code 503.

BPW00504E Dataset *dsname* does not exist

Explanation: Return code 4 or 8 is received when trying to locate the data set with OBTAIN, which indicates that the volume is not mounted or that the data set does not exist.

System Action: The stage terminates with return code 504.

BPW00505E Dataset *dsname* is not partitioned

Explanation: A member is requested and the data set control block does not indicate partitioned organization.

System Action: The stage terminates with return code 505.

BPW00506E DDNAME *name* is permanently concatenated

Explanation: QSAM does not support permanent concatenations.

System Action: The stage terminates with return code 506.

User Response: Use > to specify the particular data set into which the member should be stored.

BPW00507E Member *name* not found

Explanation: FIND or BLDL gives return code 4, indicating that the requested member is not in the data set.

System Action: The stage terminates with return code 507.

BPW00508E Output descriptor too long: *word*

Explanation: The word is longer than 26 characters. This is the limit for an output descriptor.

System Action: The stage terminates with return code 508.

BPW00514E Record length *number* is over the maximum 32767

Explanation: The first input record is longer than the maximum allowed.

System Action: The stage terminates with return code 514.

User Response: Check the input file.

BPW00515E Not a decimal range: *word*

Explanation: A decimal number or range is expected but the word shown is found.

System Action: The stage terminates with return code 515.

BPW00516E Not a record number or a range of record numbers: *word*

Explanation: Though an acceptable range of decimal numbers, the word shown cannot represent a range of records. The beginning of the range is zero or less, or the end of the range is less than the beginning.

System Action: The stage terminates with return code 516.

BPW00517E Record *number* not present in file

Explanation: The record requested is not in the file.

System Action: The stage terminates with return code 517.

BPW00536E Buffer header destroyed: *hex*

Explanation: BatchPipeWorks has discovered an internal error. The pointer to the next available byte is below the base address of the buffer.

System Action: The stage terminates with return code 536.

User Response: Ensure the message level is odd (it is unless you have changed it) and that message BPW00001I is issued to show the stage command in error. Contact your systems support staff.

System Programmer Response: This is likely to be an error in *BatchPipeWorks*.

BPW00537I Commit level *number*

Explanation: Pipeline dispatcher trace is active. The stage commits to the level shown.

System Action: None.

BPW00538I Query state of *side stream* *stream*

Explanation: Pipeline dispatcher trace is active. The stage requests the status of the stream on the side shown.

System Action: None.

BPW00539E Do not connect unused *side stream stream*

Explanation: A stream is connected that the stage does not use. This is often a symptom of an incorrect placement of a label reference.

A selection stage (for instance, FIND) detects that the secondary input stream is connected. COLLATE and LOOKUP detect that the tertiary input stream is connected. FANIN, FANINANY, MERGE, OVERLAY and SPEC detect a connected output stream other than the primary one. FANOUT detects a connected input stream other than the primary one.

System Action: The stage terminates with return code 539.

User Response: Ensure that the reference to the label that specifies the secondary output stream for a selection stage is after an end-character.

BPW00548I Sever *side*

Explanation: Pipeline dispatcher trace is active. The stage severs the connection on the side shown.

System Action: None.

BPW00549E Return code *number*, reason code *number*, R0 *hex* from IRXINIT

Explanation: The return code and reason code shown are received when trying to find the environment for the REXX program that issued a pipeline specification with Address link or Address attach. The reason code is valid only when the return code is 20.

User Response: Refer to the IRXINIT return and reason codes in MVS/REXX Reference.

System Action: The stage terminates with return code 549.

System Programmer Response: Reason code 24 means that the environment table has too few entries for the number of concurrent REXX programs that the user wishes to run. Refer to MVS/REXX Reference.

BPW00550E Unable to access variables

Explanation: The TSO service routine gives return code 40, indicating that there is no active CLIST environment.

System Action: The stage terminates with return code 550.

BPW00552I SHVBLOCK: *hex*

Explanation: The EXECCOMM parameter list is displayed on two lines.

BPW00553E Return code *number* calling IRXSUBCM *function*

Explanation: The return code shown is received.

System Action: The stage terminates with return code 553.

BPW00554E Stream identifier *string* must not be numeric

Explanation: A stream identifier in a label declaration or label reference is numeric.

System Action: Message BPW00192I is issued if the message level is odd. Pipeline scan continues to the end of the pipeline specification, at which time processing terminates with return code -554.

User Response: Begin a stream identifier with a letter. Stream numbers are assigned when labels are referenced. You cannot specify that a particular stream is to have a particular number; use a stream identifier to refer to a stream without knowing its number.

BPW00556E Asterisk cannot end output column range

Explanation: An output column range ends with an asterisk.

System Action: Return code 556 is set.

User Response: Write a single column to put a field at a particular position, extending as far as required. Use a range to put the field into a particular range of columns, padding or truncating as necessary. Use the operand NEXT instead of a range or column number to abut the field to the contents of the output buffer; this is equivalent to the concatenate operator (||) in REXX. Use the operand NEXTWORD instead of a range or column number to append a blank and the field to the output record built so far. (The blank is suppressed if the record is empty.)

BPW00560I BatchPipes/MVS BatchPipeWorks, 5655-065/SCPSP level *hex*

Explanation: The full-word version identifier is substituted.

System Action: Return code 0 is set.

BPW00563W ANYOF assumed in front of *string*

Explanation: A delimited string that contains more than one character is specified without a keyword to specify how to interpret it. It is most likely that you wish this interpreted as a string rather than as an enumerated list of characters. This message is suppressed if the delimited string contains one character; the question is clearly moot.

User Response: Use the keyword ANYOF to specify a delimited string of characters enumerating characters that match a single character position in the input record. Use STRING to specify that the target is a string of characters that must occur in the sequence shown to match.

BPW00564W Range(s) should be before keyword; put more than one in parentheses

Explanation: A range is specified after the keyword. The order should be reversed.

BPW00566W Use secondary output instead of stack

Explanation: COUNT specifies an option to put the result on the program stack.

User Response: Connect the secondary output stream and process the result without using the program stack. This does not disturb the contents of the stack and does not expose you to problems with multiple COUNT stages.

BPW00573E Last text unit or GDF order not complete

Explanation: The length field of a text unit specifies a count that is larger than the number of bytes remaining in the input record. A GDF order specifies more data than remains in the record.

System Action: The stage terminates with return code 573.

BPW00575E Block padded with *hex*; it should be X'00'

Explanation: A GDF structured field ends in the penultimate position of an input block. The last byte of the block should be zero, but it contains the data shown.

System Action: The stage terminates with return code 575.

BPW00576E Input record is *number* bytes; disk block size is *number* bytes

Explanation: The length of a non-null input record is not 10 plus the disk block size.

System Action: The stage terminates with return code 576.

BPW00577E Error *number* on STIMERM

Explanation: A non-zero return code is received on the macro to set a timer interval on MVS. This message is most likely to be the result of a programming error in *BatchPipeWorks*.

System Action: The stage terminates with return code 577.

User Response: Refer to the return codes for the STIMERM macro instructions in *MVS/ESA Assembler Services Reference*.

BPW00579E Return code *number* from DYNALOC; reason *hex*

Explanation: The error number shown is returned from dynamic allocation. The error number and reason code are substituted.

System Action: The stage terminates with return code 579.

User Response: Refer to the section "SVC99 Return Codes" in *MVS/ESA Authorized Assembler Guide*.

BPW00580I DDNAME allocated: *word*

Explanation: Display the DDNAME allocated to the data set. This message is issued only if the bit for 1024 is turned on in the message level.

BPW00581E >> cannot append to a member

Explanation: >> does not support a member name.

System Action: The stage terminates with return code 581.

BPW00582E Invalid DSNAME "*string*"

Explanation: The data set name is not well-formed (null or missing ending quote) or svc 99 gives error code X'035C' for the data set name.

System Action: The stage terminates with return code 582.

BPW00583E Invalid member name "*string*"

Explanation: The member name is not well-formed (null or no opening parenthesis) or svc 99 gives error code X'035C' for the member name.

System Action: The stage terminates with return code 583.

BPW00584I Enter PIPESTOP, PIPESTALL, or immediate subcommand

Explanation: You have entered the *BatchPipeWorks* attention exit for the first time. If you enter the command PIPEstop. or enter a null line and hit attention again, all stages that wait on an ECB will be signalled to terminate. This is likely to bring the pipeline to a halt.

System Action: None

BPW00585I ECBs posted: *number*; hit attention again to stall the pipeline

Explanation: You have entered the *BatchPipeWorks* attention exit for the second time. If you hit attention again, the pipeline will be stalled. This will terminate the pipeline unless a stage is in a loop.

System Action: All stages waiting on an external event (waiting on an ECB) are signalled to terminate.

BPW00586I Hit attention again to terminate waiting stages

Explanation: You have entered the *BatchPipeWorks* attention exit for the first time. If you hit attention again, all stages that wait on an ECB will be signalled to terminate. This is likely to bring the pipeline to a halt.

System Action: None

BPW00587E**BPW00590E User data length is over 62 or odd (it is *number*)**

Explanation: Explicit user data to STOW with a member in a partitioned data set is either too long or it contains an odd number of characters.

System Action: The stage terminates with return code 590.

BPW00591E Return code *number* reason code *hex* on BLDL

Explanation: The return code shown was received when searching for a member in the directory of a partitioned data set. The contents of register 0 (the reason code) are substituted in hexadecimal.

System Action: The stage terminates with return code 591.

BPW00592E Conflicting allocation for data set *dsname*

Explanation: Dynamic allocation sets return code 02100002, which indicates that the data set is already allocated with a disposition that conflicts with the one requested. < and PDSDIRECT allocate DISP=SHR; > allocates DISP=OLD; >> allocates DISP=MOD.

System Action: The stage terminates with return code 592.

BPW00593E Shared data set *dsname* cannot be allocated exclusive

Explanation: Dynamic allocation sets return code 020C0000, which indicates that a request for exclusive allocation of a shared data set was rejected.

User Response: Ensure you wish to modify the data set. Use the SHR operand to indicate that a shared allocation should be used.

System Action: The stage terminates with return code 593.

BPW00594E Return code *number* reason code *hex* on STOW

Explanation: The return code shown was received when adding a member to the partitioned data set. The contents of register 0 (the reason code) are substituted in hexadecimal.

System Action: The stage terminates with return code 594.

BPW00595E Member name is not allowed for this function

Explanation: The program does not support a member name.

System Action: The stage terminates with return code 595.

BPW00596E Dataset name too long: *name*

Explanation: The data set name plus the prefix (if active) is longer than forty-four characters.

System Action: The stage terminates with return code 596.

BPW00597E Member name or generation too long in DSNNAME *name*

Explanation: The argument contains a left parenthesis, indicating that a generation number or a member is present. There are more than eight characters to the end of the argument.

System Action: The stage terminates with return code 597.

BPW00598E Null member name or generation in DSNNAME *name*

Explanation: The argument contains a left parenthesis, indicating that a generation number or a member is present, but no further characters are present.

System Action: The stage terminates with return code 598.

BPW00599E Null DSNNAME *name*

Explanation: The argument consists of a single quote or two quotes, or the first character is a right parenthesis for the beginning of a member. This is not a valid DSNNAME.

System Action: The stage terminates with return code 599.

BPW00600E Return code *number* on TGET

Explanation: The return code shown is received when reading the terminal.

System Action: The stage terminates with return code 600.

BPW00601E Return code *number* on STFSMODE

Explanation: Full screen mode is not set.

System Action: The stage terminates with return code 601.

BPW00602E Unsupported data set organization *hex*

Explanation: The data set organization is neither physical sequential nor partitioned. The DSORG field is substituted.

System Action: The stage terminates with return code 602.

BPW00603E Unable to read directory for member *name*

Explanation: FIND gives a return code that is neither zero nor four.

System Action: The stage terminates with return code 603.

BPW00604E Null DDNAME

Explanation: The argument begins with the keyword DDNAME, but there are no further characters or the next character is a left parenthesis to indicate a member.

System Action: The stage terminates with return code 604.

BPW00605E DDNAME longer than 8 characters: *word*

Explanation: The argument begins with the keyword DDNAME; it is followed by a word that is more than eight characters.

System Action: The stage terminates with return code 605.

BPW00606E Null member name in DDNAME *name*

Explanation: The argument contains a left parenthesis, indicating that a member is present, but no further characters are present.

System Action: The stage terminates with return code 606.

BPW00607E Member name too long in DDNAME *name*

Explanation: The argument contains a left parenthesis, indicating that a member is present. There are more than eight characters to the end of the argument.

System Action: The stage terminates with return code 607.

BPW00608E Incorrectly specified DSNAME *word*

Explanation: A generation data group number in parentheses is followed by a character that is not a left parenthesis.

System Action: The stage terminates with return code 608.

BPW00609E ABEND *code reason code number*

Explanation: The DCB ABEND exit is driven for the abnormal termination condition substituted.

System Action: The ABEND condition is reset. The stage terminates with return code 609.

BPW00612I Parmlist: *hex*

Explanation: The contents of the EXECCOMM parameter list are substituted.

BPW00613E Pipeline specification is not issued with CALLPIPE

Explanation: The PRODUCER is requested, but the stage is not in a pipeline specification that has been issued with CALLPIPE. Thus, the integrity of the requested variable pool cannot be ensured.

System Action: The stage terminates with return code 613.

BPW00614E Caller's current input stream is not connected

Explanation: The PRODUCER is requested and the stage is in a pipeline specification that has been issued with CALLPIPE, but the caller's currently selected input stream is not connected. Thus, there is no producer stage and hence no variable pool to select.

System Action: The stage terminates with return code 614.

BPW00615E Caller's producer is not connected to caller

Explanation: The PRODUCER is requested, the stage is a pipeline specification that has been issued with CALLPIPE, and the caller's currently selected input stream is connected, but the output stream from the stage has been reconnected, or the stage has selected another output stream. Thus, input records do not correlate with the variable pool requested.

System Action: The stage terminates with return code 615.

BPW00616E Caller's producer is not blocked waiting for output

Explanation: The PRODUCER is requested, the stage is in a pipeline specification that has been issued with CALLPIPE, the caller's currently selected input stream is connected, and the output stream from the stage is connected to the caller, but the stage is not waiting for an output operation to complete. Thus, the integrity of the variable pool cannot be ensured.

System Action: The stage terminates with return code 616.

BPW00620W Unsupported code page *number*

Explanation: A code page number is requested that XLATE does not support.

System Action: The code page number is ignored.

BPW00621W Impossible target string

Explanation: The target string is longer than the column range in which to look for the string; no input record can ever be matched.

System Action: None.

BPW00622E Mask and string are not the same length

Explanation: The two delimited strings specified for MASK are not the same length.

System Action: The stage terminates with return code 622.

BPW00623E Unrecognized relational operator *word*

Explanation: A relational operator is expected, but not found. The valid operators are: ==, !=, <<, <=, >>, >=, EQ, NE, LT, LE, GT, and GE.

System Action: The stage terminates with return code 623.

BPW00624E Premature end of expression

Explanation: An operator or left parenthesis is met at the end of the expression. The expression is not complete.

System Action: The stage terminates with return code 624.

BPW00625E Target expression missing

Explanation: A keyword (for instance TO) is met, indicating that a target should follow, but there are no more arguments.

System Action: The stage terminates with return code 625.

BPW00626E Target data missing for *keyword*

Explanation: A keyword (for instance RECORD) is met, indicating the type of target to match, but there are no more arguments.

System Action: The stage terminates with return code 626.

BPW00627E Null program read from stream

Explanation: The program list contains no lines.

System Action: The stage terminates with return code 627.

BPW00635E Option *word* **conflicts with option** *word*

Explanation: Two incompatible options are specified.

System Action: The stage terminates with return code 635.

BPW00636E Error in encoded pipeline specification; reason code *number*

Explanation: Pass 1 of the scanner found a syntactical error in a pipeline specification. This is an error in *BatchPipeWorks*.

System Action: The stage terminates with return code 636.

User Response: Contact your systems support staff.

System Programmer Response: Report which built-in program issues the message, its argument string, and the reason code. The reason codes are:

- 8 The level of the encoded pipeline block is higher than supported by the version of *BatchPipeWorks* that is being used.
- 4 Pipeline is marked neither for ADDPIPE nor CALLPIPE.

- 1 Null pipeline. A pipeline-begin item is after another pipeline-begin item.
- 2 Stage after end connector. A connector that is not at the beginning of the pipeline has been processed. It is followed by a stage item rather than a pipeline-begin item.
- 3 Label after ending connector. A connector that is not at the beginning of the pipeline has been processed. It is followed by a label-reference item rather than a pipeline-begin item.
- 4 Blank label reference. The label field of a label-reference item has a leading blank.
- 5 More than one end-connector. A connector that is not at the beginning of the pipeline has been processed. It is followed by another connector item rather than a pipeline-begin item.
- 6 Bad input plist. Unrecognized item code. Most likely the item list is not terminated properly.
- 7 Incomplete pipeline. The last pipeline has no stages, no label reference, and at most one connector.
- 8 The specification does not begin with a pipeline-begin item.

BPW00637E Return code *number* on IDENTIFY for *entrypoint*

Explanation: MVS sets the return code shown when *BatchPipeWorks* attempts to identify the entry point shown.

User Response: Ensure that the PIPE command is called correctly; the module must be invoked or loaded in a way that allows the entry point to be identified.

System Programmer Response: Investigate if the entry point name is already in LINKPACK or JOBPACK.

System Action: The stage terminates with return code 637.

BPW00638I SVC 99 plist *hex*

Explanation: Message level 1024 is on or dynamic allocation indicates an error in the parameter list. The parameter list is displayed.

BPW00639E Scaling allowed with packed data only

Explanation: A conversion routine is requested and a left parenthesis follows immediately. This is valid only when converting to or from packed decimal.

System Action: The stage terminates with return code 639.

BPW00640I Text unit *type data*

Explanation: The six-byte type/count/length field is substituted followed by the contents of the first data field. If the data is entirely printable, it is shown as characters; otherwise it is shown in hexadecimal.

BPW00641I Last connected output stream severed by its consumer

Explanation: Tracing is active for the stage. All output streams are now severed. The last output stream was severed by its consumer, rather than by the stage.

BPW00642E ZONE already specified

Explanation: The keyword ZONE is specified with ZONE or it is specified twice with CASEI.

System Action: The stage terminates with return code 642.

BPW00644E Timestamp *word* not valid; reason code *number*

Explanation: An ISO-format timestamp is not valid. The input record must contain a four-character year followed by five two-character fields containing month, day, hour (24-hour clock), minute, and second. It may be followed by one to six decimal digits representing a fraction of a second.

The reason code shows which test has failed:

- 4 The input record is shorter than 14 characters or longer than 20 characters after stripping leading and trailing blanks.
- 8 Year is not a number or the number is less than 1900.
- 12 Month is not a number, it is not positive, or it is greater than 12.
- 16 Day is not a number, it is not positive, or it is greater than 31.
- 20 Hour is not a number, it is negative, or it is greater than 23.
- 24 Minute is not a number, it is negative, or it is greater than 59.
- 28 Second is not a number, it is negative, or it is greater than 59.
- 32 Fraction is not a number, it is negative, or it is greater than 999999.

System Action: The stage terminates with return code 644.

BPW00658E Too many concurrent STIMERM requests

Explanation: Return code X'1C' is received on a timer request. This indicates that 16 requests are already pending for the task. The other timer requests can be issued by DELAY stages or by host commands run through, for example, COMMAND.

System Action: The stage terminates with return code 658.

BPW00660E Unsupported code page *number*

Explanation: A FROM or TO was met, but the following word does not represent a supported code page number.

System Action: The stage terminates with return code 660.

BPW00662E Environment already specified (*keyword* is met)

Explanation: A number or one of the keywords MAIN or PRODUCER have already been specified to designate the environment to use. The keyword that is substituted is met later in the operand list.

System Action: The stage terminates with return code 662.

BPW00663E Unable to generate delimiter for *variable-name*

Explanation: The name of the variable and the characters declared as beginning a comment (these characters are not eligible to be delimiter characters) contain all 256 possible values for an eight-bit byte between them. Thus it is impossible to generate a delimiter character to be used to delimit the name of the variable.

User Response: Specify a shorter comment string.

System Action: The stage terminates with return code 663.

BPW00664E Keyword is not supported when stage is first: *word*

Explanation: The program is used as a first stage of a pipeline. The operand is valid only in a stage that is not first in a pipeline.

System Action: The stage terminates with return code 664.

BPW00665E Exponent is not valid: *word*

Explanation: A numeric constant is being scanned. The letter “E” is met. Either there is no number after the letter or the value of the exponent overflows a 32-bit integer.

System Action: The stage terminates with return code 665.

BPW00666E Syntax error in expression; reason code *number*

Explanation: The expression is not syntactically correct. The number describes the error:

- Internal error (negative length remains to be scanned).
- 0 Unexpected character at the beginning of an expression or after (.
- 1 A digit is expected for the number of a counter, but something else was found.
- 2 A counter was scanned; it was not followed by an operator or a).
- 3 An identifier or an expression has been scanned; it was not followed by an operator or a). Note that assignment operators cannot be immediately to the right of identifiers or expressions.
- 4 ! not followed by =.
- 5 Assignment attempted to something that is not a counter.
- 6 A vertical bar is not followed by another one to make up the logical OR operator. Be sure to use four vertical bars vertical bars if they are also stage separators. This self-escapes them down to two bars that are seen by SPECS.
- 7 An ampersand is not followed by another one to make up the logical AND operator.
- 100 An unpaired : is met.
- 101 Two consecutive ?s are met. Use parentheses to group a conditional expression between the ? and the : of a containing one.

System Action: The stage terminates with return code 666.

BPW00667E Arithmetic overflow

Explanation: The result of evaluating an expression or an intermediary result is beyond the range that can be represented.

System Action: The stage terminates with return code 667.

BPW00668E Dividend is zero

Explanation: A divide operation is attempted with a zero dividend.

System Action: The stage terminates with return code 668.

BPW00670E Picture longer than 255 characters: *picture*

Explanation: The word following PICTURE contains more than 255 characters.

System Action: The stage terminates with return code 670.

BPW00671E Unacceptable character *character in picture picture*

Explanation: The character is not one of the valid characters.

System Action: The stage terminates with return code 671.

BPW00672E Unacceptable picture *picture; unscanned word (reason code number)*

Explanation: An invalid sequence of picture characters is found.

System Action: The stage terminates with return code 672.

BPW00673E Picture has more than one V: *picture*

Explanation: Only one V character is allowed in a picture.

System Action: The stage terminates with return code 673.

BPW00674E Unacceptable drifting sign in picture *picture*

Explanation: A drifting sign character is not the same as the original sign character.

System Action: The stage terminates with return code 674.

BPW00675E Unacceptable zero suppress/protect in picture *picture*

Explanation: A zero suppress or currency protect character is not the same as the previous one.

System Action: The stage terminates with return code 675.

BPW00676E No digits selected in picture *picture*

Explanation: A leading sign is found in a picture, but no digits are selected.

System Action: The stage terminates with return code 676.

BPW00677E No exponent digits in picture *picture*

Explanation: The letter E is met, but no digit select characters follow.

System Action: The stage terminates with return code 677.

BPW00678E More than fifteen exponent digits in picture *picture*

Explanation: The letter E is met followed by more than fifteen digit selectors. The exponent can contain at most ten digits.

System Action: The stage terminates with return code 678.

BPW00679E Exponent too large: *number*

Explanation: The exponent has more significant digits than the picture allows. The exponent is substituted.

System Action: The stage terminates with return code 679.

BPW00680E Record length is zero

Explanation: The first byte of a logical record contains binary zeros. This is not valid, because the minimum record length is one (a record that contains a byte count of one and no data).

System Action: The stage terminates with return code 680.

BPW00681E Input record length (*number*) is over the maximum allowed (*number*)

Explanation: An input record is longer than the maximum allowed.

System Action: The stage terminates with return code 681.

User Response: Check the input file.

BPW00682I TXTunit list *hex*

Explanation: Message level 1024 is on or dynamic allocation indicates an error in the parameter list. The list of pointers to text units is displayed.

BPW00683I STAX return code *number*

Explanation: A non-zero return code is received on a IX=YES stax. macro. The attention exit is not established.

System Programmer Response: Note the conditions under which this message is issued and report the problem to IBM if *BatchPipeWorks* is being used in a supported environment.

BPW00684E Unsupported system variable *word*

Explanation: SYSVAR receives a syntax error when it tries to obtain the variable.

System Action: The stage terminates with return code 684.

BPW00687E Relational operator expected; found *word*

Explanation: A relational operator is expected, but the word is not a supported one.

User Response: Note that the operators are the “strict” operators:

== Equal.
!= Not equal
<< Less than.
<=< Less than or equal.
>> Greater than.
>=> Greater than or equal.

For example, a single equal sign is not a supported relational operator.

BPW00694E Pipeline is not called from a driving program

Explanation: FITR or FITD is invoked in a pipeline set that has not been initialized for fittings. Thus, the stages have nothing with which to interface.

System Action: The stage terminates with return code 694.

BPW00695E Fitting already defined: “*name*”

Explanation: FITTING is issued in a pipeline set that already has a fitting of that name defined.

System Action: The stage terminates with return code 695.

BPW00714E Unacceptable interval *word*

Explanation: The word is not proper for seconds with an optional fraction of microseconds. There may be at most six digits after the period; no component of the number may be negative; and the number must be positive.

System Action: The stage terminates with return code 714.

BPW00715E Not octal: *word*

Explanation: A leading zero is found, but one of the digits is not in the range from zero to seven. The second character is not “x,” which would indicate a hexadecimal value.

System Action: The stage terminates with return code 715.

BPW00718I Returning to application

Explanation: A co-pipe is returning to the application program.

BPW00719I Resuming pipeline

Explanation: The application program has resumed the co-pipe with a request.

BPW00720I Terminating pipeline

Explanation: The application program has resumed the co-pipe without a request parameter list.

BPW00721I RPL *hex*

Explanation: The Request Parameter List is displayed.

BPW00722I Resolved fitting *identifier*

Explanation: A Request Parameter List is paired with an active FITTING stage.

BPW00723I Fitting *identifier* not resolved

Explanation: A Request Parameter List was not paired with an active FITTING stage. No current stage is active for the fitting.

BPW00724I Posting fitting *identifier*

Explanation: A Request Parameter List is paired with an active FITTING stage and it has work to do.

BPW00725I Returning to the pipeline dispatcher

Explanation: All Request Parameter Lists have been processed and the FITTING stages posted to wake up.

BPW00726I No RPLs changed state

Explanation: All Request Parameter Lists have been processed, but none changed state. Thus, the application has cheated.

System Action: The status code is set accordingly.

BPW00727I *string*

Explanation: A tracing message.

System Action: The stage terminates with return code 727.

BPW00728I *number description*

Explanation: Statistics are requested. The contents of a counter are displayed.

BPW00729I Letting dispatcher wait

Explanation: The application has indicated that it does not wish to regain control until a particular FITTING stage has produced or consumed a record.

BPW00730E No data sets found matching *dsname*

Explanation: The return code from IX=YES locate. was eight. This indicates that no matching entries were found.

User Response: The leading qualification should not end with a period.

System Action: The stage terminates with return code 730.

BPW00731E Return code *number* on SVC 26

Explanation: The return code from IX=YES locate. was neither zero nor eight.

System Action: The stage terminates with return code 731.

BPW00741E Record format "*character*" is not supported

Explanation: The record format is neither F nor V. A blank indicates OS-format file; a hyphen indicates that the file is migrated.

System Action: The stage terminates with return code 741.

BPW00743I File "*file*"

Explanation: Open failed for a file. The file name parameter is shown.

BPW00744I Open flags *words*

Explanation: Open failed for a file. The open flags parameter is shown.

BPW00750E Incorrect input block format

Explanation: DEBLOCK monitor. has read a block that contains a length field of binary zeros, but the remainder of the block does not consist entirely of binary zeros.

BPW00755E Offset not shorter than width

Explanation: The length of the offset specified (either as a number or as the length of the delimited string) is equal to or greater than the width.

System Action: The stage terminates with return code 755.

BPW00756W Use the := assignment operator instead of =

Explanation: A single equal sign is scanned.

User Response: Change to use the colon-equal operator.

BPW00757W Use the ~ operator instead of !

Explanation: An exclamation point is scanned.

User Response: Change to use the not operator.

BPW00758W Do not double-up relational operators

Explanation: A double bar or a double ampersand is scanned.

User Response: Change to use a single operator character.

BPW00759E Incompatible types

Explanation: An operation is requested between a string and a counter. Relational operators must be between like types. Strings cannot be used with computational operators.

System Action: The stage terminates with return code 759.

BPW00760E No data will be available for input field

Explanation: An input range is specified after EOF without SELECT second. in effect. Thus, there are no data available to SPECS to supply.

User Response: Use SELECT second. to refer to the second reading station, where a copy of the last record is. However, if you were not using the second reading station and the field you require can be stored in a counter, it is more efficient to save the value in a counter while processing the detail record and then refer to the contents of this counter after the EOF item.

System Action: The stage terminates with return code 760.

BPW00761E Different key fields not allowed with AUTOADD

Explanation: AUTOADD was specified and the key field is defined in a different place in the detail and in the master records. This would make adding the record ambiguous.

User Response: Use SPECS to move the key field in the master or the detail records.

System Action: The stage terminates with return code 761.

BPW00762E Return code *number* reason code *number* from TSO

Explanation: The TSO command service routine (IX=YES IKJEFTSR) gave the return code and reason code shown.

User Response: Refer to *TSO/E Programming Services*.

System Action: The stage terminates with return code 762.

BPW00763E File token *word* is not valid (reason code *number*)

Explanation: The file token could not be converted from hexadecimal to binary. The reason codes are:

Reason codes 1 and 2 should not occur.

System Action: The stage terminates with return code 763.

BPW00764E Timestamp too short: *string*

Explanation: The time stamp must contain at least eight digits.

System Action: The stage terminates with return code 764.

BPW00765E Timestamp too long: *string*

Explanation: The time stamp must contain at most fourteen digits.

System Action: The stage terminates with return code 765.

BPW00766E Century incorrect in timestamp: *string*

Explanation: The first two characters of the timestamp are less than 19.

System Action: The stage terminates with return code 766.

BPW00767E Non-numeric character in timestamp: *string*

Explanation: A character of the time stamp is not numeric.

User Response: The timestamp is specified as a sequence of digits without the usual delimiter characters.

System Action: The stage terminates with return code 767.

BPW00769E SYSOUT Class *char* is not a letter.

Explanation: A single character is specified, which is neither an asterisk, a letter, nor a digit. Or the keyword CLASS is specified and not followed by a one-character operand.

User Response: Use the keyword OUTDESC to specify a one-character output descriptor.

System Action: The stage terminates with return code 769.

BPW00770E Period missing in destination *word*

Explanation: A DESTINATION keyword is met., but the following word contains no period.

User Response: Make sure the destination contains both a system ID (also known as a node ID) and a user ID:

| punch dest node.userid

System Action: The stage terminates with return code 770.

BPW00771E Leading period in destination *word*

Explanation: The first character of the destination is a period. This implies a null node ID.

System Action: The stage terminates with return code 771.

BPW00772E Ending period in destination *word*

Explanation: The last character of the destination is a period. This implies a null user ID.

System Action: The stage terminates with return code 772.

BPW00773E Node *word* **is not defined to JES**

Explanation: The first component of the destination is not known to JES.

System Action: The stage terminates with return code 773.

BPW00774E Syntax error: *explanation*

Explanation: REXX signalled a syntax error. The error text is substituted.

System Action: The stage terminates with return code 774.

BPW00792E Fitting placement incompatible with RPL

Explanation: A fitting Request Parameter List that reference the stage specifies an initial operation (read or write) that is incompatible with the placement of the FITTING stage.

System Action: The stage terminates with return code 792.

BPW00793E Initial RPL state is not valid: *number*

Explanation: A fitting Request Parameter List that reference the stage specifies an initial state that is neither IDLE, READ, nor WRITE.

System Action: The stage terminates with return code 793.

BPW00794E More than one RPL refers to stage

Explanation: Two fitting Request Parameter Lists reference the stage. This is an error, because the stage only supports one request at a time.

System Action: The stage terminates with return code 794.

BPW00795E Pipeline already active on process *number* **thread** *number*

Explanation: A PIPE command is issued while *BatchPipeWorks* is active on the process and thread shown. *BatchPipeWorks* does not support concurrent invocations on multiple threads.

User Response: Issue PIPE commands from the commands process.

System Action: The stage terminates with return code 795.

**BPW00797E Data has incorrect length for BPW stage. Input record was length *number*
Required length must be *number***

Explanation:

User Response:

System Action: The stage terminates with return code 797.

**BPW00798E Data has incorrect length for BPW stage. Input record was length *number*
Required length must be *number***

Explanation:

User Response:

System Action: The stage terminates with return code 798.

**BPW00799E Data has incorrect length for BPW stage. Input record was length *number*
Required length must be *number***

Explanation:

User Response:

System Action: The stage terminates with return code 799.

BPW00800E BPread and BPWrite stages may only be used within fittings

Explanation: The BPread and BPWrite stages are for use within BatchPipes fittings only.

User Response: Ensure that you only use these stages from within BatchPipes fittings.

System Action: The stage terminates with return code 800.

BPW00801E BPread and BPWrite stages may only be used once

Explanation: The stage detected that it was specified more than once in the pipeline.

User Response: Ensure that you only use these stages once within a pipeline.

System Action: The stage terminates with return code 801.

BPW00802E Length of record on primary input stream is not valid

Explanation: The stage detected an unexpected length for the record on the primary input stream.

User Response: Ensure that records are of the proper length for the BatchPipes pipe.

System Action: The stage terminates with return code 802.

BPW00803E The BPCOPY stage may only be used once

Explanation: The stage detected that it was specified more than once in the pipeline.

User Response: Ensure that you only use these stages once within a pipeline.

System Action: The stage terminates with return code 803.

BPW00804E BPCOPY is mutually exclusive with BPWRITE and BPREAD

Explanation: The stage detected that it was specified in the same pipeline with a mutually exclusive stage.

User Response: Ensure that you only use either BPread, BPWrite or BPCOPY.

System Action: The stage terminates with return code 804.

BPW00805E BPCOPY stage may only be used within fittings

Explanation: The BPCOPY stage is for use within BatchPipes fittings only.

User Response: Ensure that you only use this stage from within BatchPipes fittings.

System Action: The stage terminates with return code 805.

BPW01032E Not a valid field identifier: *word*

Explanation: One letter is required for the field identifier. The word is longer than one character or it does not contain one of the characters from a to z (either case).

System Action: The stage terminates with return code 1032.

BPW01033E Field *ID* is not defined

Explanation: The field being referenced has not been declared.

System Action: The stage terminates with return code 1033.

Glossary of BatchPipeWorks Terms

Glossary terms are defined as they are used in this book. If you can't find the term you are looking for, refer to the index or to the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

B

BatchPipeWorks. A function within BatchPipes that allows you to solve a complex problem by breaking the problem up into a series of smaller, less complex programs. A program can be one of the stage commands BatchPipeWorks supplies or can be a stage command a user writes.

BatchPipeWorks stage command. A stage command supplied by BatchPipeWorks.

C

commit level. Allows stages to coordinate processing so that any one of them can communicate to other stages that an error has been detected.

connected. A stream is connected if it is attached to a stream from another stage, such that data can flow from one stage to another. If a stream is not connected, then data cannot flow from one stage to another.

connector. A group of characters that connects the input or output stream of the stage issuing an ADDPIPE or CALLPIPE subcommand with the pipeline specified as the operand. A connector can be used only at the beginning or end of a pipeline specified by the ADDPIPE or CALLPIPE subcommand.

consume. To read a record from an input stream and remove the record from that input stream. Once a record has been consumed by a stage, it cannot be read again by that stage.

D

defined (streams). Streams are defined by specifying a stage, using a label reference, or using the ADDSTREAM subcommand. Once a stream is defined, it can never be undefined.

delimiting character. A character used before and after a string of characters to define a delimited string. The delimiting character cannot be blank and it must not occur within the string. Two adjacent delimiting characters represent a null string.

device driver. A stage command that reads data from or writes data to a device, MVS, or other host environments. There are also device drivers that can obtain information about MVS files or reference REXX and EXEC 2 variables.

E

end character. A character that BatchPipeWorks uses to identify separating and diverging streams in a multistream pipeline.

end of file. Means no more data can flow through a stream because the stream has been severed.

F

filter stage command. A stage command that transforms the data it processes without interfacing with devices or host environments.

fitting. A pipeline associated with an application in a BatchPipes pipeline; the fitting is invoked by BatchPipes on behalf of the application.

H

half-pipe fitting. A pipeline associated with an application that is not in a BatchPipes pipeline; the half-pipe fitting is invoked by BatchPipes on behalf of the application.

I

input stream. A stream that flows into a stage.

L

label. A string of up to eight alphanumeric characters that identifies where streams enter and leave a stage that has multiple streams. A label must be immediately followed by a stream identifier or a colon with no intervening blanks.

label definition. The first occurrence of a label in a pipeline. A label definition allows BatchPipeWorks to connect other pipeline to the labeled stage.

label reference. A second or subsequent occurrence of a label in a pipeline. A label reference defines an additional input and output stream for the stage.

M

message level. A number specifying which informational messages you can receive from BatchPipeWorks. These messages include: BPW001I, BPW002I, BPW003I, and BPW004I. Depending on the value of the message level, you can receive these messages along with other messages. These messages help to determine what stage was running, what subcommand was running, and what pipeline was running that caused the previous message to be issued.

multistream pipeline. A pipeline that contains at least one stage that has more than one input stream, more than one output stream, or both.

N

null string. A delimited string formed by two adjacent delimiters. Null strings have a length of zero.

O

option. A keyword that controls the execution of the PIPE command, ADDPIPE subcommand, CALLPIPE subcommand, or a stage.

When you specify the options immediately following PIPE, ADDPIPE, or CALLPIPE, the options control the entire PIPE command, ADDPIPE subcommand, or CALLPIPE subcommand. When you specify the options at the beginning of a stage, the options control only that stage. Options specified on a stage override options specified after PIPE, ADDPIPE, or CALLPIPE.

output stream. A stream that flows out of a stage.

P

pipeline. A series of programs called stages through which data flows. The output of a stage automatically becomes the input to the next stage.

pipeline set. A series of pipelines.

pipeline specification. Combination of stage commands and stage separators that tells BatchPipeWorks what to do.

pipeline subcommand. A command that can be issued only from a user-written stage command. Subcommands process data and interact with the calling pipeline. For example, you can read data from the input stream, process the data, and write data to the output stream using subcommands.

primary input stream. Input stream number 0. It is defined by specifying a stage.

primary output stream. Output stream number 0. It is defined by specifying a stage.

S

secondary input stream. Input stream number 1. It is defined the second time a label is specified (that is, the first time a label is referenced) or by using the ADDSTREAM subcommand.

secondary output stream. Output stream number 1. It is defined the second time a label is specified (that is, the first time a label is referenced) or by using the ADDSTREAM subcommand.

sever. To cause a stream to become unconnected.

short. To cause a stream to bypass a stage by connecting the output of the preceding stage to the input of the following stage. The shorted stage can neither read from the input stream nor write to the output stream.

stage. A program in a pipeline. A pipeline consists of one or more stages, each of which is a stage command. A stage reads input records, processes those records, and writes output records. A stage reads input records from a device or from the output of the previous stage. A stage writes output records to a device or to the input of the next stage.

stage command. A command that manipulates data, reads data from a device or host interface, or writes data to a device or host interface. BatchPipeWorks provides over 100 stage commands; you can also write your own using the subcommands that BatchPipeWorks provides.

stage separator. A character that indicates the end of one stage of a pipeline and the beginning of the next stage. The default stage separator character is the solid vertical bar (|).

stall. A condition in a multistream pipeline in which no stages can run.

stream. A flow of data records into a stage and out of a stage. In each stage, streams are numbered in ascending order starting with stream 0. In addition to having a stream number, a stream can also be assigned an identifying name.

stream name. A symbolic name for a stream, consisting of a period, one to four alphabetic characters or a combination of alphabetic characters and digits that includes at least one alphabetic character, and a colon, with no intervening blanks.

stream number. A number of a particular input or output stream, where 0 (zero) is the primary stream, 1 (one) is the secondary stream, and so on.

subroutine pipeline. The operand of the CALLPIPE subcommand.

T

target. A group of characters, a delimited string, a number of records, or a displacement value. A target can be specified as an argument on some BatchPipeWorks stage commands or subcommands.

tertiary input stream. Input stream number 2. It is defined the third time a label is specified (that is, the second time a label is referenced) or by using the ADDSTREAM subcommand.

tertiary output stream. Output stream number 2. It is defined the third time a label is specified (that is, the second time a label is referenced) or by using the ADDSTREAM subcommand.

U

user-written stage command. A stage command that the user writes. A user can write a stage command to perform a task for which there is no BatchPipeWorks stage command. A user-written stage command:

- Must be written in the REXX language.
- Can contain subcommands, REXX instructions, and host commands. To use the host commands, you must specify the REXX ADDRESS instruction to identify the host environment.

Index

Special Characters

- _ indicating blanks in searches 44
- , as a delimiter 41
- / as a delimiter 41
- * (asterisk)
 - JOIN stage command 55
- *: as a connector 153, 166, 169
- < (Read Data Set) stage command
 - description 82
- > (Rewrite Data Set) stage command
 - description 82
- >> (Append to Data Set) stage command
 - description 83
- | stage separator 6
- || as a REXX concatenation symbol 133

A

- accessing exec variables 87
- ADD REXX example user-written stage
 - command 148
- adding a pipeline 154
- adding to a file 83
- ADDDPIPE subcommand
 - description 154
- alignment operands of SPECS stage command 67
- altering the content of a record 50—72
- APPEND stage command 89
- appending data to a file 83
- arbitrary character, specifying 44
- arguments, processing 163
- arranging
 - record contents 59
 - records 74
- arrays, accessing 87
- asterisk (*)
 - JOIN stage command 55
- AUTHOR REXX example user-written stage
 - command 147, 164
- avoiding a stall 20, 102

B

- B2C operand
 - SPECS stage command 68
- BACKUP REXX example (shows ADDPIPE) 157
- balance a BatchPipes pipeline
 - illustration of 32
- BatchPipes
 - concepts
 - examples 34—36
 - fittings 27

BatchPipes (continued)

- half-pipe fittings 36
- how to code 29
- restrictions 32—34

BatchPipeWorks

- summary of 201

BatchPipeWorks environment for user-written stage commands 141

blank

- as arbitrary characters 44
- as pad characters in SPECS 61
- indicating in searches 44
- specifying in the XLATE stage command 52
- stripping from records 57

blank record, writing with the OUTPUT subcommand 165

block descriptor word 182

BLOCK stage command

- FIXED operand 180
- LINEND operand 183
- VBS operand 183

blocked stage 112

blocking and deblocking records 179—188

BPCOPY stage command

- description 84, 85
- example of use 31

BPREAD stage command

- description 84
- example of use 31

BPWRITE stage command

- description 85
- example 36
- example of use 31

BUFFER stage command

- description 77
- fixing stalls with 110

buffering records

- BUFFER stage command 77
- SORT stage command 20, 74

BYTES operand

- COUNT stage command 73

bytes, counting the number of 73

C

C2B operand

- SPECS stage command 68

C2X operand

- SPECS stage command 68

CALLPIPE subcommand

- using multiple input and output streams 153
- writing subroutine pipelines 165

- CANDELAY example exec** 116
- capital letters, translating records to** 51
- case insensitivity** 39
- case sensitivity** 39
- case, translating to upper or lower** 51
- catenating records** 54
- CENTER operand**
 - SPECS stage command 67
- CHANGE stage command**
 - description 58
- changing a file** 82
- changing the contents of records** 58
- changing the map of the pipeline** 155
- character**
 - counting the number of 73
 - translating to uppercase or lowercase 51
- character, arbitrary** 44
- CHOP stage command**
 - description 56
- chopping records** 56
- column number, negative** 71
- column range**
 - for the CHANGE stage command 58
 - for the NLOCATE stage command 43
 - for the SORT stage command 75
 - for the SPECS stage command 60
 - for the XLATE stage command 51
- column reference, relative** 71
- combining inputs from device drivers** 89
- combining records** 54
- combining streams** 96, 98
- COMBO REXX example (shows CALLPIPE)** 174
- comma (,) as delimiters** 41
- COMMAND stage command**
 - description 137
- command, issuing from a pipeline** 136
- computing the number of characters, words, or records** 73
- computing the number of duplicate records** 76
- concatenating records** 54
- concatenation symbol in REXX (||)** 133
- concepts for user-written stage commands** 139
- connecting records** 54
- connecting streams** 12
- connection variations for ADDPIPE** 155
- connections, stream**
 - restoring a connection stacked with ADDPIPE subcommand 158
 - severing 153, 158
 - stacking connections with ADDPIPE subcommand 158
- connector**
 - format of 153, 166, 169
 - using with ADDPIPE 155
 - using with CALLPIPE 166, 169
- consuming records** 112
- contents of records**
 - changing 58
 - rearranging 59
- continuation character, using** 132
- continuing pipelines on several exec lines** 132
- controlling messages from tracing** 196
- conversion operands of SPECS stage command** 68
- converting records to uppercase or lowercase characters** 51
- COUNT operand**
 - SORT stage command 76
- COUNT stage command**
 - using with primary output stream 73
 - using with secondary output stream 103
- counting**
 - characters and words 73
 - duplicate records 76
 - records 73
- COUNTWDS REXX example subroutine**
 - pipeline 168
- creating a file** 82
- creating streams** 13
- currently selected stream** 152

D

- DEBLOCK stage command**
 - FIXED operand 181
 - LINEND operand 184
 - V operand 183
- deblocked variable records** 182
- deblocking and blocking records** 179—188
- debugging pipelines** 189—199
- defining streams** 13
- defining the layout of output records** 59
- DELAY example exec** 117
- DELAY stage command**
- delaying records** 113
- delimiter**
 - between stages 6
 - for the LOCATE stage command 41
- DESCENDING operand**
 - SORT stage command 75
- detail record** 107
- DETAILS operand of LOOKUP** 107
- device driver**
 - combining inputs from 89
 - description 7, 79—91
- device drivers**
 - definition 7
- discarding duplicate records** 46, 76
- discarding unique records** 47
- disconnecting streams** 153, 158
- dispatching stages added by ADDPIPE** 158

- displaying all nonzero return codes 198
- dividing records 54
- DROP stage command** 49
- DUPLF example exec for UNIQUE MULTIPLE** 47
- duplicate records**
 - counting 76
 - discarding 46, 76
- DUPLICATE stage command**
 - description 73
- duplicating records** 73

E

- echoing lines on the terminal** 79
- end character**
 - specifying on the PIPE command 11
- ENDCHAR subparameter**
 - example 36
- environment for user-written stage commands** 141
- erasing duplicate records** 46, 76
- erasing unique records** 47
- example of**
 - exec**
 - CANDELAY (shows delayed commands) 116
 - DELAY (shows delayed commands) 117
 - DOIT (shows DELAY)
 - DUPMEM (shows UNIQUE MULTIPLE) 47
 - LFD (shows FANINANY) 96
 - LOOKSTR (shows LOOKUP) 108
 - NODELAY (shows commands not delayed) 114
 - OVERLAY (shows OVERLAY) 100
 - SELECT (shows SPECS SELECT) 102
 - STAGESEP (displays stage separator) 6
 - TIME (shows NAME option) 197
 - WORDUSE (shows COUNT) 104
 - user-written stage command
 - ADD REXX 148
 - AUTHOR REXX 147, 164
 - BACKUP REXX (shows ADDPIPE subcommand) 157
 - COMBO REXX (shows CALLPIPE subcommand) 174
 - COUNTWDS REXX (example subroutine pipeline) 168
 - FIXED REXX (shows CALLPIPE subcommand) 166
 - HOLD REXX 143
 - LOCDEPT REXX 154
 - LOGIT REXX (shows CALLPIPE subcommand) 169
 - MYFANOUT REXX 149, 151
 - REVIT REXX 141
 - SECPARM REXX (shows ADDPIPE subcommand) 158
 - SEELOG REXX (shows CALLPIPE subcommand) 170, 171, 172
 - TITLE REXX 165

- example of** (*continued*)
 - user-written stage command (*continued*)
 - TRACER REXX (shows ADDPIPE subcommand) 159
 - TRACING REXX (shows CALLPIPE subcommand) 176
 - UENG REXX (shows CALLPIPE subcommand) 173
 - VALIDATE REXX (shows LOOKUP stage command) 109

exec

- accessing variables of 87
- using pipelines in 132

exec, example

- CANDELAY (shows delayed commands) 116
- DELAY (shows delayed commands) 117
- DUPMEM (shows UNIQUE MULTIPLE) 47
- LFD (shows FANINANY) 96
- LOOKSTR (shows LOOKUP) 108
- NODELAY (shows commands not delayed) 114
- OVERLAY (shows OVERLAY) 100
- SELECT (shows SPECS SELECT) 102
- STAGESEP (displays stage separator) 6
- TIME (shows NAME option) 197
- WORDUSE (shows COUNT) 104

EXECIO command

- using the PIPE command as an alternative 133

executing commands from a pipeline 136

executing records as commands 137

expanding records 56

F

FANIN stage command

- description 98

FANINANY stage command

- description 96
- fixing stalls with 111

FANOUT stage command

- description 93

FBLOCK stage command

- creating fixed-format records 186

field

- moving within records 59

FIELDS operand

- SPECS stage command 63

fields, range of 63

file

- appending data to 83
- changing 82
- combining two 90, 91
- creating 82
- getting facts about 85
- reading data from 82
- writing data to 82

- file, tracing to** 191
- fill characters used in SPECS** 61
- filling records** 56
- filter job, in BatchPipes**
 - example 25
 - example of JCL 125
 - guidance 124
 - use BatchPipeWorks with 25
- filter stage command**
 - definition 8
 - description 8
 - using 39
- FIND stage command**
 - description 44
- finding strings in records** 39—48
- FIT subparameter**
 - define a fitting in JCL 29
 - example 36
 - use of 29
- FITDD subparameter**
 - use of 30
- fitting**
 - add comments to FITDD subparameter 31
 - description 23
 - example 23
 - example of adding comments 31
- FIXED operand**
 - BLOCK stage command 180
 - DEBLOCK stage command 181
- FIXED REXX example (shows CALLPIPE)** 166
- fixed-format blocks, blocking and deblocking** 180, 186
- fixing stalls** 20
- folding records to uppercase** 51
- format**
 - of connectors 153, 166, 169
- formatting data on records** 59
- FRLABEL stage command**
 - description 45

G

- GDG with BatchPipes**
 - use a fitting to make a copy 35
- getting facts about a file** 85
- global options on the pipeline specification**
 - description 19
- glossary information** 265

H

- half-pipe fitting**
 - description 24
 - example 25
- hexadecimal values**
 - specifying on the XLATE stage command 52

- HOLD REXX example user-written stage command** 143
- HOLD REXX, fixing stalls with** 111
- hours, specifying with DELAY stage command**
- how a pipeline runs** 141
- how stage commands use multiple streams** 9

I

- identifying streams** 97
- ignoring characters in searches** 44
- illustration of a BatchPipes pipeline** 24
- input connector** 153, 155, 166, 169
- input operand of SPECS** 60
- input range**
 - for the LOCATE stage command 41
- input stream**
 - description 4
 - reading records from 144
- input stream, primary** 9
- input stream, secondary** 9
- inserting stages in pipelines** 161, 166
- interacting with BatchPipeWorks from a stage command** 139
- issuing TSO commands from a pipeline** 136

J

- JOBLOG stage command**
 - example 36
- JOIN stage command**
 - description 54
- joining records** 54
- joining streams** 96, 98

L

- label**
 - definition 12
- label definition**
 - description 12
- label reference**
 - description 12
- LAST operand**
 - of DROP 49
 - of TAKE 49
- layout of output records, defining** 59
- leading character, removing** 57
- LEFT operand**
 - SPECS stage command 67
- length, selecting records by** 42, 43
- LFD example exec for FANINANY** 96
- limiting the range of CHANGE** 58
- limiting the range of LOCATE** 41
- limiting the range of XLATE** 51

- line**
 - See record
- line-end character**
 - example 183
 - specifying for BLOCK 183
 - specifying for DEBLOCK 184
 - using 180
- line-end character format, blocking and deblocking** 183
- LINEND operand**
 - BLOCK stage command 183
 - DEBLOCK stage command 184
- LINES operand**
 - COUNT stage command 73
- linking records** 54
- LISTERR option**
 - PIPE command 198
- LISTMRG example exec for MERGE** 106
- LITERAL stage command**
 - description 40, 80
- literals, writing to a pipeline** 40, 80
- literals, writing with SPECS** 64
- LOCATE stage command**
 - description 40
 - OR operation 96
- LOCDEPT REXX example user-written stage command** 154
- LOGIT REXX example (shows CALLPIPE)** 169
- looking for strings in records** 39—48, 108
- LOOKUP stage command**
 - description 107
- lowercase characters, translating records to** 51

M

- maintaining relative order of records** 112
- manipulating output records** 59
- map of pipeline, redrawing** 155
- mapping the contents of records** 59
- master record** 107
- MAXSTREAM subcommand**
 - description 150
- MERGE stage command**
 - description 105
- message**
 - controlling tracing messages 196
- moving fields within records** 59
- MULTIPLE operand**
 - UNIQUE stage command 47
- multiple pipelines, writing** 11
- Multiple Virtual Storage (MVS)**
 - variable-format records 182
- multistream pipeline**
 - stage commands for 93—110
 - subcommands for 148—163
 - using 9—19, 93—119

- MVS (Multiple Virtual Storage)**
 - See Multiple Virtual Storage (MVS)
- MYFANOUT REXX example user-written stage command** 149, 151

N

- NAME option**
 - PIPE command 197
- names for streams** 98
- naming pipelines** 197
- negative column number** 71
- negative locate** 43
- NEXT operand**
 - SPECS stage command 66
- NEXTWORD operand**
 - SPECS stage command 67
- NFIND (NOTFIND) stage command**
 - description 44
- NLOCATE (NOTLOCATE) stage command**
 - description 43
- NODELAY example exec** 114
- NOMSGLEVEL option** 196
- not locate** 43
- number of a stage** 147
- number of characters, words, or records, counting** 73
- number of duplicate records, counting** 76
- numbers for streams** 97

O

- operands, processing** 163
- option**
 - specifying on the PIPE command 19
- OR function for LOCATE** 96
- order of records**
 - maintaining 112
 - predicting 113
- ordering records** 74
- output connector** 153, 155, 166, 169
- output operand of SPECS** 60
- output stream**
 - definition 4
 - writing records to 144
- output stream, primary** 9
- output stream, secondary** 9
- OUTPUT subcommand**
 - description 144
 - examples 143
- OVERLAY example exec for OVERLAY stage command** 100
- OVERLAY stage command**
 - description 100
- overlying data** 66

P

PACK stage command

description 185

packed records, deblocking 184

packing records 184

pad characters used in SPECS 61

PAD stage command

description 56

padding records 56

parentheses () as used in the CHANGE stage command 59

peeking at a record 145

PEEKTO subcommand 145, 173

pipe balance

placement of fittings 32

PIPE command

debugging 189—199

description 131

ENDCHAR option 11

LISTERR option 198

NAME option 197

NOMSGLEVEL option 196

specifying options on 19

STAGESEP option 19

TRACE option 190

tracing execution of stages on 189

pipeline

adding 154

continuing on several exec lines 132

debugging 189—199

description 3

how records are processed by 20, 141

in execs 132

inserting stages into 161, 166

multistream 9—19, 93—119

naming 197

preserving 132

return codes from 135

stalling of 20

tracing 189

used in BatchPipes filter jobs 124—129

used in BatchPipes fittings 23—38

using labels within 12

virtual storage used by 20

pipeline basics 21

pipeline set

definition 11

predicting relative record order 113

PREFACE stage command 91

preserving a pipeline 132

primary input stream

multistream pipeline 9

primary output stream

multistream pipeline 9

primary stream

definition 9

processing arguments in user-written stage

commands 163

Q

QSAM stage command 202

R

range

for the CHANGE stage command 58

for the NLOCATE stage command 43

for the SORT stage command 75

for the SPECS stage command 60

for the XLATE stage command 51

READ operand

SPECS stage command 69

reading exec variables 87, 89

reading from a file 82

reading from the terminal 80

reading input stream records 144

reading lines with SPECS READ operand 69

READTO subcommand

description 144

examples 143

rearranging the contents of records 59

RECNO operand

SPECS stage command 65

reconnecting streams 158

record

blocking 179—188

buffering 20, 77

changing 50—72

chopping 56

consuming 112

counting the number of 73

deblocking 179—188

delaying 113

description

discarding duplicates 46, 76

discarding unique 47

duplicating 73

executing as commands 137, 192

expanding 56

joining 53

looking at end of 72

maintaining relative order 112

overlying 100

packing 184

padding 56

peeking at 145

predicting relative order 113

reading from a file 144

rearranging the contents of 59

record *(continued)*

- removing
 - blank lines 42
 - leading characters 57
 - trailing characters 57
- selecting
 - by content 39—48
 - by length 42, 43
 - by position 48—50
- sorting 74
- splitting 53
- translating 51
- unpacking 185
- writing to a file 40, 144
- record descriptor word** 180, 182
- records that span blocks** 187
- redrawing a pipeline map** 155
- reference (for LOOKUP)** 107
- referencing labels** 13
- relative column reference** 71
- remapping the contents of records** 59
- removing blank lines** 42
- removing duplicate records** 46, 76
- removing leading characters** 57
- removing trailing characters** 57
- removing unique records** 47
- replacing one string with another** 58
- restoring connections** 158
- return code**
 - displaying all nonzero 198
 - from pipelines 135
 - use of, in user-written stage commands 140
- return code 12, meaning of**
- reusing sequences of stage commands** 167
- REVIT REXX example user-written stage command** 141
- REXX concatenation symbol (||)** 133
- REXX continuation character** 132
- REXX language usage in user-written stage commands** 139
- REXX stage command**
 - running your own stage command 144
- REXX variable**
 - accessing 87
 - putting command results into 137
- RIGHT operand**
 - SPECS stage command 67
- running commands from a pipeline** 136
- RUNPIPE stage command** 191

S

sample exec

- CANDELAY (shows delayed commands) 116
- DELAY (shows delayed commands) 117
- DUPMEM (shows UNIQUE MULTIPLE) 47

sample exec *(continued)*

- LFD (shows FANINANY) 96
- LOOKSTR (shows LOOKUP) 108
- NODELAY (shows commands not delayed) 114
- OVERLAY (shows OVERLAY) 100
- SELECT (shows SPECS SELECT) 102
- STAGESEP (displays stage separator) 6
- TIME (shows NAME option) 197
- WORDUSE (shows COUNT) 104
- screen, displaying results on** 79
- searching for strings in records** 39—48
- secondary input stream**
 - connecting to 16, 17
 - description 9
- secondary inputs, connecting to** 16, 17
- secondary output stream**
 - connecting to 13, 17
 - description 9
- secondary outputs, connecting to** 13, 17
- SECPARM REXX example (shows ADDPIPE)** 158
- SEELOG REXX example (shows CALLPIPE)** 170, 171, 172
- segment descriptor word** 182
- segments of records** 182
- SELECT ANYINPUT stage command**
- SELECT example exec for SPECS** 102
- SELECT operand**
 - SPECS stage command 101
- SELECT subcommand**
 - description 149
- selecting records by content** 39—48
- selecting records by position** 48—50
- selecting the current stream** 149
- SEVER subcommand**
 - description 153
 - INPUT operand 158
- severing streams** 153, 158
- SHORT subcommand**
 - description 145
- slashes (/) as delimiters** 41
- small letters, translating records to** 51
- snapshots of data, as used in debugging** 196
- SORT stage command**
 - description 74
 - fixing stalls with 112
- SORT UNIQUE stage command as compared to UNIQUE stage command** 46
- sorting pipeline records** 74
- spacing data on records** 61
- spanned blocks** 179
- spanned records** 187
 - fixed 180, 186
 - line-end character 183
 - MVS variable 182
 - packed 184

- specifying options on the PIPE command** 19
- specifying ranges in SORT** 75
- specifying ranges in SPECS** 60
- specifying ranges of fields** 63
- specifying ranges of words** 62
- SPECS stage command**
 - description 59
 - with multistream pipelines 101
- SPLIT stage command**
 - description 54
- splitting records** 54
- stage**
 - blocked 112
 - description of 3, 4
 - inserting into pipelines 161, 166
- stage command example, user-written**
 - ADD REXX 148
 - AUTHOR REXX 147, 164
 - BACKUP REXX (shows ADDPIPE subcommand) 157
 - COMBO REXX (shows CALLPIPE subcommand) 174
 - COUNTWDS REXX (example subroutine pipeline) 168
 - FIXED REXX (shows CALLPIPE subcommand) 166
 - HOLD REXX 143
 - LOCDEPT REXX 154
 - LOGIT REXX (shows CALLPIPE subcommand) 169
 - MYFANOUT REXX 149, 151
 - REVIT REXX 141
 - SECPARM REXX (shows ADDPIPE subcommand) 158
 - SEELOG REXX (shows CALLPIPE subcommand) 170, 171, 172
 - TITLE REXX 165
 - TRACER REXX (shows ADDPIPE subcommand) 159
 - TRACING REXX (shows CALLPIPE subcommand) 176
 - UENG REXX (shows CALLPIPE subcommand) 173
 - VALIDATE REXX (shows LOOKUP stage command) 109
- stage command, BatchPipeWorks**
 - See also* user-written stage command
 - < (Read a Data Set) 82
 - > (Rewrite a Data Set) 82
 - >> (Append to a Data Set) 83
 - APPEND 89
 - BLOCK 180, 183
 - BUFFER 77
 - CHANGE 58
 - CHOP 56
 - COMMAND 137
 - COUNT 73, 103
 - DEBLOCK 181, 183, 184

- stage command, BatchPipeWorks** *(continued)*
 - DELAY
 - description 3
 - dispatching of 141
 - DROP 49
 - DUPLICATE 73
 - FANIN 98
 - FANINANY 96
 - FANOUT 93
 - FBLOCK 186
 - FIND 44
 - FRLABEL 45
 - JOIN 54
 - LITERAL 40, 80
 - LOCATE 40
 - LOOKUP 107
 - MERGE 105
 - NFIND 44
 - NLOCATE 43
 - OVERLAY 100
 - PACK 185
 - PAD 56
 - PIPCMD 206
 - PREFACE 91
 - REXX 144
 - RUNPIPE 191
 - SORT 74
 - SPECS 59, 101
 - SPLIT 54
 - STATE 85
 - STEM 87
 - storing sequences of 167
 - STRIP 57
 - summary of 205
 - TAKE 48
 - TOLABEL 45
 - tracing 195
 - TSO 136
 - UNIQUE 46
 - UNPACK 185
 - using multiple streams of 9
 - VAR 89
 - XLATE 44, 51
- stage number** 147
- stage separator** 6
- STAGENUM subcommand** 147
- STAGESEP example exec to display stage separator** 6
- STAGESEP option of the PIPE command** 19
- stall, pipeline** 20
- STATE stage command**
 - description 85
- STATEW stage command**
 - description 85
- STEM stage command**
 - description 87

- stem variables, accessing** 87
- storing sequences of stage commands** 167
- stream**
 - combining 96, 98
 - connecting 12
 - copying 93
 - defining 13
 - description 4
 - input 4
 - multiple 9—19, 93—119
 - names for 98
 - numbers for 97
 - output 4
 - primary 9
 - reconnecting 158
 - secondary 9
 - selecting 149
 - severing 153
 - tertiary 94
 - testing for existence of 152
- STREAMNUM subcommand** 152
- string**
 - changing 58
 - continuing 133
 - searching for 39—48
 - writing to a pipeline 80
- STRIP stage command**
 - description 57
- stripping leading or trailing characters** 57
- SUBCOM stage command**
- subcommand**
 - pipeline
 - ADDDPIPE 154
 - CALLPIPE 153
 - for multistream pipelines 119, 148—163
 - MAXSTREAM 150
 - SELECT 149
 - SEVER 153
 - STREAMNUM 152
 - summary of 207
- subcommand, BatchPipeWorks**
 - ADDDPIPE 154
 - CALLPIPE 153, 165
 - for multistream pipelines 148—163
 - MAXSTREAM 150
 - OUTPUT 143, 144
 - PEEKTO 145, 173
 - READTO 143, 144
 - SELECT 149
 - SEVER 153
 - SHORT 145
 - STAGENUM 147
 - STREAMNUM 152
 - summary of 207
- subroutine pipeline**
 - invoking with CALLPIPE 153, 165

- substituting one string for another 58
- summary of BatchPipeWorks 201
- summary of stage commands 205
- summary of subcommands 207
- suppressing indicator words
- suppressing trace messages 196
- switching streams 149
- SYSTSPRT on DD statement**
 - description 126

T

- TAKE stage command**
 - description 48
- taking snapshots of data** 196
- tallying characters, words, and records** 73
- tallying the number of duplicate records** 76
- TERMINAL stage command**
 - use of 79
- terminal, working with** 79
- tertiary output stream**
 - example 94
- testing stage commands** 176
- TIME example exec for NAME option** 197
- TITLE REXX example user-written stage command** 165
- TOLABEL stage command**
 - description 45
- topology of pipeline, changing** 155
- totaling the number of characters, words, or records** 73
- totaling the number of duplicate records** 76
- TRACE operand**
 - RUNPIPE command 190
- TRACE option**
 - PIPE command 190
- TRACER REXX example (shows ADDPIPE)** 159
- traces, controlling messages generated by** 196
- tracing individual stages** 195
- tracing pipelines**
 - with the TRACE operand of RUNPIPE 189
 - with the TRACE option of PIPE 189
- TRACING REXX example (shows CALLPIPE)** 176
- tracing stage commands** 177
- tracing to a file** 191
- trailing blanks, considerations for** 42
- trailing character, removing** 57
- translating individual characters** 51
- translating records** 51
- troubleshooting pipelines** 189—199
- truncating records** 56
- TSO command**
 - issuing from a pipeline 136
- TSO foreground interface**
 - description 131

TSO stage command
description 136

U

UENG REXX example (shows CALLPIPE) 173
underscore (_) indicating blanks in searches 44
understanding pipelines 20
UNIQUE operand of SORT 76
unique records, discarding 47
UNIQUE stage command
description 46
UNPACK stage command
description 185
unpacking records 184
uppercase characters, translating records to 51
user-written stage command
concepts for 139
description 3, 139
dispatching of 141
environmental considerations for 141
processing arguments in 163
REXX language considerations for 139
setting return codes for 140
testing 176
tracing 177
using 143
with batch jobs 141
writing 139—177
user-written stage command, example
ADD REXX 148
AUTHOR REXX 147, 164
BACKUP REXX (shows ADDPIPE subcommand) 157
COMBO REXX (shows CALLPIPE subcommand) 174
COUNTWDS REXX (example subroutine pipeline) 168
FIXED REXX (shows CALLPIPE subcommand) 166
HOLD REXX 143
LOCDEPT REXX 154
LOGIT REXX (shows CALLPIPE subcommand) 169
MYFANOUT REXX 149, 151
REVIT REXX 141
SECPARM REXX (shows ADDPIPE subcommand) 158
SEELOG REXX (shows CALLPIPE subcommand) 170, 171, 172
TITLE REXX 165
TRACER REXX (shows ADDPIPE subcommand) 159
TRACING REXX (shows CALLPIPE subcommand) 176
UENG REXX (shows CALLPIPE subcommand) 173
VALIDATE REXX (shows LOOKUP stage command) 109

using CALLPIPE with other subcommands 171
using pipelines as part of an exec 133
using several secondary streams 17
using user-written stage commands 143

V

V operand
DEBLOCK stage command 183
VALIDATE REXX example (shows LOOKUP DETAIL) 109
VAR stage command
description 89
variable block spanned records 182
variable blocked records 182
variable in an exec
accessing 87
putting command results into 137
variable spanned records 182
VB operand of DEBLOCK
DEBLOCK stage command 183
VBS operand of BLOCK
BLOCK stage command 183
VBS operand of DEBLOCK
DEBLOCK stage command 183

W

word
changing 58
continuing 133
searching for 39—48
writing to a pipeline 80
WORDS operand
SPECS stage command 62
WORDS operand of COUNT
COUNT stage command 73
words, range of 62
WORDUSE example exec for COUNT 104
working with the terminal 79
working with TSO commands 136
WRITE operand of SPECS
SPECS stage command 70
writing a file 82
writing exec variables 87, 89
writing lines with SPECS WRITE operand 70
writing multiple pipelines 11
writing records to the output stream 144
writing stage commands 139—177
writing to the terminal 79

X

XLATE stage command
description 51
transposing underscore character 44

Communicating Your Comments to IBM

IBM BatchPipes OS/390 V2R1
BatchPipeWorks Users Guide
Publication No. SA22-7457-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a reader's comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
 - FAX: (International Access Code)+1+845+432-9405
- If you prefer to send comments electronically, use one of these network IDs:
 - IBM Mail Exchange: USIB6TC9 at IBMMAIL
 - Internet e-mail: mhvrcfs@us.ibm.com
 - World Wide Web: <http://www.ibm.com/s390/os390/>

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.

Reader's Comments — We'd Like to Hear from You

**IBM BatchPipes OS/390 V2R1
BatchPipeWorks Users Guide
Publication No. SA22-7457-00**

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: _____

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

- | | | | |
|--------------------------|-------------------------------|--------------------------|------------------------|
| <input type="checkbox"/> | As an introduction | <input type="checkbox"/> | As a text (student) |
| <input type="checkbox"/> | As a reference manual | <input type="checkbox"/> | As a text (instructor) |
| <input type="checkbox"/> | For another purpose (explain) | | |

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number:

Comment:

Name

Address

Company or Organization

Phone No.



Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Department 55JA, Mail Station P384
2455 South Road
Poughkeepsie, NY 12601-5400



Fold and Tape

Please do not staple

Fold and Tape



File Number:

Program Number: 5655-D45



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SA22-7457-00

