Customization: Using Pipes

Version 5 Release 1
Customization: Using Pipes

Version 5 Release 1
Tivoli NetView for z/OS Customization: Using Pipes

Copyright Notice

© Copyright IBM Corporation 1997, 2002. All rights reserved. May only be used pursuant to a Tivoli Systems Software License Agreement, an IBM Software License Agreement, or Addendum for Tivoli Products to IBM Customer or License Agreement. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without prior written permission of IBM Corporation. IBM Corporation grants you limited permission to make hardcopy or other reproductions of any machine-readable documentation for your own use, provided that each such reproduction shall carry the IBM Corporation copyright notice. No other rights under copyright are granted without prior written permission of IBM Corporation. The document is not intended for production and is furnished “as is” without warranty of any kind. All warranties on this document are hereby disclaimed, including the warranties of merchantability and fitness for a particular purpose.

U.S. Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corporation.

Trademarks

IBM, the IBM logo, Tivoli, the Tivoli logo, AIX, BookManager, DATABASE 2, DB2, MVS, NetView, Open Edition, OS/390, RACF, SecureWay, Tivoli Enterprise, TME 10, VTAM, and z/OS are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Notices

References in this publication to Tivoli Systems or IBM products, programs, or services do not imply that they will be available in all countries in which Tivoli Systems or IBM operates. Any reference to these products, programs, or services is not intended to imply that only Tivoli Systems or IBM products, programs, or services can be used. Subject to valid intellectual property or other legally protectable right of Tivoli Systems or IBM, any functionally equivalent product, program, or service can be used instead of the referenced product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by Tivoli Systems or IBM, are the responsibility of the user. Tivoli Systems or 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, New York 10504-1785, U.S.A.

Programming Interfaces

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain services of Tivoli NetView for z/OS.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>vii</td>
</tr>
<tr>
<td>Who Should Read This Document</td>
<td>vii</td>
</tr>
<tr>
<td>What This Document Contains</td>
<td>vii</td>
</tr>
<tr>
<td>Publications</td>
<td>vii</td>
</tr>
<tr>
<td>Prerequisite and Related Documents</td>
<td>vii</td>
</tr>
<tr>
<td>Accessing Publications Online</td>
<td>vii</td>
</tr>
<tr>
<td>Ordering Publications</td>
<td>vii</td>
</tr>
<tr>
<td>Providing Feedback about Publications</td>
<td>vii</td>
</tr>
<tr>
<td>Contacting Customer Support</td>
<td>vii</td>
</tr>
<tr>
<td>Accessibility Information</td>
<td>ix</td>
</tr>
<tr>
<td>Keyboard Access</td>
<td>ix</td>
</tr>
<tr>
<td>Conventions Used in This Document</td>
<td>ix</td>
</tr>
<tr>
<td>Platform-specific Information</td>
<td>ix</td>
</tr>
<tr>
<td>Terminology</td>
<td>ix</td>
</tr>
<tr>
<td>Reading Syntax Diagrams</td>
<td>x</td>
</tr>
<tr>
<td>Required Syntax</td>
<td>xi</td>
</tr>
<tr>
<td>Optional Keywords and Variables</td>
<td>xi</td>
</tr>
<tr>
<td>Default Values</td>
<td>xi</td>
</tr>
<tr>
<td>Long Syntax Diagrams</td>
<td>xii</td>
</tr>
<tr>
<td>Syntax Fragments</td>
<td>xii</td>
</tr>
<tr>
<td>Commas and Parentheses</td>
<td>xiii</td>
</tr>
<tr>
<td>Highlighting, Brackets, and Braces</td>
<td>xiv</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>xv</td>
</tr>
</tbody>
</table>

### Chapter 1. NetView Pipelines Introduction and General Concepts

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Is a Pipeline</td>
<td>1</td>
</tr>
<tr>
<td>Pipeline Stages</td>
<td>2</td>
</tr>
<tr>
<td>PIPE Command</td>
<td>3</td>
</tr>
<tr>
<td>Stage Input and Output</td>
<td>4</td>
</tr>
<tr>
<td>First and Subsequent Stages</td>
<td>6</td>
</tr>
<tr>
<td>Complex Pipelines</td>
<td>7</td>
</tr>
<tr>
<td>Creating a Complex Pipeline</td>
<td>7</td>
</tr>
<tr>
<td>Processing a Complex Pipeline</td>
<td>9</td>
</tr>
<tr>
<td>Stage Commands</td>
<td>11</td>
</tr>
<tr>
<td>Device Drivers</td>
<td>11</td>
</tr>
<tr>
<td>Filters</td>
<td>12</td>
</tr>
<tr>
<td>Understanding NetView Pipelines</td>
<td>12</td>
</tr>
<tr>
<td>How a Pipeline Begins</td>
<td>12</td>
</tr>
<tr>
<td>How a Pipeline Ends</td>
<td>13</td>
</tr>
<tr>
<td>Online Help Facility</td>
<td>13</td>
</tr>
<tr>
<td>Getting Started with NetView Pipelines</td>
<td>13</td>
</tr>
</tbody>
</table>

### Chapter 2. Pipeline Stages and Syntax

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIPE (NCCF)</td>
<td>20</td>
</tr>
<tr>
<td>PIPE APPEND</td>
<td>27</td>
</tr>
<tr>
<td>PIPE BETWEEN</td>
<td>29</td>
</tr>
<tr>
<td>PIPE CASEI</td>
<td>31</td>
</tr>
<tr>
<td>PIPE CHANGE</td>
<td>33</td>
</tr>
<tr>
<td>PIPE CHOP</td>
<td>35</td>
</tr>
<tr>
<td>PIPE COLLECT</td>
<td>37</td>
</tr>
<tr>
<td>PIPE CONSOLE</td>
<td>42</td>
</tr>
<tr>
<td>PIPE CORRCDM</td>
<td>45</td>
</tr>
<tr>
<td>PIPE CORRWAIT</td>
<td>49</td>
</tr>
<tr>
<td>PIPE COUNT</td>
<td>54</td>
</tr>
<tr>
<td>PIPE CPDOMAIN</td>
<td>58</td>
</tr>
</tbody>
</table>
Using DSIVSMX with Alternate Index VSAM Files ............................................ 293
  Defining a VSAM File with Alternate Index .................................................. 293
  Accessing VSAM Files Using Alternate Keys ................................................. 294
  Deleting an Alternate Index File ..................................................................... 294
  Using the AUTOTOKE Value Provided by DSIVSMX ........................................ 294
DSIVSAM: Access to Keyed VSAM Files Defined by NetView DSTs .................. 294
  Using the AUTOTOKE Value Provided by DSIVSAM ........................................ 295

Chapter 8. Debugging NetView Pipelines ............................................................... 297
  Understanding Error Messages ........................................................................ 297
  Online Help ...................................................................................................... 297
  Clogged Pipelines ........................................................................................... 298
  Tracing Pipelines ............................................................................................. 299
    Displaying Stage Results ............................................................................. 299
    Displaying Data Stream Information (DEBUG) ............................................ 300
  Displaying Return Codes .............................................................................. 303
  Additional Trouble Shooting Tips .................................................................... 303

Appendix. Additional NetView Pipeline Examples ................................................. 305
  Displaying Part of a Lengthy Automated Message .......................................... 305
  Transferring a Large Variable to Another Task .............................................. 306
  Searching for APARs and PTFs ....................................................................... 308
  Displaying Task Information Summary .......................................................... 309
  Displaying or Canceling a JES2 Job ................................................................. 310

Index .................................................................................................................... 315
Preface

This document describes how to use Tivoli® NetView® for z/OS™ pipelines to customize your NetView installation.

Who Should Read This Document

This document is intended to help system programmers customize the NetView program.

What This Document Contains

This book is organized into the following sections.

- "Chapter 1. NetView Pipelines Introduction and General Concepts” on page 1 describes NetView pipelines and how they can be used.
- "Chapter 2. Pipeline Stages and Syntax” on page 19 describes the stages and syntax for NetView pipelines.
- "Chapter 3. NetView Pipelines Device Drivers” on page 235 describes how to use NetView pipelines device driver stages.
- "Chapter 4. NetView Pipeline Filters” on page 255 describes how to use NetView pipelines filter stages.
- "Chapter 5. Full-Screen Automation” on page 267 describes how to use NetView pipelines for full-screen automation.
- “Chapter 7. REXX Access to VSAM Files” on page 293 describes how to access VSAM files in NetView pipelines.
- "Chapter 8. Debugging NetView Pipelines” on page 297 describes techniques to use to debug NetView pipelines.
- “Appendix. Additional NetView Pipeline Examples” on page 303 describes a few additional uses of NetView pipelines.

Publications

This section lists prerequisite and related documents. It also describes how to access Tivoli publications online, how to order Tivoli publications, and how to make comments on Tivoli publications.

Prerequisite and Related Documents

To read about the new functions offered in this release, refer to the Tivoli NetView for z/OS Installation: Migration Guide.

You can find additional product information on these Internet sites:

<table>
<thead>
<tr>
<th>IBM®</th>
<th><a href="http://www.ibm.com/">http://www.ibm.com/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tivoli Systems</td>
<td><a href="http://www.tivoli.com/">http://www.tivoli.com/</a></td>
</tr>
<tr>
<td>Tivoli NetView for z/OS</td>
<td><a href="http://www.tivoli.com/nv390">http://www.tivoli.com/nv390</a></td>
</tr>
</tbody>
</table>
Preface

The Tivoli NetView for z/OS Web site offers demonstrations of the NetView product, related products, and several free NetView applications you can download. These applications can help you with tasks such as:

- Getting statistics for your automation table and merging the statistics with a listing of the automation table
- Displaying the status of a JES job or cancelling a specified JES job
- Sending alerts to the NetView program using the program-to-program interface (PPI)
- Sending and receiving MVS™ commands using the PPI
- Sending TSO commands and receiving responses

Accessing Publications Online

You can access many Tivoli publications online using the Tivoli Information Center, which is available on the Tivoli Customer Support Web site:

http://www.tivoli.com/support/documents/

These publications are available in PDF format. Translated documents are also available for some products.

Ordering Publications

You can order many Tivoli publications online at the following Web site:

http://www.ibm.com/shop/publications/order

You can also order by telephone by calling one of these numbers:

- In the United States: 800-879-2755
- In Canada: 800-426-4968
- In other countries, for a list of telephone numbers, see the following Web site:
  http://www.tivoli.com/inside/store/lit_order.htm

Providing Feedback about Publications

We are very interested in hearing about your experience with Tivoli products and documentation, and we welcome your suggestions for improvements. If you have comments or suggestions about our products and documentation, contact us in one of the following ways:

- Send an e-mail to pubs@tivoli.com.
- Complete our customer feedback survey at the following Web site:
  http://www.tivoli.com/support/survey/

Contacting Customer Support

If you have a problem with any Tivoli product, you can contact Tivoli Customer Support. See the Tivoli Customer Support Handbook at the following Web site:

http://www.tivoli.com/support/handbook/

The handbook provides information about how to contact Tivoli Customer Support, depending on the severity of your problem, and the following information:

- Registration and eligibility
• Telephone numbers and e-mail addresses, depending on the country you are in
• What information you should gather before contacting support

Note: Additional support for Tivoli NetView for z/OS is available at the NetView for z/OS Web site:

http://www.tivoli.com/nv390

Under Related Documents, select Other Online Sources.

The page displayed contains a list of newsgroups, forums, and bulletin boards.

Accessibility Information

Refer to Tivoli NetView for z/OS User’s Guide for information about accessibility.

Keyboard Access

Standard shortcut and accelerator keys are used by the product and are documented by the operating system. Refer to the documentation provided by your operating system for more information.

Refer to Tivoli NetView for z/OS User’s Guide for more information about keyboard access.

Conventions Used in This Document

The document uses several typeface conventions for special terms and actions. These conventions have the following meaning:

**Bold** Commands, keywords, flags, and other information that you must use literally appear like **this**, in **bold**.

*Italics* Variables and new terms appear like *this*, in *italics*. Words and phrases that are emphasized also appear like *this*, in *italics*.

**Monospace** Code examples, output, and system messages appear like **this**, in a monospace font.

**ALL CAPS** Tivoli NetView for z/OS commands are in ALL CAPITAL letters.

Platform-specific Information

For more information about the hardware and software requirements for NetView components, refer to the Tivoli NetView for z/OS Licensed Program Specification.

Terminology

For a list of Tivoli NetView for z/OS terms and definitions, refer to http://www.networking.ibm.com/NSG/NSGMain.htm

For brevity and readability, the following terms are used in this document:

**NetView**
• Tivoli NetView for z/OS Version 5 Release 1
• Tivoli NetView for OS/390® Version 1 Release 4
• Tivoli NetView for OS/390 Version 1 Release 3
Preface

- TME 10™ NetView for OS/390 Version 1 Release 2
- TME 10 NetView for OS/390 Version 1 Release 1
- IBM NetView for MVS Version 3
- IBM NetView for MVS Version 2 Release 4
- IBM NetView Version 2 Release 3

**MVS**
OS/390 or z/OS operating systems.

**RACF®**
RACF is a component of the SecureWay® Security Server for z/OS and OS/390, providing the functions of authentication and access control for OS/390 and z/OS resources and data, including the ability to control access to DB2 objects using RACF profiles. Refer to:


**Tivoli Enterprise™ software**
Tivoli software that manages large business networks.

**Tivoli environment**
The Tivoli applications, based upon the Tivoli Management Framework, that are installed at a specific customer location and that address network computing management issues across many platforms. In a Tivoli environment, a system administrator can distribute software, manage user configurations, change access privileges, automate operations, monitor resources, and schedule jobs. You may have used TME 10 environment in the past.

**TME 10**
In most product names, TME 10 has been changed to Tivoli.

**V and R**
Specifies the version and release.

**VTAM® and TCP/IP**
VTAM and TCP/IP are included in the IBM Communications Server element of the OS/390 and z/OS operating systems. Refer to


Unless otherwise indicated, references to programs indicate the latest version and release of the programs. If only a version is indicated, the reference is to all releases within that version.

When a reference is made about using a personal computer or workstation, any programmable workstation can be used.

---

**Reading Syntax Diagrams**

Syntax diagrams start with double arrowheads on the left (▷) and move along the main line until they end with two arrowheads facing each other (◁).

As shown in the following table, syntax diagrams use position to indicate the required, optional, and default values for keywords, variables, and operands.

<table>
<thead>
<tr>
<th>Element Position</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>On the command line</td>
<td>Required</td>
</tr>
<tr>
<td>Above the command line</td>
<td>Default</td>
</tr>
</tbody>
</table>

---

x  Tivoli NetView for z/OS Customization: Using Pipes
Table 2. How the Position of Syntax Diagram Elements Is Used (continued)

<table>
<thead>
<tr>
<th>Element Position</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Below the command line</td>
<td>Optional</td>
</tr>
</tbody>
</table>

**Required Syntax**

The command name, required keywords, variables, and operands are always on the main syntax line. Figure 1 specifies that the *resname* variable must be used for the CCPLOADF command.

**CCPLOADF**

```
CCPLOADF resname
```

*Figure 1. Required Syntax Elements*

Keywords and operands are written in uppercase letters. Lowercase letters indicate variables such as values or names that you supply. In *Figure 2*, MEMBER is an operand and *membername* is a variable that defines the name of the data set member for that operand.

**TRANSMSG**

```
TRANSMSG MEMBER=membername
```

*Figure 2. Syntax for Variables*

**Optional Keywords and Variables**

Optional keywords, variables, and operands are below the main syntax line. *Figure 3* specifies that the ID operand can be used for the DISPREG command, but is not required.

**DISPREG**

```
DISPREG ID=resname
```

*Figure 3. Optional Syntax Elements*

**Default Values**

Default values are above the main syntax line. If the default is a keyword, it appears only above the main line. You can specify this keyword or allow it to default.

If an operand has a default value, the operand appears both above and below the main line. A value below the main line indicates that if you choose to specify the operand, you must also specify either the default value or another value shown. If you do not specify an operand, the default value above the main line is used.
Figure 4 shows the default keyword STEP above the main line and the rest of the optional keywords below the main line. It also shows the default values for operands MODNAME=* and OPTION=* above and below the main line.

**RID**

![Diagram of RID syntax]

*Figure 4. Sample of Defaults Syntax*

**Long Syntax Diagrams**

When more than one line is needed for a syntax diagram, the continued lines end with a single arrowhead (▷). The following lines begin with a single arrowhead (▷), as shown in Figure 4.

**Syntax Fragments**

Commands that contain lengthy groups or a section that is used more than once in a command are shown as separate fragments following the main diagram. The fragment name is shown in mixed case. See Figure 5 on page xiii for a syntax with the fragments ReMote and FromTo.
Commas and Parentheses

Required commas and parentheses are included in the syntax diagram. When an operand has more than one value, the values are typically enclosed in parentheses and separated by commas. In Figure 6 on page xiv, the OP operand, for example, contains commas to indicate that you can specify multiple values for the testop variable.

Figure 5. Sample Syntax Diagram with Fragments
If a command requires positional commas to separate keywords and variables, the commas are shown before the keyword or variable, as in Figure 4 on page xii.

For example, to specify the BOSESS command with the \textit{sessid} variable, enter:

\texttt{NCCF BOSESS applid,,sessid}

You do not need to specify the trailing positional commas. Positional and non-positional trailing commas either are ignored or cause the command to be rejected. Restrictions for each command state whether trailing commas cause the command to be rejected.

**Highlighting, Brackets, and Braces**

Syntax diagrams do not rely on highlighting, underscoring, brackets, or braces; variables are shown italicized in hardcopy or in a differentiating color for NetView help and BookManager® online books.

In parameter descriptions, the appearance of syntax elements in a diagram immediately tells you the type of element. See Table 3 for the appearance of syntax elements.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
This element... & Looks like this... \\
\hline
Keyword & CCPLOADF \\
Variable & \textit{resname} \\
Operand & MEMBER=\textit{memberof} \\
Default & \textit{today} or INCL \\
\hline
\end{tabular}
\end{table}

\textit{Figure 6. Sample Syntax Diagram with Commas}

If a command requires positional commas to separate keywords and variables, the commas are shown before the keyword or variable, as in \textit{Figure 4 on page xii}

For example, to specify the BOSESS command with the \textit{sessid} variable, enter:

\texttt{NCCF BOSESS applid,,sessid}

You do not need to specify the trailing positional commas. Positional and non-positional trailing commas either are ignored or cause the command to be rejected. Restrictions for each command state whether trailing commas cause the command to be rejected.
Abbreviations

Command and keyword abbreviations are described in synonym tables after each command description.
Chapter 1. NetView Pipelines Introduction and General Concepts

This chapter introduces NetView pipelines. It also documents general-use programming interface and associated guidance information.

Note: If you are already familiar with pipeline concepts, you may want to go directly to “Chapter 2. Pipeline Stages and Syntax” on page 19.

NetView pipelines help you solve a complex problem by dividing the problem into a set of smaller, simpler steps. Each step or stage handles one part of the overall problem. PIPE stages can:

- Read data from system sources, such as files on DASD or variables in command procedures.
- Filter and refine the data.
- Export (output) the data from the pipeline.

You can connect stages in logical sequence until they collectively cover all steps required to solve your problem.

You determine the function of each stage by coding a stage command as described in “Chapter 2. Pipeline Stages and Syntax” on page 19. A stage command and its related parameters is called a stage specification.

When you have completed a series of stage specifications, you can run them with the PIPE command. The PIPE command identifies the series of stage specifications you want to run and, through command parameters, controls other run characteristics to be described later. A collection of stage specifications and the instructions for connecting them is called a pipeline specification.

What Is a Pipeline

It may help you to understand pipelines if you think of them as a plumbing pipeline. In Table 4, a NetView pipeline is compared to a common plumbing pipeline in a water treatment system:

<table>
<thead>
<tr>
<th>A Plumbing Pipeline</th>
<th>A NetView Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receives water from some source: a reservoir or a well.</td>
<td>Receives data from some source: a keyboard or a disk.</td>
</tr>
<tr>
<td>Passes water through the system.</td>
<td>Passes data through stages.</td>
</tr>
<tr>
<td>Combines different sizes and shapes of pipes to perform complex purification processes.</td>
<td>Combines different stage specifications to perform complex data refinement.</td>
</tr>
<tr>
<td>Delivers purified water: to taps or showers.</td>
<td>Delivers refined data: to other programs or storage.</td>
</tr>
</tbody>
</table>

Keep that metaphor in mind as you read, and as you view succeeding graphic illustrations in this chapter, imagine data flowing from left to right in each diagram.
Pipeline Stages

Imagine a stage as a small black box inserted into the plumbing pipeline described in Table 4 on page 3. Also, imagine a series of such boxes, all connected serially, one after the other, throughout the length of the pipeline. Furthermore, imagine that each box performs one specific task on the water passing through it: adjust temperature, remove salt, or add chlorine. Even though each box does only one thing, the cumulative result is salt-free, temperature-controlled, chlorinated water.

Something similar happens in a NetView pipeline: data passes through a stage wherein a stage command performs some action on it. In Figure 7 you can see several stages linked together to form a pipeline that takes data from a disk, processes it, and displays it on an operator console.

![Figure 7. Stages within a Pipeline](image)

Data in the pipeline is viewed as a series of discrete records called messages. They are so called because, once read into the pipeline, each record becomes a message consisting of message text and message attributes.

![Figure 8. Messages Flowing through a Stage](image)

In this example, practically anything can happen to the messages: they can be modified, discarded, split apart, joined together, and so on. Precisely what happens depends on the stage command that is being used. Many stage commands generate one output message for each input message; some commands do not.
this example, three messages went in, but only two came out. Without knowing exactly what stage command was in effect, we cannot say for sure what happened to the third message, but we do know that such disappearances can be legitimate.

Figure 9 shows a more explicit example. In this case, the stage specification is:

LOCATE /BOB/

LOCATE is a stage command and its purpose here is to locate every occurrence of the string BOB in the data passing through the stage. Here, we see three messages flow into the stage. LOCATE looks at the content of each incoming message. If the incoming message contains the string BOB, the message remains in the pipeline. Otherwise, the message is removed from the pipeline.

PIPE Command

You can issue the NetView PIPE command anywhere you would use any NetView regular (Type=R) command:

- The NetView command line
- A NetView command list
- A REXX command list
- A high-level-language command procedure such as PL/I or C
- An environment that allows timer commands.

In a PIPE command, stages are separated by a character called a stage separator.

PIPE STAGE1 | STAGE2 | ... | STAGEn

A stage separator placed before the first stage or after the last stage is optional.

The default stage separator is the character X’4F’. Depending on your workstation, this stage separator is either a solid vertical bar (|) or a split vertical bar (¦).

PIPE commands can be shown in two ways: the portrait format or the landscape format. In portrait format, parameters are stacked vertically in the manner shown in Figure 10 on page 4. In landscape format, parameters are strung horizontally as shown in Figure 11 on page 4. When entering a PIPE command from the command line, you may prefer landscape form. When issuing a PIPE command from a command procedure, you may prefer the portrait form. Either form, or any combination of the two, is valid.
Introduction and Concepts

Note: In portrait form you must include the appropriate continuation character for the programming language after each line except the last.

For readability, most examples in this book are shown in portrait form.

```
PIPE STAGE1
  STAGE2
  STAGE3
  STAGE4
  STAGE5
  STAGE6
```

*Figure 10. A PIPE Command Coded in Portrait Format*

```
PIPE STAGE1 | STAGE2 | STAGE3 | STAGE4 | STAGE5 | STAGE6
```

*Figure 11. A PIPE Command Coded in Landscape Format*

For more information, see "Chapter 2. Pipeline Stages and Syntax" on page 13.

Stage Input and Output

An important concept in the processing of pipelines is the passing of data, or messages, from one stage to another by data streams or streams. A data stream is a logical link between one stage and another that provides for the transfer of messages.

The messages entering a stage are passed on its input stream. The messages leaving a stage are passed on its output stream. In the example in *Figure 9 on page 3*, LOCATE reads all messages from its input stream, but writes only the messages containing the string BOB to its output stream.

*Figure 12* shows how messages flow through several stages. The output of the LOCATE stage becomes the input to the TAKE stage.

```
| Bob Smith |
| Fred Ford |
| Mary Bobbit |

Locate /Bob/ Stage Command

Bob Smith
Mary Bobbit

Take 1 Stage Command

Bob Smith

Bit Bucket

Bit Bucket
```

*Figure 12. Messages Flowing through Multiple Stages*

The LOCATE stage reads three messages from its input stream: BOB SMITH, FRED FORD, and MARY BOBBIT. It writes the messages containing BOB to its output stream. The TAKE stage reads messages from its input stream. There are only two messages: BOB SMITH and MARY BOBBIT. TAKE selects the first message and writes a single message to its output stream.
A stage may have up to ten input and output streams, numbered from 1 to 10. The first two streams are called the primary stream and the secondary stream. Some stages have a third stream, which is called the tertiary stream. Streams 4 through 10 are referred to only by their stream numbers.

There are two additional data stream terms to understand:

**Defined**

A data stream is defined when the pipeline specification calls for data to flow between two stages. For example, the following pipeline displays HI THERE on the console:

```
PIPE LITERAL /HI THERE/
    |CONSOLE
```

In this case, the CONSOLE stage has one input and no output stream defined. The input is from the LITERAL /HI THERE/ stage. In stages that do not allow multiple input and output streams, the position of the stage within the pipeline specification defines how the data will flow to and from the stage.

The primary input and output streams are usually defined by the position of the stage specification within the pipe specification. The primary input, if required, is usually from the previous stage within the pipe specification. The primary output is usually to the following stage within the pipe specification, unless the stage is the last stage within the pipeline specification. In the latter case, the primary output differs depending on the individual stage.

Streams, other than the primary input and primary output, are defined using labels. For information on labels and complex pipelines, see "Complex Pipelines" on page 7.

**Connected**

Data streams are connected and disconnected during the processing of the pipeline. A data stream is connected until the stages connected to the data stream disconnect. A stage will disconnect when a condition, specific to the stage, is encountered. Most stages retain their connections until they terminate. When a stage disconnects its output stream, the corresponding input stream will disconnect as soon as any messages being passed through the stream have been read in, or consumed, by the partner stage.

For example, in Figure 13, Stage A and Stage B are connected by a data stream. The output stream of Stage A is the input stream of stage B. Stage A completes processing and disconnects. Only after Stage B completes reading any messages sent from Stage A does the data stream itself disconnect.

![Figure 13. Input and Output Data Streams](image)

**Note:** Defined is the state of the stream as coded in the pipeline specification and connected is the status of the stream during pipeline processing.
For information on the number of supported streams and the termination conditions of each stage, see the description of each stage in Chapter 2, Pipeline Stages and Syntax.

First and Subsequent Stages

Another important pipeline concept is that of first and subsequent stages.

A stage that generates an output data stream without requiring an input data stream is called a first stage. First stages are used to start the process.

Attention: First stages may occur at the beginning of a pipeline specification or anywhere in a complex pipeline where an input stream is not defined. Examples of first stages are:
- `< (From Disk)
- ENVDATA
- HELDMSG
- QSAM

A stage that accepts input from a stage located before it within a pipeline specification is called a subsequent stage. Examples of stages that can only be subsequent stages are:
- CHANGE
- CONSOLE
- LOCATE
- STRIP

Some stages can be used as a first stage or a subsequent stage. Examples of these are:
- LITERAL
- VAR

Some stages can be used as a first stage or a subsequent stage. However, they have a different syntax for first stage and subsequent stage forms. Examples of these are:
- NETVIEW
- VET

Figure 14 on page 7 shows an example of a pipe stage that can be used as a first stage and as a subsequent stage. In the first stage example, the string HI THERE! is written to the console. In the subsequent stage example, the NETVIEW stage runs the HELP command for PIPE LITERAL. The NetView online help information for the PIPE LITERAL command is displayed on the console after writing the string THE FOLLOWING IS HOW TO USE THE LITERAL PIPE STAGE.
LITERAL as a First Stage

PIPE LITERAL /HI THERE!/ 
| CONSOLE

LITERAL as a Subsequent Stage

PIPE NETVIEW (NOPANEL) HELP PIPE LITERAL 
| LITERAL /THE FOLLOWING IS HOW TO USE THE LITERAL PIPE STAGE/ 
| CONSOLE

Figure 14. Examples of First and Subsequent Stages

“Chapter 2. Pipeline Stages and Syntax” on page 19 describes each NetView pipeline stage.

Complex Pipelines

Some stages accept multiple input streams and others generate multiple output streams.

An example of a stage that generates multiple output streams is the LOCATE stage as shown in Figure 9 on page 3 and Figure 12 on page 4.

LOCATE is used to select specific data from the input stream. In both examples, only the input data, including the string Bob, flow through the stage. But, what if you wanted a way to act on both the selected data and the data that was not selected? You really want a pipeline that looks something like that shown in Figure 15.

Figure 15. Complex Pipeline

The pipeline shown in Figure 15 is called a complex pipeline. A complex pipeline is made up of simple pipelines connected with labels, such as the one shown in Figure 12 on page 4. Complex pipelines are simple programs rather than complicated commands.

Creating a Complex Pipeline

This section describes the way to create a complex pipeline using stages with multiple inputs and outputs. When stages are adjacent to each other in a pipeline, the output stream of a stage is connected to the input stream of the stage that follows.
Introduction and Concepts

Use a label to connect the streams of stages that are not adjacent. A label is 1–8 alphanumeric characters followed by a colon. For example, the B in the following is a label:

...|B: LOCATE /X/|...

To use multiple streams, first include the label on the stage that results in multiple output streams. This first label defines or declares the label. Then, place a matching label in the pipeline specification as if it were a stage. The stages following the label will act on the data passed as the primary output of the stage defining the label. The label acts as a connector and provides the input stream to the subsequent pipeline stages, for example:

| PIPE (END %)
| < NAMES
|  | A: LOCATE /BOB/
|  | CHANGE //HERE IS A NAME CONTAINING 'BOB' ==>/
|  | CONSOLE
|  | %A:
|  | CONSOLE

The < (From Disk) stage reads data from a file called NAMES, containing three names. The three names are:

- BOB SMITH
- FRED FORD
- MARY BOBBIT

The selected data is written to the console using the CONSOLE stage. All records containing the string BOB will be prefixed with the string HERE IS A NAME CONTAINING 'BOB' ==>.

In this example, the string BOB is located in the input data. Label A: was defined on the LOCATE stage; data that does not contain the string BOB will be passed as an input stream to the stage following the stage labeled A:.

This complex pipeline is logically made up of the following parts:

- The definition of the end character that is to be used to separate the different simple pipelines.
 
  PIPE (END %)

- The first simple pipeline. The LOCATE stage, which generates multiple output streams, is labeled with an A: indicating that data not selected by the LOCATE stage will be passed to the connector A: later in the pipeline specification.

  < NAMES
  | A: LOCATE /BOB/
  | CHANGE //HERE IS A NAME CONTAINING 'BOB' ==>/
  | CONSOLE

- The end character indicates the end of the first simple pipeline and the beginning of the second simple pipeline.

  %

- The next occurrence of label A: is as a connector that connects the secondary output of LOCATE /BOB/ as an input stream to CONSOLE in the second simple pipeline. This A: is a connector and not a label definition because this A: is not included in a stage.

  A:
The second simple pipeline that will handle the data not selected by LOCATE /BOB/.

| CONSOLE |

The resulting output of this complex pipeline is shown in Figure 16.

Figure 16. Complex Pipeline Example Output

Notes:
1. You can have as many simple pipelines within a complex pipeline specification as you need. Each stage with multiple outputs may pass data to different connectors. Or, multiple stages may pass data to a single connector.
2. Each pipeline stream acts as independently as possible. For example, in Figure 16, the record FRED FORD is processed in the second simple pipeline, A:, while the first simple pipeline is still processing the records selected by LOCATE /BOB/.
3. If a connector immediately follows an end character, then it defines an output stream to the stage where the label is defined. If an end character, or the end of the pipeline specification, immediately follows a connector, the connector defines input to the stage where the label was defined. Otherwise, the connector defines both an input and output stream to the stage where the label was defined.

Processing a Complex Pipeline

During processing, labels must be defined on a stage before being used as a connector. The label on a stage is the definition or declaration of the label. When the label is later used by itself it is known as a connector.

A label is used to create multiple data streams for a stage. Data streams are numbered, starting with 1, and can go as high as 10 depending on the stage. When a stage is processed, the number 1, or primary, input stream is connected to the previous stage, if any, and the number 1, or primary, output stream is connected to the following stage, if any.

An end character placed before or after a stage prevents connection to the adjacent stage on the side the end character is located. For example, if the end character for the following pipeline fragment was defined with the % character, a connection does not occur between CONSOLE stage and the STEM stage. In this example, STEM acts as a first stage:

```
... 
| CONSOLE % 
| STEM VARN. 
... 
```
When a connector is encountered later in the pipeline specification, a data stream is defined and then connected from the stage where the label was defined to the connector. The lowest stream number available is assigned to the data stream.

If the labeled stage has an output in a simple pipeline within a complex pipeline, the data stream will be an output from the stage defining the label and an input to the stage following the connector. If the labeled stage is an output to a stage in the pipeline specification, the data stream will be an input to the stage defining the label and an output to the stage preceding the connector.

It is possible for a connector to be neither first nor last, in which case, the connector defines both an input and an output for the labeled stage. It is also possible to use two connectors in a row. This usage connects the output of one labeled stage to the input of another.

In the following example, the secondary output of LOCATE is connected to the secondary input of FANIN:

```
PIPE (END ¬)
| < SOMEMEM
| COLOR YELLOW
| RD: LOCATE /GREEN/
| COLOR GREEN
| BK: FANIN
| CONSOLE ONLY
| ¬ RD:
| BK:
```

The PIPE stages in this example are explained in detail in [Chapter 2. Pipeline Stages and Syntax] on page 19. For now, understand that < reads data from a data set member called SOMEMEM, the COLOR stage changes the color of text presented on the CONSOLE, and FANIN collects data from multiple input streams and passes the data to a single output stream.

In this pipeline, all the records in the member SOMEMEM are read and given the color attribute YELLOW. Then, all records containing the word GREEN are colored green. Records containing the word GREEN flow through the pipeline directly to the FANIN stage and then to the console. Records that do not contain the word GREEN flow to the RD: connector from the LOCATE/GREEN/ stage, which defines the RD: label. Because the BK: connector follows the RD: label, the data flows from the BK: connector as input to the stage defining it (BK: FANIN).

**Stages that Disconnect Streams before Termination**

Some stages can disconnect a stream before terminating. An example is the TAKE stage. The TAKE stage disconnects its primary output as soon as the specified count is reached. However, if a secondary output stream is defined, the TAKE stage is not terminated. It continues to pass messages to its secondary output stream.

The processing of the TAKE stage is important to the following REXX pipeline, because FANIN will not begin to read its secondary input until its primary input has disconnected:

```
'PIPE (NAME REDNAME END ¬)',
  ' < NAMELIST',     /* Read list of names */
  ' C: TAKE 12',     /* Hope to get 12 or fewer names */
  ' R: FANIN',       /* Bringing names together */
  ' $STEM NAMEVAR.', /* Save names in order FANIN reads */
  ' CONSOLE',        /* and display them */
```
The first 12 names are displayed on the console in the default color. The remaining names are displayed in red.

**Stage Commands**

Depending on their function, stage commands can be grouped into two categories: device drivers and filters.

Device drivers are stage commands that interact with devices or other system resources. They are used to get data into or out of the pipeline.

Filters work on data already in the pipeline.

**Device Drivers**

When we speak of device drivers, we define a device loosely as a disk file, a terminal, a command procedure variable, or the system environment. Although not all of these are true devices, they all are entities with which a device driver interacts to read or write data.

Device drivers do not act on data; they merely transport it. In general, device drivers write their input stream to their output stream.

The simplest pipeline consists of two device drivers. Data read from one device moves through the pipeline to the other device, as shown in Figure 17.

```
PIPE < TESTDATA | CONSOLE
```

This PIPE command performs the functions shown in Figure 17.

The < (From Disk) stage reads data from DASD into the pipeline where each record becomes a message, receiving the attributes of a message. Then the < (From
Disk) stage writes each message to its output stream. In Figure 17 on page 11, the output of the < stage is the input of the CONSOLE stage. The CONSOLE stage reads the messages from its input stream, displays them on the screen and copies them to its output stream, if one exists.

Filters

Device drivers get data into and out of a pipeline; filters, also known as selection stages, work on data (that is, messages) already in the pipeline. Therefore, a filter must be used with at least two device drivers: one to provide the input stream to the filter and one to receive the output stream from the filter.

The LOCATE stage is a filter. LOCATE examines the messages from its input stream, and searches for those containing a specified string. The messages that match are written to the output stream; those that do not match are discarded or passed to a secondary output stream, if one is connected.

Filters perform many functions of general use. For example, they select messages based on the content of the message or on the position of the message in the stream flowing through the pipeline.

Understanding NetView Pipelines

When you issue the PIPE command, NetView pipelines check the spelling and syntax of all the stage specifications. If spelling and syntax are correct, pipeline processing begins. Otherwise, the pipeline is stopped and a nonzero return code is generated.

How a Pipeline Begins

After processing begins, the PIPE command decides which stage to run and when to run it. It is not a matter of turning on all stages at once or of turning on one stage and running it to completion before starting the next stage. For the most part, the processing resembles the plumbing pipeline described earlier. That is:

- Data begins flowing from a source through a device driver. At this point, no subsequent stages are active.
- The device driver passes the data to the next stage, a filter, perhaps. The driver then gets more data. At this point only the driver and the filter are active.
- Data flows from stage to stage, activating each stage as it goes.
- Soon, the entire pipeline is active with a flow of data just as a plumbing pipeline is active with a flow of water.
- Ultimately, data begins to leave the pipeline through a device driver and the source of data will be exhausted.
- As the last bits of data flow through the pipeline, stages disconnect from their input and output streams as they become inactive.
- After all the stages have disconnected, the pipeline ends.

By operating in this fashion, a pipeline can process an extremely large volume of data without having to keep the entire volume in storage. However, some stage commands need to read all the data before they can begin processing messages. For example, the COLLECT command must collect all the messages from the input stream before writing the messages as one multiline write-to-operator message (MLWTO) to its output stream.
How a Pipeline Ends

Each stage uses its own rules to determine when (and whether) to disconnect. For many stages, a disconnect from one side causes the stage to disconnect from the other side. Some stages (TOSTRING, for example) examine the message stream to determine when to disconnect the output stream.

Usually, a pipeline continues to process as long as any stages are connected.

A pipeline ends when all of its stages end. A stage ends when one of the following events occurs:

- The stage completes its function.
- The stage detects an unrecoverable error.
- The stage detects that its termination conditions have been reached. See the stage descriptions in "Chapter 2. Pipeline Stages and Syntax" for more information.
- The stage detects that there is no more data to read from a device (for device drivers only).
- The pipeline becomes clogged. A deadlock occurs between the data streams within a complex pipeline.

Online Help Facility

You can obtain information about the PIPE command and stage commands with the NetView online help facility. To display online help for the pipe command, enter:

HELP PIPE SYNTAX

To display online help for a specific stage command, enter:

HELP PIPE stage_command_name

Where: stage_command_name is any NetView PIPE stage command.

Getting Started with NetView Pipelines

The PIPE command specification consists primarily of options and stage specifications with a stage separator between each stage. The default stage separator character is usually a vertical bar (|) on 3270 terminals, but may be a split vertical bar (¦) on workstation terminals.

The following examples use several pipeline specifications to manipulate messages in different ways. They are intended to show basic pipeline possibilities without exploring all the filters and device drivers available. For more information on other filters, see "Chapter 4. NetView Pipeline Filters". For information on device drivers, see "Chapter 3. NetView Pipelines Device Drivers".

As an example, consider two fictitious people and a fictitious event: Pete and Sam planning their annual vacation. They have created a member named WISHLIST in a partitioned data set that is associated with the DSIPARM ddname. WISHLIST contains travel information, including sites to see and various attractions. Pete and Sam are working in an MVS environment.

Pete decides to write a PIPE command that will list all the destinations on their list.
He enters on the command line:

```
PIPE < WISHLIST | CONSOLE
```

The < (From Disk) stage accesses a disk file and writes its contents to the pipeline, thus bringing data into the pipeline. The complete stage specification for the < (From Disk) stage is `< WISHLIST`, which consists of the stage name, `<`, and its operand, WISHLIST.

The CONSOLE stage displays the results to the operator console. The complete CONSOLE stage specification is `CONSOLE`, because none of its operands are used in this example. The < (From Disk) and CONSOLE stages are both device drivers.

The output to Pete’s operator console looks like this:

```
+ CNM19 PIPE < WISHLIST | CONSOLE
| CNM19 CAIRO, EGYPT          AFRICA
| CNM19 CASABLANCA, MOROCCO   AFRICA
| CNM19 KRUGER PARK           AFRICA
| CNM19 NILE RIVER            AFRICA
| CNM19 BANGKOK, THAILAND     ASIA
| CNM19 GREAT WALL, CHINA     ASIA
| CNM19 TOKYO, JAPAN          ASIA
| CNM19 YANGTZE RIVER         ASIA
| CNM19 ALICE SPRINGS         AUSTRALIA
| CNM19 PARIS, FRANCE         EUROPE
| CNM19 RHINE RIVER           EUROPE
| CNM19 ROME, ITALY           EUROPE
| CNM19 GRAND CANYON PARK, ARIZONA, USA N. AMERICA
| CNM19 MISSISSIPPI RIVER     N. AMERICA
| CNM19 NEW YORK, NEW YORK, USA N. AMERICA
| CNM19 YOSEMITE PARK, CALIFORNIA, USA N. AMERICA
| CNM19 DANALI PARK, ALASKA, USA N. AMERICA
| CNM19 AMAZON RIVER          S. AMERICA
| CNM19 ANDES MOUNTAINS       S. AMERICA
| CNM19 RIO DE JANEIRO, BRAZIL S. AMERICA
| ???
```

Pete is an avid canoeist, so he decides to change the pipeline specification to limit his selection of vacation spots to those with rivers. He accesses the same disk file shown in the previous example, but he now enters this PIPE command:

```
PIPE < WISHLIST
   LOCATE /RIVER/
   LITERAL /Pete's canoeing adventures/
   CONSOLE
```

Actually, if Pete were to enter that command on a command line, he would have been limited to three lines and would have entered it in landscape form, but it is shown here in portrait form for ease of reading.

Pete’s command uses the LOCATE stage to select all messages that contain the character string RIVER. Messages not selected are discarded from the pipeline and are no longer available for subsequent processing by other stages. To indicate that the choices are his, Pete uses the LITERAL stage to add a comment to the pipeline. The LITERAL stage writes text to the pipeline ahead of messages already there. As before, the results are displayed to the operator console, using the CONSOLE stage.

The specification for the LOCATE stage is `LOCATE /RIVER/`. The operand RIVER is supplied as a search argument for the stage to use when examining messages. The slash (/) character is the string delimiter.
The specification for the LITERAL stage is:
LITERAL /Pete's canoeing adventures/

The operand (Pete's canoeing adventures) represents the text to be placed in the pipeline.

In this example, the <, LITERAL, and CONSOLE stages are device drivers, whereas the LOCATE stage is a filter.

The results are shown as they would appear on the operator console. Because NETVASIS was not specified, the lowercase literal value becomes uppercase when it is displayed.

Sam, a hiking enthusiast, changes the pipeline specification to display which destinations are parks. He plans to budget his trip carefully and decides to eliminate destinations far from his home, such as Asia, Africa, and Australia.

Sam enters this command:
PIPE < WISHLIST
    LOCATE /PARK/
    NLOCATE /AFRICA/ /ASIA/ /AUSTRALIA/
    LITERAL /Sam's hiking choices/
    CONSOLE

Sam changes the LOCATE stage to select all messages that contain the character string PARK. He adds the NLOCATE filter stage to his pipeline to discard messages for the continents he is not interested in visiting. All messages not discarded by NLOCATE remain in the pipeline for further processing.

The stage specification for the NLOCATE stage is NLOCATE /AFRICA/ /ASIA/ /AUSTRALIA/. The three character strings, AFRICA, ASIA, and AUSTRALIA are the search arguments.

Note: The LITERAL stage appears after the LOCATE and NLOCATE stages because:
• It is undesirable for the text string Sam's hiking choices to be subjected to filtering done by the LOCATE and NLOCATE stages.
• LITERAL writes the text string in front of messages already present in the pipeline.

In this example, the <, LITERAL, and CONSOLE stages are device drivers, and the LOCATE and NLOCATE stages are filters.

The results are shown as they would appear on Sam's console:
After much discussion, the young men decide to combine choices and add one other filter (TAKE FIRST 1) as well. Pete enters this command:

```
PIPE < WISHLIST
  LOCATE /RIVER/ /PARK/
  NLOCATE /AFRICA/ /ASIA/ /AUSTRALIA/
  TAKE FIRST 1
  LITERAL /OUR FINAL VACATION CHOICE/
  COLLECT
  CONSOLE
```

They changed the LOCATE stage to select messages that contain either the word RIVER or PARK. The NLOCATE stage subsequently sees only messages containing RIVER or PARK and from those messages discards all that contain ASIA, AFRICA, or AUSTRALIA.

Next, the TAKE stage selects the first message remaining in the pipeline after all the previous stages are completed. This is a message showing a river or park that is not in Asia, Africa, or Australia.

The complete specification for the TAKE stage is TAKE FIRST 1, which indicates that the first message should be selected from the input stream and all other messages should be discarded from the pipeline.

Pete changes the text string used by the LITERAL stage and also adds the COLLECT stage to gather all messages in the pipeline into one multiline message before displaying them.

In this example, the <, LITERAL, and CONSOLE stages are device drivers, and the LOCATE, NLOCATE, COLLECT, and TAKE stages are filters.

The PIPE command generates this display on Pete’s console:

```
NCCF NETVIEW CNM19 PETE 03/26/99 13:50:00
  PIPE < WISHLIST | LOCATE /RIVER/ | NLOCATE /AFRICA/ /ASIA/ /AUSTRALIA/ | TAKE FIRST 1 | LITERAL /OUR FINAL VACATION CHOICE/ | COLLECT | CONSOLE
```

As a final step, Sam writes and runs a small command procedure, WISHCLST, written in the NetView command list language. This command procedure uses PIPE commands to write the final vacation choice to a command procedure variable, then read the variable, and display the results to a console.

The command procedure is in Figure 18 on page 17. The output from the procedure in Figure 18 on page 17 is shown in Figure 19 on page 17.
WISHCLST CLIST
&CONTROL ERR
*
******************************************************************************
** THIS CLIST USES THREE PIPE COMMANDS.  **
** - THE FIRST WRITES A MESSAGE TO A CLEARED SCREEN.  **
** - THE SECOND WRITES A MESSAGE TO A CLIST VARIABLE.  **
** - THE THIRD READS THE VARIABLE AND DISPLAYS THE RESULTS.  **
******************************************************************************
*
******************************************************************************
** WRITE MESSAGE TO TERMINAL USING PIPE COMMAND **
******************************************************************************
Pipe LITERAL /WISHCLST IS PROCESSING/ +
   | CONSOLE CLEAR
******************************************************************************
** CHOOSE A VACATION DESTINATION USING THE PIPE COMMAND TO **
** READ RECORDS FROM A DISK FILE INTO THE PIPELINE, **
** MANIPULATE THEM AND STORE ONE RESULTING MESSAGE IN THE **
** VARIABLE NAMED VACVAR. **
******************************************************************************
Pipe < WISHLIST +
   | LOCATE /RIVER/ /PARK/ +
   | NLOCATE /AFRICA/ /ASIA/ /AUSTRALIA/ +
   | TAKE FIRST 1 +
   | VAR VACVAR
******************************************************************************
** READ VARIABLE NAMED VACVAR INTO THE PIPELINE AND ADD **
** 'VACATION CHOICE' TEXT AHEAD OF IT, THEN DISPLAY. **
******************************************************************************
Pipe VAR VACVAR +
   | LITERAL /VACATION CHOICE/ +
   | COLLECT +
   | CONSOLE
******************************************************************************
** WRITE RETURN CODE INFORMATION TO TERMINAL AND EXIT CLIST **
******************************************************************************
&WRITE RETURN CODE = &RETCODE
&EXIT
******************************************************************************

Figure 18. A Pipeline Invoked from a Command Procedure Called WISHCLST

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW</th>
<th>CNM19 OPER6</th>
<th>03/26/99 14:40:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNM19</td>
<td>WISHCLST IS PROCESSING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNM19</td>
<td>VACATION CHOICE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RHINE RIVER</td>
<td>EUROPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C CNM19</td>
<td>RETURN CODE = 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 19. Pipeline Output from WISHCLST Command Procedure. Created by the pipeline in Figure 18 on page 17
Chapter 2. Pipeline Stages and Syntax

This chapter documents general-use programming interface and associated guidance information.

This chapter also describes the syntax, keywords, and parameters of the PIPE command and shows examples of the PIPE command and its stages.
Pipeline Stages and Syntax

PIPE (NCCF)

Syntax

PIPEREF

PIPE (Pipe Options) label: (DEBUG) Stages

Stages

stage_specification

Pipe Options:

STAGESEP | STAGESEP value
ESC value | END value
NAME PIPE
NAME pipename

LOWQENAB | DEBUG 1 | DEBUG 2

Command Description

NetView Pipelines help you solve a complex problem by dividing it into a series of smaller steps. Each step or stage solves one part of the overall problem. Some stages read data from system sources, such as files on DASD or variables in command procedures. Other stages filter and refine that data in some way. Still other stages export (output) data from the pipeline. You can connect stages in logical sequence until they collectively cover all steps required to solve your problem.

You determine the function of each stage by coding a stage command as described in this chapter.

When you have completed a series of stage specifications, you can run them with the PIPE command. The PIPE command identifies the series of stage specifications you want to run and, through command parameters, controls other run characteristics, which will be described later. A series of stage specifications and the instructions for connecting them is called a pipeline specification. A PIPE command containing multiple pipeline specifications, labels, and end characters is called a complex pipeline. A simple pipeline contains a pipeline specification, but does not contain labels or end characters.

You can obtain information about the PIPE command and stage commands using the NetView online help facility. To display online help for the PIPE command, enter:

HELP PIPE SYNTAX
To display online help for a specific stage command, enter:

```
HELP PIPE stage_command_name
```

**Where:**

*stage_command_name*

Is any NetView PIPE stage command.

See [“PIPE Stages” on page 24](#) for an alphabetical listing and brief summary of each PIPE stage.

### Operand Descriptions

#### DEBUG

Generates connection and data stream trace information that can be used to debug pipelines. DEBUG, when used as a PipeOption, must have one of the following specified:

1. Produce debug output for all pipeline stages. This is the same as coding (DEBUG) on each stage.

2. Produce additional debug information whenever a BNH155E message is generated. BNH155E indicates that the pipeline is clogged. The additional information produced by DEBUG 2 will help you diagnose the clog.

Both DEBUG 1 and DEBUG 2 can be specified in the same pipeline specification.

For more information on the DEBUG option see [Chapter 8. Debugging NetView Pipelines” on page 297](#).

#### END

The end character allows multiple, simple pipelines to operate within a complex pipeline. The pipeline specification, included after the end character, operates independently of the pipeline specified before the end character. The end character, with stage labels, and stages with multiple input or output streams is used to create complex pipelines. For information on creating complex pipelines, see [“Complex Pipelines” on page 7](#).

The valid value of END can be a character acceptable for STAGESEP, but the value cannot be the same value as STAGESEP or ESC in the same PIPE command.

If you want to include the end character within your pipe where it must not be interpreted as an end character, you can either include the ESC character immediately before it or use the “self escape” technique. Two side-by-side END characters will resolve to one character taken literally. For example, if your ESC character is defined as % and your END character is defined as ?, use either of the following:

```
PIPE (END ?) LITERAL 'MY END CHARACTER IS ?'
```

```
PIPE (ESC % END ?) LITERAL 'MY END CHARACTER IS ?'
```

The following is displayed on the console:

```
MY END CHARACTER IS ?
```

#### ESC

Indicates that the character following the specified character is treated literally.
Pipeline Stages and Syntax

when the pipeline specification is parsed. For example, if you specify STAGESEP | ESC % as options and the string ABC%|XYZ is encountered in the pipeline specification, then the % character is removed and the following | character is not treated as a stage separator. This leaves the string ABC|XYZ in your stage specification.

The valid value of ESC is that of any character acceptable for STAGESEP, but it cannot be the same value as that used for STAGESEP or END in the same PIPE command.

Alternatively, you can use the stage separator character to “self escape” itself. Two side-by-side separators resolve to one such character taken literally. For example:

PIPE LITERAL 'MY CHAR IS ||' | CONSOLE

Will result in the display of the following:

MY CHAR IS |

label

The label must be 1–8 alphanumeric characters followed by a colon. A label can be followed by blanks. Although you can assign a label to any stage, label is only useful when used with the end character and stages with multiple output streams to create complex pipelines. For information on creating complex pipelines, see "Complex Pipelines" on page 3.

LOWQENAB

Commands that are queued at a low priority are not processed until pipeline processing is complete. Commands blocked include low-priority commands from the automation table. Blocking low-priority commands assures that the commands are processed first in-first out (FIFO).

High-priority commands pre-empt pipeline processing.

When LOWQENAB is specified, pipeline processing is temporarily suspended whenever any command is queued. Even low-priority commands will preempt pipeline processing.

LOWQENAB effects only the pipeline where it is specified. All other automation and low priority commands continue to run in FIFO order.

NAME

Indicates the name of this pipeline. The name given is used in various messages about the processing of the pipeline and can be an aid in debugging.

pipename

The value should be 1–8-alphanumeric characters. The default value is PIPE.

stage_specification

A NetView stage command and its operands. This may be a label previously defined in the pipeline that is used as connector in a complex pipeline. At least one stage command or connector label must be specified.

STAGESEP

Specifies the character used to separate the stages in a PIPE command.

The STAGESEP, END, and ESC characters must be different within a single pipe specification.

value

Is a single-byte, nonalphanumeric EBCDIC character except blank, null, ), (, @, #, and $.
The default stage separator character is X'4F'. On American and English 3270 displays, X'4F' is represented as a vertical bar (|). In other countries or on some workstations, X'4F' can be displayed as an exclamation mark or a split vertical bar.

**Usage Notes**

- For the PIPE command and its stages, a delimiter can be any character except alphanumeric characters, parentheses, blanks, nulls, and national characters (@, #, and $). The maximum length for a delimited string is 255 characters. Multiple delimited strings must be separated by blanks. In the formats shown, the delimiter is a /.

  Ensure delimited strings do not contain:
  - The delimiter character in use
  - The stage separator (unless escaped)
  - The escape character (unless escaped)
  - The end character (unless escaped)

- The presence of the escape character has no effect on characters that have special meaning to individual stages. For example, do not use the escape character within a delimited string in an attempt to include the delimiter character in the string.

- See the individual stage commands for other restrictions that apply.

**Return Codes**

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The pipeline ran successfully.</td>
</tr>
<tr>
<td>8</td>
<td>An option field error occurred.</td>
</tr>
<tr>
<td>12</td>
<td>A PIPE syntax error occurred, or a stage command separator character was used improperly.</td>
</tr>
<tr>
<td>16</td>
<td>A stage specification error occurred.</td>
</tr>
<tr>
<td>20</td>
<td>A storage failure occurred during PIPE interpretation.</td>
</tr>
<tr>
<td>24</td>
<td>The pipeline is clogged. A deadlock occurred during pipeline processing.</td>
</tr>
</tbody>
</table>

For information on debugging clogged pipeline conditions see "Chapter 8. Debugging NetView Pipelines" on page 297.

-1          | An unrecoverable error occurred during processing, it was possibly a looping condition. |

-5          | A RESET condition occurred.                      |

**Examples**

**Example: Changing the Separation Character with STAGESEP**

To change the stage command separation character from the default value of a vertical bar (|) to a period (.), run a NetView LIST command, and display the resulting messages, enter:

`PIPE (STAGESEP .) NETVIEW LIST STATUS=TASKS . CONSOLE`
Example: Changing the Separation Character and Setting an Escape Character

To change the stage command separation character from the default value of a vertical bar (|) to a period (.), use double quotes (") as an escape character, run a NetView LIST command, discard messages containing the phrase NOT ACTIVE in positions 55 through 64, and display the resulting messages, enter:

```
PIPE (STAGESEP . ESC ") NETVIEW LIST STATUS=TASKS
   NLOCATE 55".10 /NOT ACTIVE/
   CONSOLE
```

In this example, the escape character is used, so that the period separating the NLOCATE search parameters is not read as a stage command separator.

PIPE Stages: Table 5 contains an alphabetical summary of the stage commands in the PIPE command and shows the minimum synonym allowed for each stage command in a pipeline specification.

The NetView sample CNMS1101 contains many of the examples shown in the stage descriptions.

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Task Performed</th>
<th>Synonym</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND</td>
<td>Defines a pipeline which will run after other stages are complete.</td>
<td>APPEND</td>
<td>27</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>Divides message streams into sections.</td>
<td>BETWEEN</td>
<td>28</td>
</tr>
<tr>
<td>CASEI</td>
<td>Compares character strings without respect to case.</td>
<td>CAS</td>
<td>31</td>
</tr>
<tr>
<td>CHANGE</td>
<td>Replaces occurrences of one string with another.</td>
<td>CHAN</td>
<td>33</td>
</tr>
<tr>
<td>CHOP</td>
<td>Truncates lines after a specified character, column, or string.</td>
<td>CHOP</td>
<td>35</td>
</tr>
<tr>
<td>COLLECT</td>
<td>Creates a multiline message, or messages, from input lines.</td>
<td>COL</td>
<td>37</td>
</tr>
<tr>
<td>CONSOLE</td>
<td>Displays the contents of the pipeline.</td>
<td>CON</td>
<td>42</td>
</tr>
<tr>
<td>CORRCMD</td>
<td>Runs commands and adds timer and termination stages.</td>
<td>CC</td>
<td>45</td>
</tr>
<tr>
<td>CORRWAIT</td>
<td>Allows asynchronous messages into the pipeline.</td>
<td>CORR, WAIT</td>
<td>49</td>
</tr>
<tr>
<td>COUNT</td>
<td>Counts the number of messages, lines, or bytes in the input stream.</td>
<td>COUNT</td>
<td>54</td>
</tr>
<tr>
<td>CPDOMAIN</td>
<td>Converts control point (CP) names to NetView domain names.</td>
<td>CPD</td>
<td>55</td>
</tr>
<tr>
<td>DELDUPES</td>
<td>Deletes duplicate messages.</td>
<td>DELDUP</td>
<td>60</td>
</tr>
<tr>
<td>DIVERT</td>
<td>Routes primary input to primary or secondary output.</td>
<td>DIVERT</td>
<td>63</td>
</tr>
<tr>
<td>DROP</td>
<td>Specifies the number of messages to be discarded from the pipeline.</td>
<td>DROP</td>
<td>64</td>
</tr>
<tr>
<td>DUPLICAT</td>
<td>Copies messages in the input stream.</td>
<td>DUP</td>
<td>64</td>
</tr>
<tr>
<td>EDIT</td>
<td>Creates or reformats messages.</td>
<td>EDIT</td>
<td>68</td>
</tr>
<tr>
<td>ENVADATA</td>
<td>Outputs environment data.</td>
<td>ENV</td>
<td>53</td>
</tr>
<tr>
<td>EXPOSE</td>
<td>Causes messages to be exposed for automation and logging.</td>
<td>EXPOSE</td>
<td>100</td>
</tr>
<tr>
<td>FANIN</td>
<td>Merges multiple input streams, in stream order, into a single output stream.</td>
<td>FANIN</td>
<td>102</td>
</tr>
<tr>
<td>FANINANY</td>
<td>Merges multiple input streams, preserving the message order, into a single output stream.</td>
<td>FANINANY</td>
<td>105</td>
</tr>
<tr>
<td>FANOUT</td>
<td>Passes a single input stream to multiple output streams.</td>
<td>FANOUT</td>
<td>105</td>
</tr>
<tr>
<td>Stage Command</td>
<td>Task Performed</td>
<td>Synonym</td>
<td>Page</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------------------------------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>HELDMSG</td>
<td>Reads a copy of the held messages queue into the pipeline.</td>
<td>HELD</td>
<td>108</td>
</tr>
<tr>
<td>HOLE</td>
<td>Discards the contents of the pipeline. Determines whether a command has correlated output.</td>
<td>HOLE</td>
<td>110</td>
</tr>
<tr>
<td>INSTORE</td>
<td>Adds, deletes or replaces in-storage members.</td>
<td>INSTORE</td>
<td>112</td>
</tr>
<tr>
<td>INTERPRT</td>
<td>Builds and runs stage commands from input command data. Facilitates long pipe commands.</td>
<td>INT</td>
<td>115</td>
</tr>
<tr>
<td>JOINCONT</td>
<td>Joins consecutive messages that match a specified string.</td>
<td>JOINCONT</td>
<td>116</td>
</tr>
<tr>
<td>KEEP</td>
<td>Defines a task global place to store messages.</td>
<td>KEEP</td>
<td>122</td>
</tr>
<tr>
<td>LITERAL</td>
<td>Inserts text into the pipeline.</td>
<td>LIT</td>
<td>124</td>
</tr>
<tr>
<td>LOCATE</td>
<td>Selects messages that match a specified character string to remain in the pipeline.</td>
<td>LOC</td>
<td>125</td>
</tr>
<tr>
<td>LOGTO</td>
<td>Sends a copy of the contents of the pipeline to a specific log.</td>
<td>LOG</td>
<td>127</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>Matches data within a pipeline.</td>
<td>LOOKUP</td>
<td>129</td>
</tr>
<tr>
<td>MEMLIST</td>
<td>Creates a list of members in one or more partitioned data sets (PDS) or data definitions (DD).</td>
<td>MEML</td>
<td>133</td>
</tr>
<tr>
<td>MVS</td>
<td>Runs specified MVS commands.</td>
<td>MVS</td>
<td>135</td>
</tr>
<tr>
<td>NETVIEW</td>
<td>Runs specified NetView commands.</td>
<td>NETV</td>
<td>137</td>
</tr>
<tr>
<td>NLOCATE</td>
<td>Discards messages that match a specified character string.</td>
<td>NLOC</td>
<td>141</td>
</tr>
<tr>
<td>NLS</td>
<td>Converts input messages to their translated versions.</td>
<td>NLS</td>
<td>143</td>
</tr>
<tr>
<td>NOT</td>
<td>Changes the way output is treated by those stages that discard part of their output.</td>
<td>NOT</td>
<td>145</td>
</tr>
<tr>
<td>PERSIST</td>
<td>Specifies the disposition for correlated output after a pipeline ends.</td>
<td>PERSIST</td>
<td>147</td>
</tr>
<tr>
<td>PICK</td>
<td>Selects messages to remain in the pipeline based on a comparison of two strings.</td>
<td>PICK</td>
<td>149</td>
</tr>
<tr>
<td>PIPEND</td>
<td>Causes a pipeline to end and return a return code.</td>
<td>PIPEND</td>
<td>152</td>
</tr>
<tr>
<td>PPI</td>
<td>Passes data to a PPI receiver.</td>
<td>PPI</td>
<td>154</td>
</tr>
<tr>
<td>PRESATTR</td>
<td>Changes the way messages are displayed on the NetView console.</td>
<td>COLOR, COLOUR</td>
<td>160</td>
</tr>
<tr>
<td>QSAM</td>
<td>Reads from and writes to dynamically allocated data definition names or data sets.</td>
<td>QSAM, &gt;</td>
<td>153</td>
</tr>
<tr>
<td>REVERSE</td>
<td>Changes the order of message text and message lines.</td>
<td>REV</td>
<td>156</td>
</tr>
<tr>
<td>ROUTE</td>
<td>Sends messages to another task.</td>
<td>ROUTE</td>
<td>158</td>
</tr>
<tr>
<td>SAFE</td>
<td>Reads or writes messages to a command procedure message queue.</td>
<td>SAFE</td>
<td>170</td>
</tr>
<tr>
<td>SEPARATE</td>
<td>Breaks MLWTOs into multiple single-line messages.</td>
<td>SEP</td>
<td>172</td>
</tr>
<tr>
<td>SORT</td>
<td>Sorts input stream messages.</td>
<td>SORT</td>
<td>176</td>
</tr>
<tr>
<td>SPLIT</td>
<td>Divides a line of text into multiple lines.</td>
<td>SPLIT</td>
<td>176</td>
</tr>
<tr>
<td>SQL</td>
<td>Queries DB2 tables, inserts rows into DB2 tables, and issues DB2 commands.</td>
<td>SQL</td>
<td>181</td>
</tr>
<tr>
<td>SQLCODES</td>
<td>Used for diagnostics when using the SQL stage</td>
<td>SQLCODES</td>
<td>188</td>
</tr>
<tr>
<td>STEM</td>
<td>Reads or writes records to or from command procedure variables.</td>
<td>STEM</td>
<td>189</td>
</tr>
<tr>
<td>STRIP</td>
<td>Removes characters from the beginning or end of a message.</td>
<td>STRIP</td>
<td>192</td>
</tr>
</tbody>
</table>
Pipeline Stages and Syntax

Table 5. Stage Commands of the PIPE Command (continued)

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Task Performed</th>
<th>Synonym</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSYM</td>
<td>Substitutes MVS and user-defined system symbolics in messages in the pipeline.</td>
<td>SUBS</td>
<td>196</td>
</tr>
<tr>
<td>TAKE</td>
<td>Specifies the number of messages to be kept in the pipeline.</td>
<td>TAKE</td>
<td>197</td>
</tr>
<tr>
<td>TOSTRING</td>
<td>Ends the data stream when a specific character string is located.</td>
<td>TOS</td>
<td>199</td>
</tr>
<tr>
<td>TSO</td>
<td>Runs specified TSO commands.</td>
<td>TSO</td>
<td>201</td>
</tr>
<tr>
<td>TSROUTE</td>
<td>Passes data to Topology Display Servers.</td>
<td>TSR</td>
<td>207</td>
</tr>
<tr>
<td>UNIX®</td>
<td>Runs specified UNIX commands.</td>
<td>UNIX</td>
<td>202</td>
</tr>
<tr>
<td>VAR</td>
<td>Reads or writes records to or from command procedure variables.</td>
<td>VAR</td>
<td>214</td>
</tr>
<tr>
<td>VARLOAD</td>
<td>Sets variables to a specified value.</td>
<td>VARLOAD</td>
<td>217</td>
</tr>
<tr>
<td>VET</td>
<td>Reads or writes data to, or from, a virtual screen belonging to a virtual OST (VOST).</td>
<td>VOSTIO</td>
<td>222</td>
</tr>
<tr>
<td>VTAM</td>
<td>Runs specific VTAM commands in a local or remote domain.</td>
<td>VTAM</td>
<td>227</td>
</tr>
<tr>
<td>XLATE</td>
<td>Translates uppercase, lowercase, ASCII, and EBCDIC characters.</td>
<td>XLATE</td>
<td>230</td>
</tr>
<tr>
<td>$STEM</td>
<td>Same as STEM, plus reads or writes VIEW attribute variables associated with specified data variables.</td>
<td>$STEM</td>
<td>192</td>
</tr>
<tr>
<td>$VAR</td>
<td>Same as VAR, plus reads or writes VIEW attribute variables associated with specified data variables.</td>
<td>$VAR</td>
<td>214</td>
</tr>
<tr>
<td>&lt; (From Disk)</td>
<td>Reads data from DASD into the pipeline.</td>
<td>&lt;</td>
<td>232</td>
</tr>
</tbody>
</table>
PIPE APPEND

Syntax

APPEND:

Command Description

APPEND defines a pipeline that runs after the preceding stage is complete. Arguments to APPEND are a series of stage specifications separated by the same stage separator (STAGESEP) character defined for the pipeline. Because the same stage separator is used, it must be escaped using either the defined escape character (ESC) or using the self-escaping technique of coding duplicate stage separators.

APPEND begins processing by initializing the stages specified as arguments. The first stage coded after APPEND will be treated as a first stage and the last stage generated by APPEND will have its primary output stream connected to the input stream of the stage following APPEND.

For more information on STAGESEP, ESC, and the self-escaping technique, refer to "PIPE (NCCF)” on page 20.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Operand Descriptions

|| The stage separator used within the APPEND definition. This stage separator must be the same as that used in the rest of the pipeline. However, it must be escaped using either the defined ESC character or using the self-escaping technique of coding two stage separators together.

If an APPEND is included within an APPEND, the inner APPEND must escape the stage separators of the higher-level APPEND. For example, if the first APPEND uses || for its stage separator, an APPEND within that APPEND would use ||||.

stage_specification

A complete stage specification including parameters. The stage_specification cannot include labels or connectors.

Usage Notes

- All processing of the preceding stage must complete before the APPEND stages are run.
- APPEND stages can be nested.
Examples

**Example: Switch and Send LIST DST Results to Operator**
The following switches the DSILOG, waits for the completion of the switch, issues a LIST DST and sends the results of both commands to the operator:

/* Route SWITCH and LIST DST outputs to operator with attributes */

```
'PIPE (NAME APPNXMP)'
  NETVIEW SWITCH DSILOG,P', /* Begin the switch */
  CORRWAIT 30', /* CORRWAIT several seconds */
  PRESATTR UND', /* Underscore SWITCH output only */
  APPEND', /* following stages run after wait */
  NETV LIST DSILOG', /* list runs AFTER switch completes */
  WAIT 5', /* this wait starts AFTER LIST runs */
  PRESATTR REVERSE', /* reverse-video LIST output */
  PRESATTR YELLOW', /* color output of both commands yellow */
  CONSOLE' /* send command outputs to the console */
```
PIPE BETWEEN

Syntax

BETWEEN:

```
INCL
NOINCL
pos.len
```

Command Description

The BETWEEN stage command is a selection stage that divides a message stream into sections. The selected sections begin with a message containing a specified string and end with either a specified string or a number of messages.

The selected sections are passed to the primary output stream. The sections that were not selected are passed to the secondary output stream.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

BETWEEN terminates when the input stream or both output streams disconnect.

Operand Descriptions

**INCL**

Specifies that the message or messages that match the first string and, if specified, the second string should be included in the output. A message that matches a count as a second condition is always included.

**NOINCL**

Specifies that the message or messages that match the first string and, if specified, the second string should not be included in the primary output.

**number**

Specifies a message count to be included in the selected section.

**pos.len**

Specifies the character position and length within each message where searching should begin. When you specify a wildcard (*) for len, the remainder of each line is searched. When you do not specify pos.len, each entire line is searched.
PIPE BETWEEN

/1st string/
A delimited string that specifies the string to be found that begins a selected section. This parameter is required and can be used with /2nd string/ or number.

/2nd string/
A delimited string that specifies the string to be found that ends a selected section.

Usage Notes
The BETWEEN stage examines only the first line of multiline messages. Specifying the SEPARATE stage before BETWEEN causes all lines to be examined.

Examples

Example: Dividing Messages
The following divides a message stream into three groups:

```
PIPE < CNMPNL1.EUYSLIST
    TAKE 90
    BETWEEN 1.14 /:IF DTYPE=MSG/ 1.6 /:ENDIF/
    CONSOLE

:IF DTYPE=MSGS
    Enter HELP PIPE DSIVSAM to display syntax and general usage.
    Enter HELP PIPE DSIVSAM DEL to receive help on deleting records.
    Enter HELP PIPE DSIVSAM GET to receive help on reading VSAM records.
    Enter HELP PIPE DSIVSAM GETREV to receive help on reading VSAM records in reverse sequence.
    Enter HELP PIPE DSIVSAM INQUIRE to receive help on displaying data set characteristics for a file.
    Enter HELP PIPE DSIVSAM PUT to receive help on creating or replacing records.
:ENDIF

:IF DTYPE=MSGS
    Enter HELP PIPE DSIVSMX to display syntax and general usage notes.
    Enter HELP PIPE DSIVSMX CLOSE to receive help on closing records.
    Enter HELP PIPE DSIVSMX DEL to receive help on deleting records.
    Enter HELP PIPE DSIVSMX GET to receive help on reading records.
    Enter HELP PIPE DSIVSMX GETREV to receive help on reading records in reverse sequence.
    Enter HELP PIPE DSIVSMX IDCAMS to receive help on data set maintenance.
    Enter HELP PIPE DSIVSMX INQUIRE to receive help on displaying data set characteristics for a file.
    Enter HELP PIPE DSIVSMX OPEN to receive help on opening files locally to the issuing task.
:ENDIF
```
Syntax

\[
\text{CASEI:} \\
\underline{\text{-CASEI stage_specification-}}
\]

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASEI</td>
<td>CAS</td>
</tr>
</tbody>
</table>

Command Description

The CASEI stage command causes the specified stage to compare character strings without sensitivity to uppercase or lowercase EBCDIC characters. For example, LOCATE /AbC/ without CASEI will not match a line containing 'ABC', but CASEI LOCATE /AbC/ will match such a line.

It is only useful to use CASEI with stages specifying character strings, like LOCATE, TOSTRING, CHOP, and CHANGE. It is also only useful to use CASEI in environments that do not uppercase the entire PIPE command (NETVASIS environments).

Termination Conditions

CASEI modifies the stage specified in stage_specification. Because it is a modifier stage, CASEI does not have termination conditions of its own. See the information on the stage CASEI is modifying for termination conditions.

Operand Descriptions

\[\text{stage_specification}\]

The stage specification being modified, including its operands.

Examples

**Example: Locating Strings Regardless of Case**
The following example locates lines containing a particular letter, regardless of case.

```
NETVASIS PIPE LITERAL /Abcdefghi/
 | LITERAL /abc/  
 | LITERAL /xyz/   
 | CASEI LOCATE /AbC/  
 | CONSOLE

----> abc
----> Abcdefghi
```

**Example: Using CASEI with CHANGE Stage**
The following example locates all occurrences of a particular string, regardless of case, and outputs the substitute string exactly as entered.
PIE CASEI

NETVASSIS PIPE LITERAL /He did not say "IT IS NOT"!/
   CASEI CHANGE / not/ too/
   CONSOLE

--- He did too say "IT IS too"!

See also "Example: Using CHANGE with CASEI Stage" on page 34.
PIECE CHANGE

Syntax

CHANGE:

```
<table>
<thead>
<tr>
<th>CHANGE</th>
<th>1.*</th>
<th>changestring</th>
<th>/string/</th>
<th>/string/</th>
<th>numchg</th>
</tr>
</thead>
<tbody>
<tr>
<td>position.length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGE</td>
<td>CHAN</td>
</tr>
</tbody>
</table>

Command Description

The CHANGE stage command replaces occurrences of the first specified string with the second string. Either string can be null. If the first string is null, the second string is inserted at the beginning of each line. If the second string is null, all occurrences of the first one are deleted. Data between substitutions are copied without change.

If a secondary output stream is defined, then messages in which a change was made are passed to the primary output stream and messages in which no changes were made (because first string is not found) are passed to the secondary output stream. If either output stream is defined, but disconnected, messages normally sent to the disconnected output stream are discarded. If no secondary output stream is defined, then all changed and unchanged messages are passed to the primary output stream.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

CHANGE terminates when the input stream or both output streams disconnect.

Operand Descriptions

- `changestring`
  
  A shorthand form of writing two consecutive delimited strings. For example, `/a/ /b/` may be combined into `/a/b/` (use the same delimiter all three times).

- `numchg`
  
  The maximum number of substitutions to be made in each message. Note that there can be multiple lines per message. If `numchg` is not specified, the default is to change all occurrences in each message.
### PIPE CHANGE

**position.length**

Specifies the character position in each message where searching begins, and the length of the search. If you specify a `length` of *, the remainder of each line is searched. If you do not specify a `position.length`, the entire line is searched.

**/string/**

A character string enclosed in delimiters.

### Usage Notes

CHANGE cannot be the first stage command.

### Examples

**Example: Changing All Occurrences of a String**
The following example changes all occurrences of 'AAA' to 'ZZ'.

```plaintext
PIPE LITERAL /AAAABBBAAA/
| CHANGE /AAA/ZZ/
| CONSOLE
```

**Example: Inserting Constants**
The following example inserts a constant at the start of each line.

```plaintext
PIPE NETVIEW LIST ''
| CHANGE //MSG TOM /
| NETVIEW
| CONSOLE
```

```plaintext
---> DSI001I MESSAGE SENT TO TOM
---> DSI001I MESSAGE SENT TO TOM
---> DSI001I MESSAGE SENT TO TOM
---> DSI001I MESSAGE SENT TO TOM
---> DSI001I MESSAGE SENT TO TOM
---> DSI001I MESSAGE SENT TO TOM
---> DSI001I MESSAGE SENT TO TOM
---> DSI001I MESSAGE SENT TO TOM
```

**Example: Using CHANGE with CASEI Stage**
Starting from column 2, the following example changes up to 4 occurrences of 'A' or 'a' to 'Zz'.

```plaintext
NETVASIS PIPE LITERAL /AaBbAaCcAa/
| CASEI CHANGE 2.* /a/Zz/ 4
| CONSOLE
```

See also the PFKDEF command list (CNME1010) and "Example: Using CASEI with CHANGE Stage" on page 31.
Syntax

CHOP:

Synonyms

CHOP has no synonyms.

Command Description

The CHOP stage command truncates lines after a specified column, character, or string.

The data kept by CHOP is passed to the primary output stream. The data discarded by CHOP is passed to the secondary output stream, if connected.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

CHOP terminates when the input stream or both output streams disconnect.

Operand Descriptions

AFTER

Specifies that the offset refers to the number of positions after the end of the matching string or character.

ANYof

Specifies that CHOP should search for the first occurrence of any character in /string/.

BEFORE

Specifies that the offset refers to the number of positions before the beginning of the matching string or character. This is the default.

column

Column after which truncation is to occur.

NOT

Specifies that truncation should be relative to the first character or string that does not match the specified target.
PIPE CHOP

offset
The truncation column relative to the beginning or end of the matching string or character. This can be a negative value. The default is zero.

STRING
Specifies that CHOP should search for the first exact occurrence of /string/.

/string/
A character string enclosed in delimiters.

width
The default column. For OST tasks, the screen width is used. For automation tasks and other tasks, zero is used.

Usage Notes
CHOP cannot be the first stage command.

Examples

Example: Truncating Lines
The following example truncates all lines after column 44.

PIPE NETV QRYGLOBL COMMON VARS**
  | SEPARATE
  | LOCATE /BNH039/
  | CHOP 44
  | CONS ONLY

---> BNH039I SMFVPD
---> BNH039I CGAUTHID2
---> BNH039I LONGONE
...

Example: Truncating Text
The following example truncates all text from the character before the first number.

PIPE LITERAL /GOOD STUFF 00001204/
  | CHOP 1 BEFORE ANYOF /1234567890/
  | CONSOLE

---> GOOD STUFF

Example: Isolating Text Within a Line
Coded as a REXX example:

'PIPE NETV necessary command ',
  ' | SEPARATE',
  ' | LOCATE 1.7 /IST486I/','
  ' | CONSOLE',
  ' | NOT CHOP AFTER STRING /STATUS= /','
  ' | CHOP BEFORE STRING /,/',
  ' | VAR state'
SAY 'The value of state is ' state

---> IST486I STATUS= ACTIV    , DESIRED STATE= ACTIV
---> The value of state is  ACTIV
PIEPE COLLECT

Syntax

COLLECT:

ASBEFORE

BREAK

AFTER

BEFORE

pos.len

/string/

Syonyms

<table>
<thead>
<tr>
<th>Stage Command or Operand</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLECT</td>
<td>COL</td>
</tr>
<tr>
<td>ASBEFORE</td>
<td>ASB4</td>
</tr>
</tbody>
</table>

Command Description

The COLLECT stage command creates a multiline message, or messages, from the input messages. The attributes of the output are inherited from the first line collected for that message.

If a secondary input stream is defined for the COLLECT stage:

- COLLECT reads input from the secondary input stream until it becomes disconnected. After the disconnect, it reads from the primary input stream.
- COLLECT uses all lines read from the secondary input stream as label lines for the produced multiline messages.

Note: COLLECT accepts any number of lines on the secondary input stream to be used as label lines. However, NetView presentation services will only recognize the first six lines as label lines. The remainder will be treated as data lines.

- All lines, except the last, read from the primary input stream will be used by COLLECT as data lines, regardless of their current state, in the produced multiline message. The last line received on the primary input stream will be used as an end line.

If a secondary input stream is not defined for the COLLECT stage:

- For each multiline message produced, COLLECT will accept up to six label lines from the primary input stream. These lines must have previously been designated as label lines.
- All lines after the sixth, or after the first data line, are handled as data lines regardless of how they were previously designated.
PIPE COLLECT

- The last line received on the primary input stream is used as an end line.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

COLLECT terminates when the primary input stream or the output stream is disconnected.

Operand Descriptions

If no operands are given, COLLECT will create one multiline message containing all input lines, and must delay its output until the previous stage disconnects.

AFTER
Specifies that the message containing the matching string should be appended to whatever is collected at that point.

ASBEFORE
Specifies that the output should be collected as it was previously. This usually applies to messages obtained using the SEPARATE stage, but can also apply to messages replicated exactly using other techniques.

AT
Discard the entire message containing the matching string. This is the default.

BEFORE
Specifies that when the message containing the matching string is detected, it should be kept to begin a new multiline message and not be part of the output produced.

BREAK
Causes collected lines to be issued when a string match is detected. A delimited string is required if BREAK is used. See the AT, AFTER, and BEFORE keywords for disposition of the message containing the matching string.

MAX
Specifies setting a limit for the maximum number of messages (not lines). This option is used with the number variable.

number
Specifies the maximum number of input messages that are to be collected.

pos.len
Specifies where, within each line of input, COLLECT should attempt to find the specified delimited string.

/string/
Specifies the character string for which to search. If BREAK is specified, the delimited string is required; otherwise, it is not allowed. You can specify /string/ up to 40 times.
The first nonblank character encountered after BREAK AT, BREAK BEFORE, or BREAK AFTER is the delimiter which establishes the boundary of the text string used by the stage. The delimited string ends when the same character is encountered a second time.

Usage Notes

- COLLECT cannot be a first stage command.
- Each of the COLLECT parameters specifies a condition to stop collecting. With the addition of the MAX parameter, a single COLLECT stage can have two such conditions. When either condition is met, a multiline message is produced and COLLECT starts again.
- If you use the COLLECT command following a STEM command, see the description of the COLLECT operand of the STEM command (Operand Descriptions on page 174) for a more efficient alternative.
- The COLLECT stage directly affects the way that messages in the pipeline are displayed, logged, and searched by other stages. It can be used to improve the readability of messages when displayed to the operator console.
- The COLLECT stage delays the stream. This means that subsequent stages cannot process data until COLLECT completes its collection of a multiline message. When COLLECT is used without arguments, COLLECT will not produce output until the previous stage disconnects.
- The message attributes of an output message from COLLECT is derived from the first message included in that output. For example, if the first message received by COLLECT had a JOBNAME of TSU00041, then the entire multiline output would bear that same JOBNAME regardless of the origin of the subsequent lines.
- COLLECT does not alter the message type (HDRMTYPE) of the lines it collects. This means that the output is not necessarily one of the types J, K, or L as was the practice in early releases of NetView.
- COLLECT can cause a complex pipeline to become deadlocked or clogged. For information on resolving clogged pipelines, see Chapter 8, Debugging NetView Pipelines on page 297.

Examples

Example: Converting Single-Line Messages to Multiline
To present the output of LIST STATUS=TASKS as a multiline message, enter:

```
PIPE NETVIEW LIST STATUS=TASKS
| COLLECT
| CONSOLE
```

The many single-line messages issued by LIST are saved by COLLECT until all output of the NETVIEW stage has been gathered. Then a single MLWTO is produced. If this command is running in a remote autotask (using RMTCMD), then the processing effort for the network is far less for the one multiline message than for the many single-line messages.

Example: Displaying Multiline Messages and Preserving Output Structure
In the following example the 'IST350I DISPLAY TYPE = LINES' is removed without disturbing the multiline structure of the output; IST097 is by itself and the others are together.
PIPE COLLECT

PIPE CORRCMD D NET,LINES
| SEPARATE
| NLOCATE 1.7 /IST350I/
| COLLECT ASBEFORE
| CONSOLE

---> IST097I DISPLAY ACCEPTED
--->
IST354I PU T4/S MAJOR NODE = NT7EVTAM
IST172I NO LINES EXIST
IST231I CA MAJOR NODE = NT7ECTCA
IST170I LINES:
IST080I CTCALN7E ACTIV----E
IST314I END

See also the example in the LOGPROF1 command list (CNME1049).

Example: Formatting LIST STATUS=TASKS Output

In the following example the common information contained in all LIST STATUS=TASKS messages, normally found in each output line, is presented in multiple label lines before presenting the data.

The simple pipeline FAN:FANIN|LABS: passes the data to COLLECT in the order the streams were defined. This allows color or other processing to be done on the message prior to the COLLECT stage.

By adding DEBUG you see the connection flow within the pipeline. For more information about debug, see "DEBUG Stage Option" on page 300.

/* REXX Example */

address NETVASIS,
| PIPE (NAME TASKLIST END \)
| NETV LIST STATUS=TASKS, /* generate the data */
| DROP LAST 1', /* no need of "END OF DATA" */
| COLOR GREEN', /* standardize buffers */
| EDIT WORD 2 1', /* reformat data from the lines */
| '19.8 8',
| '38.8 19',
| '55.* 35',
| 'LABS: COLLECT', /* data and labels, labels read first */
| CONSOLE ONLY', /* display results */
| /* --- END of simple pipeline, begin new pipeline... */
| \ FAN: FANIN', /* feed data to "LABS", in order */
| \ LABS:', /* --- END of simple pipeline, begin new pipeline... */
| \ LIT !-------- Status of NetView Tasks ---------!
| \ COLOR YEL', /* Control line becomes yellow */
| \ FAN:', /* give line to "FAN" (primary input) */
| \ LIT !Task Task's Taskname or Current!
| \ COLOR PINK', /* First label line becomes pink */
| \ FAN:', /* give line to "FAN" (2nd input) */
| \ LIT !type ID Resource Status!
| \ COLOR PINK', /* Second label line becomes pink */
| \ FAN:', /* give line to "FAN" (3rd input) */

Example: Formatting MVS D A,L Output

In the following example, each new output line contains information about only one job. The original three label lines will still be label lines.

The source for this example is in CNMS1101.

/* REXX Example */

'PIPE (NAME DISPAL END \)',

Tivoli NetView for z/OS Customization: Using Pipes
PIPE COLLECT

CORCMD MVS D A,L', /* generate data */
SEPARATE', /* handle lines individually */
  /* SEP preserves the line type (ctl, label, data, end) */
A: TAKE 3', /* first 3 to primary, rest to 2ndary */
COLOR WHITE', /*Ctl & label become white */
Z: FANINANY', /* all lines return here... */
COLLECT', /* collect all lines, preserve type */
CONSOLE ONLY',
\ A:', /* data lines come here... */
\ C: CHOP 35', /* left side to primary, rest to "C" */
  /* CHOP also preserves the line type */
COLOR BLUE', /* some data becomes blue */
Z:', /* give in back to FANINANY */
\C:', /* right side of data comes here */
\ COLOR TUR', /* other data becomes turquoise */
Z:' /* give in back to FANINANY */
PIPE CONSOLE

Syntax

CONSOLE:

- CLEAR
- LOCK
- ONLY
- DUMP
- DELETE
- XDUMP

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSOLE</td>
<td>CON</td>
</tr>
</tbody>
</table>

Command Description

The CONSOLE stage command specifies one of the following actions:

- Displays messages on the screen and write them to the output stream.
- Reverses the held status of a message (using the DELETE option).

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

CONSOLE terminates when the input stream is disconnected.

Operand Descriptions

CLEAR

Specifies to clear the screen immediately before the first message is displayed.

DELETE

Specifies to remove a held message status. The message on the screen is immediately reverted to the color and highlighting of normal messages and is removed when the screen is refreshed.

DUMP

Specifies that each line of each message is to be presented in dump format, showing HEX and EBCDIC representations.

LOCK

Specifies to lock the screen. Three asterisks (***), are displayed on the indicator line immediately before the first message is displayed. After the screen is locked, normal NetView autowrap rules apply.

ONLY

Specifies that messages are to be displayed but not held or exposed. For
example, use the ONLY option when redisplaying messages brought into the pipeline with the HELDMSG stage command.

An exposed message is a message that fits all of the following criteria:
- Passed to DSIEX02A
- Copied or routed to satisfy ASSIGN command actions
- Matched for &WAIT or TRAP conditions
- Matched for actions specified in the NetView automation table
- Passed to DSIEX16
- Logged to the network log, hardcopy log, or system log.

Note: Display of messages is subject to the DISPLAY setting of the DEFAULTS and OVERRIDE commands. Messages displayed this way are subject to NLS translation.

**XDUMP**

Produces the same output as DUMP. Also included are the AIFR body and other information that accompanies some messages. For example, MVS messages include MDB data as indicated in mapping DSIAIFRO. See "Example: Displaying Messages in Dump Format" on page 44.

Reference: Refer to the Tivoli NetView for z/OS Customization: Using Assembler for mapping for DSIIFR for AIFR information.

**Attention:** DUMP and XDUMP are not intended as a programming interface, because formats might change and additional data might be included in the future. Use DUMP and XDUMP only for problem determination.

**Usage Notes**

- CONSOLE cannot be the first stage command.
- The CONSOLE stage presents copies of the messages that come to it in the same way as other NetView commands. Thus, when CONSOLE is found on an inner pipeline (a pipeline running as a result of a PIPE command on a NETVIEW stage on another outer pipeline), the output of CONSOLE is trapped by the outer pipeline and is passed to the next stage in that outer pipeline.
- The ONLY option can be used to avoid filling the log with extraneous messages.
- A PIPE command submitted to NetView from the MVS environment can have a command and response token (CART) value associated with the command. (You can do this from TSO command procedures.) In this case, output from the CONSOLE stage carries the CART for accurate correlation.
- When using multiple CONSOLE stages, the order of displayed messages (whether they are single-line messages or MLWTOs) is unpredictable. To control the order, use the COLLECT stage preceding the CONSOLE stage to collect pipeline messages into an MLWTO.
- If you use the CONSOLE DELETE stage subsequent to a CONSOLE or a CONSOLE ONLY stage, some messages can be deleted by the CONSOLE DELETE stage before they are displayed.
Examples

Example: Displaying Messages on a Console
To issue a NetView LIST command, discard messages containing the phrase NOT ACTIVE in positions 55 through 64, and display the resulting messages (after clearing the screen), enter:

```
PIPE NETVIEW LIST STATUS=TASKS
   | NLOCATE 55.10 /NOT ACTIVE/
   | CONSOLE CLEAR
```

Example: Displaying Messages in Dump Format
The following example shows message output using the XDUMP parameter.

```
* NTV7E TOM PIPE CORRCMD MVS D T | CONS XDUMP
- NTV7E TOM

-------- AIFR body ---------------------
040AE260 000C0100 00C90024 | ü* I *
040AE270 1127520C D5E3E5F7 C5404040 00000000 ***ê*NTV7E
040AE280 00000000 E3D60440 40404040 0017A000 **TOM **μ
040AE290 00000624 007FD068 01F50000 04486568 ** ")Ç*5 *ç@I
040AE2A0 04486658 40410400 00030000 00000000 *ç@Iâ** *
040AE2B0 00000000 00000000 E3D60440 40404040 **TOM
040AE2D0 00000000 00000000 00000000 00000000
040AE2F0 00000000 00000000 00000000 00000000
040AE310 01000002 C6F7F904 E5E24040 00000000 *F97MVS
040AE320 00000000 00000000 00000000 00000000
040AE330 00000000 00000000 00000000 00000000
040AE340 00000000 00000000 00000000 00000000
040AE350 00000000 00000000 00000000 00000000
040AE360 00000000 040AE8B8 00C90024 | ü* I *
- NTV7E TOM

-------- Aifro (GDS) ---------------------
040AE800 000C0100 00C90011 | S ** *
040AE810 00000000 00000000 00020006 00CE0007 K * O *
040AE820 040AE820 00000001 00380001 000624 MDB ** ** **
040AE830 F1F14BF2 F74BF5F2 4BF3F200 F1F9F5F5 11.27.32.32 1995 325
040AE840 F3F2F500 00000000 00000000 00000000
040AE850 C6F7F904 E5E24040 E2E3F904 F0F0F0F5 F74BF5F2 00000000 00000000
040AE860 008A0002 00000004 D4E5E240 C8C2C2F5 ** * MVS HB5
040AE870 F5F1F040 00000000 00000000 00000000 510
040AE880 00000000 00000000 00000000 00000000 ** * - 
040AE890 007FD068 00000000 0001800 00E3C3F0 "}Ç * STCO
040AE8A0 F0F0F0F5 40404040 40404040 40404040 40404040 00000000 00000000
040AE8B0 00000000 00000000 00C13833 3279325 01000002 00000000
040AE8C0 00000000 00000000 00000000 00000000 00000000 00000000
040AE8D0 00000000 00000000 00000000 00000000 00000000 00000000
040AE8E0 D5E3D36 00C90011 00C90011 | E320EG
- NTV7E TOM

-------- Message data ---------------------
04486650 004C0007A 00C500E2 | < : E *
04486660 1127520C D5E3E5F7 C5404040 00000000 ***ê*NTV7E
04486670 00000000 E3D60440 40404040 00570004 **TOM **μ
04486680 00000000 00E4C9C5 00F1F9F5 00F1F9F5 + 4 UIEE136I L0
04486690 C3C1D37A 00E3C904 00C5E1F9 04F8F748 12:00 TIME=11.27.
04486700 F5F2400C C1E3C578 00F1F9F5 04F8F748 52 DATE=1995.325
04486710 4004C07D 0037A4E3 00C4C57E F1F648F2 00F1F9F5 04F8F748 GMT: TIME=16.2
04486720 F74BF5F2 004C41E3 00C5E1F9 09F548F3 7.52 DATE=1995.3 25
04486730 F2F5
PIPE CORRCMD

Syntax

CORRCMD:

```plaintext
- CORRCMD
  (CGI NOPANEL MOE)
  60 number
  cmdlabel: cmdtext
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORRCMD</td>
<td>CC</td>
</tr>
</tbody>
</table>

Command Description

The CORRCMD stage processes a command. In addition, CORRCMD inserts appropriate correlation wait and termination condition stages to gather data from the command being processed. Included in the inserted stages is CORRWAIT.

If you use the label syntax, CORRCMD automatically synchronizes the output of the command at the destination. This eliminates the need for “PIPE in PIPE” structures in many cases.

If you use the RMTCMD alternative remote processing label syntax and the target command is not PIPE, CORRCMD forces the output to be one multiline message for efficient cross-domain transfer. You can avoid this collection by coding a PIPE command as the target command of the label syntax.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

When specified as a first stage, CORRCMD terminates when it finishes processing its output. As a subsequent stage, CORRCMD terminates when its input stream is disconnected.

Operand Descriptions

- **cmdlabel:**
  Specifies a valid label. All NetView commands can be prefixed with a label. For more information on using labels with NetView commands refer to the [Tivoli NetView for z/OS User's Guide](#).

- **cmdtext:**
  Specifies the command to be processed. If `cmdtext` consists of a label only and
PIPE CORRCMD

CORRCMD is not the first stage, then CORRCMD will route commands in the input stream as specified by the label. Such commands cannot contain a form feed character ('OCX) unless the command being routed is PIPE.

Note: A PIPE command can contain any character. If your command contains the form feed character ('OCX), you should embed the command in a PIPE. For example,

```
yourlabel: PIPE CORRCMD cmd with possible ff chars | CONS ONLY
```

**CGI**

Use the CGI option for a command that is able to produce either a 3270 display or HTML, to inform the command that HTML is preferred. The direct effect of the CGI option is on the REXX function, CGI(), and causes the function to return a value of 1. CGI cannot be specified with ECHO.

**ECHO**

When ECHO is specified, the text of the command itself is written to the pipeline before the command is executed. ECHO cannot be specified with CGI.

**MOE**

Message on error (MOE) examines the return code from the command. If the return code is not zero, it inserts message DWO369I containing the return code into the stream after any messages the command could have returned. If you do not specify MOE, return codes from commands are ignored.

**NOPANEL**

When NOPANEL is specified, the command will not be allowed to display a full-screen panel. If it attempts to do so, message BNH113W will be inserted into the pipeline and the command will receive an I/O error code from NetView presentation services.

**number**

Optional first argument is a timeout override.

If the first token following CORRCMD is numeric, it is interpreted as a new specification of timeout which will override the value specified for the command by CCDEF. Valid values are 1–10000000. The default is 60, unless another value was defined with CCDEF.

**Usage Notes**

- The timeouts and termination conditions to be used are defined by the customer using the CCDEF command.
- When RMTCMD or service point commands are entered using CORRCMD, termination conditions are inherent in the command. Other termination conditions defined by CCDEF are redundant.
- When the target command (following the RMTCMD label syntax) is PIPE, the CORRCMD stage does not force output to be collected into one multiline message. You could add a COLLECT stage to your pipeline specification for better performance.
- If commands are to be issued by RMTCMD to another NetView program, both the local and the remote NetView domains must be at the V2R4 or later level. When you use the RMTCMD label syntax to transfer commands to a NetView V2R3 domain, the target command must be PIPE. However, you can use the pipeline to run other commands. See “PIPE NETVIEW” on page 137 for a list of some commands for which command and response correlation is supported.
Return Codes

The following return codes are valid only when the MOE operand is used.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>Installation exit 03 generated USERDROP.</td>
</tr>
<tr>
<td>-500 to -599</td>
<td>Failure attempting to call installation exit 03. Report specific return code to the Tivoli Customer Support.</td>
</tr>
<tr>
<td>-108</td>
<td>Command is type=I or type=P.</td>
</tr>
<tr>
<td>-112</td>
<td>Command search failed. This is usually because the command is too long.</td>
</tr>
<tr>
<td>-116</td>
<td>ACCESS not authorized. Command authorization restrictions prevent processing.</td>
</tr>
<tr>
<td>-120</td>
<td>Command is type=D.</td>
</tr>
</tbody>
</table>

There are other possible return codes indicating storage failure. The code you get depends upon the processing phase when storage failure was detected. Look for DSI124I at the system console for this condition.

Examples

Example: Issuing a VTAM Command
To issue the VTAM command D NET,APPLS, trap the resulting messages, and display them, enter:

```
PIPE CORRCMD D NET,APPLS | CONSOLE
```

Example: Issuing a Command to the Remote Domain
To transfer a VARY command to a remote domain (by the RMTCMD command) and to arrange the timer and termination condition stages at both the local and remote domains, enter:

```
PIPE CORRCMD CNM18: VARY NET,ACT,ID=XYZ | CONSOLE ONLY
```

Response
- At the local domain:
  - An appropriate wait time for the RMTCMD command is arranged.
- At the remote domain (CNM18):
  - The VARY command is run.
  - An appropriate wait time for the response to the command is arranged.
  - The wait is terminated when the expected response is received.
  - Output from the command is collected into a single multiline message for cross-domain transfer.
- At the local domain:
  - The local wait is terminated when the response is received.
  - Results are displayed on the screen.

Example: Issuing Multiple Commands at a Remote Domain
By coding a label as the only argument of CORRCMD you can execute multiple commands at a remote domain. Commands are passed on the input stream. For example the following commands will execute each command stored in the stem SOMECMDS. and store all the responses from the commands in the RESPONSES. stem:
Example: Routing Commands and Data to a Remote Domain through an Autotask

In this example the data contained in the stem MYDATA is sent to the remote domain CNM02 where it will be used by the USEDAT command. The remote command conversation is also shared with others by using the RMT02 autotask. The command, with the data, will be routed first to the autotask and then to the remote domain.

'PIPE (NAME DBL_HOP)',
'  STEM MYDATA',
'  CC /RMT02: CNM02: USEDAT',
'  STEM RESPONSES'

Output from USEDAT is routed back through the same path used to send the command.
PIPE CORRWAIT

Syntax

CORRWAIT:

Syntax Diagram:

1
interval

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORRWAIT</td>
<td>CORR, WAIT</td>
</tr>
</tbody>
</table>

Command Description

The CORRWAIT stage command allows time to pass while asynchronous messages generated by the previous command stage are returning to the pipeline. For every message processed by CORRWAIT, the timeout value is reset to allow up to \( n \) seconds between messages where \( n \) is determined by the interval parameter.

Asynchronous or delayed messages are those which return to the NetView program from commands running in another application, such as MVS or VTAM, or commands running at another NetView. When asynchronous responses are expected from any command issued in a pipe NETVIEW or VTAM stage, the next stage command must be CORRWAIT. Otherwise, the stage commands in between do not see the asynchronous messages, and results are unpredictable. The CORRWAIT stage command can be used anywhere in a pipeline.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

CORRWAIT normally terminates when the preceding command stage completes. For example, if you enter:

PIE NETV RMTCMD LU=CNM02,LIST XXX|CORRWAIT 60|STEM RMTDATA.

RMTCMD completes its local processing in less than a second, but processing of the command continues, first at a DST, next in the network, then at a remote NetView. LIST produces one or more messages, which CORRWAIT captures and writes to its primary output, STEM RMTDATA., and then CORRWAIT resets its timeout value.

When the LIST command completes, CORRWAIT is notified and the wait ends immediately. Functionally, it is as if the preceding stage did not complete.
processing until the command at the remote NetView completed. At that point, the
NetView stage disconnects and CORRWAIT ends with its input stream
disconnected.

CORRWAIT processing is the same when more than one command is issued by the
preceding command stage. For example, if you create a stem variable containing a
series of LIST DST= commands for all your DSTs and then issue:
PIPE STEM LISTCMDS.|NETVIEW|CORRWAIT 20| ...

Each LIST command queues a request for data to a DST and ends, but CORRWAIT
continues to wait until each of the queued requests completes.

Notification support to CORRWAIT is not available for VTAM and MVS
commands. Because this support is not available, these commands are considered
never-ending. CORRCMD and CCDEF can be used to handle VTAM and MVS
commands.

You can define a secondary output for CORRWAIT if you want CORRWAIT to
process in a different manner. CORRWAIT produces a message on its secondary
output stream each time a completion event occurs. When a secondary output
stream is defined, CORRWAIT continues to wait only while the secondary output
stream remains connected.

The messages passed to the secondary output stream consist of a plus or minus
sign followed by a ten character, left justified, zero padded completion code. This
completion code is followed, beginning in position 13, by information which might
be useful in debugging your pipeline. The debugging information includes:
• The domain ID where the event occurred
• TVBOPID where the event occurred
• The command completing, if applicable

Attention: Domain ID, TVBOPID, and the command completing are not intended
as a programming interface, because formats might change and additional data
might be included in the future. Use these only for problem determination.

The completion codes are based on the type of event, not the return code from a
command.

<table>
<thead>
<tr>
<th>Completion Code</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0000000000</td>
<td>A command thread completed (see note).</td>
</tr>
<tr>
<td>+0000000008</td>
<td>A timeout occurred.</td>
</tr>
<tr>
<td>+0000000012</td>
<td>A GO command was issued.</td>
</tr>
<tr>
<td>+0000000016</td>
<td>A task executing a thread terminated.</td>
</tr>
<tr>
<td>+0000000032</td>
<td>An ABEND occurred on a task that was executing a thread.</td>
</tr>
<tr>
<td>-0000000005</td>
<td>A RESET occurred on a remote task that was executing a thread.</td>
</tr>
</tbody>
</table>

Note: A command thread is any command executing under the control of the
pipeline or any secondary command created or queued by such a command.
When secondary commands create additional commands, these are not
counted individually, rather, the completion of the last secondary command
is reported as the completion of its parent secondary event.
Operand Descriptions

interval
Specifies the maximum time, in seconds, between messages before messages are no longer collected. Valid values are in the range of 1–10000000. The default is 1.

An asterisk (*) can be specified for interval. When an asterisk is specified, CORRWAIT will never time out. The following will end the wait:
- A GO command
- A RESET command
- A PIPEND pipe stage
- Secondary output disconnect
- Other conditions that end a wait

MOE
Message on error (MOE) inserts message DWO369I containing a return code into the stream when a timeout occurs, after any messages the command could have returned.

CORRWAIT recognizes an artificial timeout if the operator enters the GO command while the wait is in effect.

Usage Notes
- The display of \( \star \) in the upper-right of the operator’s screen indicates that CORRWAIT is actively waiting for messages.
- Another stage must follow CORRWAIT in order to actually wait for a specific interval of time.
- When routing a PIPE command to another domain (via RMTCMD), ensure that your CORRWAIT values are long enough. The system discards asynchronous, correlated messages that arrive after a CORRWAIT times out.
- When a terminating stage (TOSTRING or TAKE) is used to end a wait, the terminating stage must immediately follow CORRWAIT. This applies only to MVS or VTAM commands, because NetView commands automatically end CORRWAIT when the commands end.
- For performance considerations when issuing a command to MVS or VTAM, use a stage command containing terminating conditions (for example, TOSTRING or TAKE FIRST) after a CORRWAIT stage command. Terminating conditions end data streams early in the pipeline and allow the pipeline to end before the timeout period.

Return Codes
The following return codes are reported when the MOE operand is used.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>A timeout occurred (message interval exceeded).</td>
</tr>
<tr>
<td>12</td>
<td>A GO command was entered.</td>
</tr>
<tr>
<td>16</td>
<td>A task executing a thread terminated.</td>
</tr>
<tr>
<td>32</td>
<td>An ABEND occurred on a task executing a thread.</td>
</tr>
</tbody>
</table>

Examples

Example: Causing a Wait with CORRWAIT
To display We will now wait 9 seconds. and wait for nine seconds, enter:
Example: Using CORRWAIT to Wait for Messages
To issue the NetView LIST STATUS=OPS command for a logical unit (LU) named A157C9 in a remote domain, allow 60 seconds for each resulting asynchronous message to return to the pipeline and display the results, enter:

```
PIPE NETVIEW RMTCMD LU=A157C9,LIST STATUS=OPS
    CORRWAIT 60
    CONSOLE
```

Example: Terminating a CORRWAIT
An important consideration in using CORRWAIT with non-NetView commands, such as VTAM and MVS commands, is proper termination of the wait. Allowing a timeout to occur can result in lost messages. Instead, explicitly end the CORRWAIT with a following TOSTRING, TAKE FIRST, or GO command.

When the last expected message can be detected by a simple comparison or count, use TOSTRING or TAKE FIRST after the CORRWAIT stage. When more complicated conditions apply, use the GO command.

In this example, we expect two VTAM ACTIVE messages. We terminate immediately on receipt of IST0611 VARY ACT...FAILED message, but if good responses are received, we must count them.

Note: Certain VTAM message IDs are release dependent.

```
PIPE VTAM V NET,ACT,SCOPE=ALL,ID=NTFELN7E
    CORRWAIT 100
    TOSTRING 1.7 /IST0611/
    SAFE VTAMRESP
    LOCATE 1.7 /IST093I/
    DROP 1
    PIPEND 0
```

The desired termination condition is stop as soon as you receive IST061 (failure message) or when you receive the second IST093. Notice that the first VTAM ACTIVE message is dropped after being stored in a name SAFE for later examination. The second VTAM ACTIVE message that is received drives PIPEEND 0, which causes the pipeline to end with a 0 return code.

Still more complex termination decisions can require you to drive a command procedure from a NETVIEW stage following your CORRWAIT. This procedure would be able to examine the incoming messages one at a time and could create a named SAFE, if necessary. Like the DROP stage in this example, your procedure would produce a message to drive PIPEND when it is appropriate to end the wait.

Example: Using a Secondary Output Stream with CORRWAIT
CORRWAIT can be coded to terminated when one of the following occurs:
- 60 seconds has elapsed.
- The operator entered the GO command.
- A task terminates or ABENDS.

For CORRWAIT to process in this manner, include the following in your pipeline specification:

```
... |
A: CORRWAIT 60
...```
Assuming that the pipeline end character was defined as a %, the simple pipeline after the A: connector processes the secondary output stream from CORRWAIT 60. If a completion code other than +000000000 is processed, CORRWAIT terminates.
PIPE COUNT

Syntax

COUNT:

Syntax Diagram:

- COUNT
- MESSAGES
- BYTES
- EACHLINE
- EACHMSG
- LINES
- MAXLINE
- MINLINE
- FROM 0
- FROM number

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTES</td>
<td>BYTE, B</td>
</tr>
<tr>
<td>EACHLINE</td>
<td>EL</td>
</tr>
<tr>
<td>EACHMSG</td>
<td>EM</td>
</tr>
<tr>
<td>LINES</td>
<td>LINE, L</td>
</tr>
<tr>
<td>MAXLINE</td>
<td>MAXL</td>
</tr>
<tr>
<td>MESSAGES</td>
<td>MESSAGE, M</td>
</tr>
<tr>
<td>MINLINE</td>
<td>MINL</td>
</tr>
</tbody>
</table>

Command Description

The COUNT stage counts the number of messages, lines, or bytes received on its primary input stream and passes the count to its primary output stream when the input stream is disconnected.

For all keywords other than EACHLINE and EACHMSG, the original input stream is passed to the secondary output stream, if connected. Output to the secondary output stream is not delayed. A secondary output stream is not supported for COUNT EACHLINE and COUNT EACHMSG.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

COUNT terminates when the input stream or both output streams disconnect.
Operand Descriptions

BYTES
Specifies that the total bytes in all messages received on the input stream are to be passed to the primary output stream. Both leading and trailing blanks are counted.

EACHLINE
Specifies that COUNT set a line attribute in each message line indicating the line's position within the data stream. The modified line is passed to the primary output stream. A secondary output stream is not supported.

The line attribute can be accessed with EDIT LINECOUNT and CONSOLE DUMP.

The line attribute remains unchanged in the pipeline and in safes created by the SAFE stage. However, commands executed subsequently in the pipeline may change the line count attributes.

See "Example: Line Counts Reset by Command" on page 56 for information on modification of set line attributes, "PIPE EDIT" on page 68 for information on the EDIT stage, and "PIPE CONSOLE" on page 42 for information on the CONSOLE stage.

Note: EACHLINE should not be used if the processed messages are to be used with the REPLY command.

EACHMSG
Specifies that COUNT set a message attribute in each message indicating the message's position within the data stream. The modified message is passed to the primary output stream. A secondary output stream is not supported.

The message attribute remains unchanged in the pipeline and in safes created by the SAFE stage. However, message count attributes are reset if the message is processed by a NETVIEW or CORRCMD stage.

FROM
Specifies the initial value for the counter. The default is zero.

LINES
Specifies that the total number of lines in all messages received on the input stream is to be passed to the primary output stream.

MAXLINE
Specifies that COUNT should count each message line passed to it and return the number of bytes contained in the longest line. Both leading and trailing blanks are counted.

MESSAGES
Specifies that the total number of messages passed on the input stream is to be passed to the primary output stream.

Each multiline message is counted as one message.

MINLINE
Specifies that COUNT should count each message line passed to it and return the number of bytes contained in the shortest line. Both leading and trailing blanks are counted.

Usage Notes

• If you do not want leading and trailing blanks to be counted when using COUNT BYTES, add a STRIP stage before the COUNT BYTES stage.
**PIPE COUNT**

- Line count attributes set by EACHLINE are preserved, when calling a command with a message providing the target command, contains no processing that alters the current message.
- Message count attributes set by EACHMSG are not preserved when calling a command with the message.

**Examples**

**Example: Counting Messages**
The following example counts the number of messages copied by ASSIGN to the operators listed in the OPS. stem. REXX resolves the value for `total` in the PIPE for each iteration of the DO loop. The PIPE then adds the count of the next operator’s assigned messages to `total`. After the messages for all operators in OPS. stem are counted, the total count for all operators is returned in the SAY statement.

```rexx
/* REXX Fragment */
...

total = 0;
DO dex = 1 TO OPS.0
  PIPE NETV LIST ASSIGN=COPY OP='\OPS.dex,
     | LOCATE 1.7 /DSI637I/,
     | COUNT LINES FROM 'total,
     | VAR total'
END
SAY 'There are 'total 'messages copied to the operators.'
...
```

**Example: Line Counts Reset by Command**
The line count values set by COUNT EACHLINE can be reset by a command. Consider the following REXX CLIST:

```rexx
/* REXX Example - Echo safe to console */
'PIPE SAFE *',
  ' | CONSOLE'
EXIT
```

Now, if we name this CLIST ECHO and we call ECHO from the following pipeline, the line count attributes will be preserved.

```rexx
PIPE < INFILE
  | COUNT EACHLINE
  | NETVIEW /AUTO1: ECHO
  | WAIT 10
  | SAFE KEEPER
```

In this example a series of lines are read from INFILE. Each line is counted and the line attributes added to the lines before they are passed to ECHO. ECHO reads the current line passed from the PIPE through the SAFE stage and writes it as output in this correlated environment. The line returns from ECHO to the above pipe which is waiting for 10 seconds for the return from ECHO. The unchanged line is stored in the SAFE named KEEPER.

Now, if ECHO was changed as follows:

```rexx
/* REXX Example - Echo safe to console */
'PIPE SAFE *',
  ' | COUNT EACHLINE',
  ' | CONSOLE'
EXIT
```
Instead of returning the line unchanged to the invoking PIPE for storage in the KEEPER safe, the line attributes of each line are changed to a line count of 1. Each line from INFFILE is passed individually to ECHO, therefore each time COUNT EACHLINE is encountered in ECHO the line processed is always the first and is assigned a line attribute of 1.
Pipe CPDOMAIN

Syntax

CPDOMAIN:

| CPDOMAIN cpname |

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPDOMAIN</td>
<td>CPD</td>
</tr>
</tbody>
</table>

Command Description

The CPDOMAIN stage command converts control point (CP) names to NetView domain names. Input CP names can be network qualified. If the CP name is not network qualified, the local network name is used. CP names can be input as a parameter on the stage specification, or can be input to the stage. If more than one CP name is specified, they must be in separate messages. Note that only the first line of each message is examined.

If CPDOMAIN is specified as a first stage, cpname is required. If CPDOMAIN is not specified as a first stage, cpname is optional.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

CPDOMAIN terminates when the input stream or both output streams are disconnected.

Operand Descriptions

cpname

Specifies the name of the CP

Usage Notes

- The input CP name must be known to the local VTAM.
- Task DS16DST must be active on both the local node and target node.
- The program-to-program interface (PPI) must be active on the target node.
- CNMI=YES is specified in CNMSTYLE at both the local node and the target node.
- Because CPDOMAIN uses an asynchronous request, it must be followed by the CORRWAIT stage.
**Return Codes**

If a secondary output stream is connected, each nonzero return code is sent there as a signed, 10-digit decimal number with an indication of which CP name caused that return code. For example, if task DSI6DST is inactive, CPDOMAIN CPX would produce return code +0000000608 CPX.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0000000008</td>
<td>The input is not valid.</td>
</tr>
<tr>
<td>+000000060xx</td>
<td>DSIMQS to DSI6DST failed. The return code is 600 plus the DSIMQS return code.</td>
</tr>
</tbody>
</table>

**Examples**

**Example: Converting a Hardcoded CP Name**

The following example converts a CP name that is hardcoded:

```
PIPE (END ;) a: CPD USIBMNT.NTFEMVS | WAIT 10 | CONS;
  a: COLOR WHI | CONS
```

**Example: Converting a CP Name Specified In a Variable**

The following example converts a CP name that is specified in a variable:

```
PIPE (END ;) VAR var1 a: CPD | WAIT 10 | CONS;
  a: COLOR PIN | CONS
```
PIPE DELDUPES

Syntax

DELDUPES:

```
DELDUPES: PAD '00'X /char/ KEEPFIRST /char/ ALL position.length
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELDUPES</td>
<td>DELDUP</td>
</tr>
<tr>
<td>KEEPFIRST</td>
<td>KEEP1ST</td>
</tr>
<tr>
<td>'xx'X</td>
<td>X'xx'</td>
</tr>
</tbody>
</table>

Command Description

DELDUPES compares the first line of consecutive messages and deletes duplicates. Only consecutive duplicates are deleted. The duplicate messages are written to the secondary output stream if the secondary stream is connected.

To delete duplicate lines within a message, use SEPARATE prior to DELDUPES.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

DELDUPES terminates when the input stream or the output stream is disconnected.

Operand Descriptions

**ALL**

Specifies that all duplicate messages are deleted from the primary output stream and are written to the secondary output stream if the secondary stream is connected.

**KEEPFIRST**

Specifies that the first message in a sequence of duplicated messages is written to the output stream.

**KEEPLAST**

Specifies that the last message in a sequence of duplicated messages is written to the output stream.
PAD
Specifies the padding character to be used when comparing fields extending beyond the end of the data in the examined message.

PAD must be followed by a single delimited character or a single hex character.

The default PAD value is hex zero ('00'X).

position.length
The starting position and number of characters to be compared.

Position indicates the starting character within the line. Position can be any positive number.

Length is an unsigned positive number indicating the number of characters from position to be compared. An asterisk (*) can be specified for length indicating that all characters after position are to be used. Position without length and the period (.) separator will default length to 1.

If length is larger than the available characters, all available characters are used and the compared field will be padded with the value specified by PAD.

The default is to compare the entire message line (1.*).

Up to eight position.length pairs can be specified.

Usage Notes
DELDUPES delays the stream. Each message for the primary output stream is held by DELDUPES until a nonmatching message is found. When ALL or KEEPLAST is specified, each message for the secondary stream is delayed until another matching message is received. When KEEPFIRST is specified, messages for the secondary stream are not delayed.

Examples
Example: Display Last Logtime
The following displays a report of last time each person was logged:

```
PIPE < LOGTIMES
| SORT 1.18
| DELDUPES KEEPLAST 1.18
| CONSOLE
```

If LOGTIMES contains:

```
DOE, JOHN 98/02/18 13:25:04
SMITH, FRED 98/02/18 13:29:21
COLLINS, MARY 98/02/23 17:01:55
DOE, JOHN 98/02/23 09:00:00
HOWE, TOM 98/02/23 04:14:20
JONES, FRED 98/02/23 11:16:44
COLLINS, MARY 98/03/01 10:15:40
```

Then, the output to the console would show the latest entry for each person:

```
COLLINS, MARY 98/03/01 10:15:40
DOE, JOHN 98/02/23 09:00:00
HOWE, TOM 98/02/23 04:14:20
JONES, FRED 98/02/23 11:16:44
SMITH, FRED 98/02/18 13:29:21
```

In this example, the stable nature of SORT keeps records with the same sort field in their original order. Optionally, you can use another SORT to examine the date
fields and return the records to their original order. If sortable date fields were not included in the records, the same result can be achieved by adding COUNT EACHLINE before the SORT and EDIT LINECOUNT 1 prior to CONSOLE.
**Syntax**

```
DIVERT:
```

**Command Description**

The DIVERT stage writes messages received on its primary input to either its primary output or its secondary output, depending on what is received on its other input streams. When a message is available on the secondary input, the message from the primary input is written to the primary output. Otherwise, when a message is available on the tertiary input, the message from the primary input is written to the secondary output. Any message read from the secondary or tertiary input is discarded, unless COLLECT is specified.

**Streams**

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>3</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

**Termination Conditions**

DIVERT terminates when any of the following conditions occur:
- When its primary input becomes disconnected.
- When both its secondary and tertiary inputs become disconnected.
- When both its primary and secondary outputs become disconnected.

**Operand Descriptions**

**COLLECT**

Specifies that the trigger message received on the secondary or tertiary stream will be concatenated with the message being written to the output.

**Usage Notes**

DIVERT requires at least two input streams.

**Usage**

When DIVERT has only two input streams, it acts as a gating function by enabling the primary input to pass to the primary output as messages become available on the secondary input. In this case, no message is passed to the secondary output.

**Examples**

For example usage, refer to sample CNMSRPLY.
**Syntax**

```
DROP:
```

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINES</td>
<td>LINE</td>
</tr>
<tr>
<td>MSGS</td>
<td>MSG</td>
</tr>
</tbody>
</table>

**Command Description**

The DROP stage command enables you to specify how many messages or lines are to be discarded from the primary output stream. When the specified number of messages or lines are read from the input stream and discarded, all other messages or lines are copied to the primary output stream.

Discarded messages or lines are passed to the secondary output stream, if connected.

**Streams**

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

**Termination Conditions**

DROP terminates when the input stream or both output streams disconnect.

**Operand Descriptions**

- **count**
  - Specifies the number of messages or lines on the input stream to be discarded. Valid values are in the range of 1–10000000. The default is 1.

- **FIRST|LAST**
  - Specifies whether the messages or lines dropped are the first or last messages or lines in the stream. The default is FIRST.

- **LINES**
  - Specifies that `count` is the number of lines within the input stream to be dropped. If LINES is not specified, DROP will discard the number of messages indicated by `count`. 
**MSGS**

Specifies that *count* is the number of messages within the input stream to be dropped.

**Usage Notes**

DROP cannot be the first stage command.

**Examples**

**Example: Discarding Messages with DROP**

The pipe in the following example takes the output of the NetView LIST " command, deletes the END OF STATUS DISPLAY line using the DROP stage, collects the results into a multiline message, and displays them.

```
PIPE NETVIEW LIST "
  DROP LAST 1
  COLLECT
  CONSOLE
```

The DROP stage buffers one message so that it can determine which is last. It then discards this last message. Notice that the order is important. If COLLECT had preceded DROP, then DROP would have seen exactly one (multiline) message on its input stream. That entire message would have been dropped.
PIPE DUPLICAT

Syntax

DUPLICAT:

\[
\begin{array}{c}
\text{DUPLICAT} \\
\text{number} \\
\end{array}
\]

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUPLICAT</td>
<td>DUP</td>
</tr>
</tbody>
</table>

Command Description

DUPLICAT copies messages in the input stream and writes the copied messages to the output stream.

The copies are marked as "copy" (IFRAUCPY on) rather than "primary" (IFRAUPRI off). The message text is unchanged.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

DUPLICAT terminates when the input stream or the output stream disconnects.

Operand Descriptions

\(\text{number}\)

Specifies the number of copies to make in addition to the original message.

If \(\text{number}\) is zero (0), then the input message is written to the output stream and no additional copies are made. If \(\text{number}\) is −1, no additional copies are made and the input message is discarded from the pipeline.

\(Number\) must be −1 or greater. The default is 1.

\* An asterisk (*) indicates that DUPLICAT is to make copies indefinitely.

Examples

See also [Example: Discover TSO Stacks Serving a User] on page 205 for an example of how to use the asterisk (*) DUPLICAT option.
Example: Creating Four Consecutive Utilization Reports
In the following AUTO1 executes four TASKUTIL commands and returns the results to the console.

PIPE LITERAL /TASKUTIL/ | DUP 3 | CORRCMD /AUTO1: | CONSOLE
Syntax

EDIT:

Global Order:

COPY
COPYREST
COPYREV
FINDLINE n
FINDLINE string
FWDLINE n
LASTLINE
NAMEBIND /name/
NOEXPOSE
ONTO /string/
PAD / /
PAD /char/ hexstring
PARSE
READLINE
RESET
RESETAUTO
SETAUTO
SKIPTO /string/ number
TOPLINE
UPTO /string/ number
WRITELINE

Edit Phrase:
Chapter 2. Pipeline Stages and Syntax
PIPE EDIT

Conversion Order:

Output Order:

70  Tivoli NetView for z/OS Customization: Using Pipes
Synonyms

<table>
<thead>
<tr>
<th>Edit Order</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOTOKE</td>
<td>AUTOTOKE, IFRAUTOK</td>
</tr>
<tr>
<td>C2VG</td>
<td>C2VG</td>
</tr>
<tr>
<td>COLOR</td>
<td>COLOUR, LINEATTR, LINEATTRS</td>
</tr>
<tr>
<td>COPYREV</td>
<td>COPYREVERSE, REVERSECOPY, REVCOPY</td>
</tr>
<tr>
<td>DESC</td>
<td>IFRAUWDS</td>
</tr>
<tr>
<td>GV2C</td>
<td>VG2C</td>
</tr>
<tr>
<td>IFRAUHND</td>
<td>AUTOMATED</td>
</tr>
<tr>
<td>JOBNAME</td>
<td>JOBID, IFRAUWJA</td>
</tr>
<tr>
<td>JOBNUM</td>
<td>IFRAUWJU</td>
</tr>
<tr>
<td>LEFT</td>
<td>L</td>
</tr>
<tr>
<td>LASTLINE</td>
<td>LL</td>
</tr>
<tr>
<td>LINEORIGIN</td>
<td>LINEORIGN</td>
</tr>
<tr>
<td>MSGORIGIN</td>
<td>MSGORIGN</td>
</tr>
<tr>
<td>NEXT</td>
<td>N</td>
</tr>
<tr>
<td>NEXTLINE</td>
<td>NL</td>
</tr>
<tr>
<td>NEXTWORD</td>
<td>NW</td>
</tr>
<tr>
<td>READLINE</td>
<td>RL</td>
</tr>
<tr>
<td>RIGHT</td>
<td>R</td>
</tr>
<tr>
<td>ROUTECODES</td>
<td>ROUTCDE, IFRAUWRT</td>
</tr>
<tr>
<td>SYSCONID</td>
<td>IFRAUCON</td>
</tr>
<tr>
<td>TOP LINE</td>
<td>TL</td>
</tr>
<tr>
<td>WORD</td>
<td>TOKEN</td>
</tr>
<tr>
<td>WRTILELINE</td>
<td>WL</td>
</tr>
</tbody>
</table>

Command Description

The EDIT stage is an extremely powerful stage which enables you to make a wide variety of changes, or edits, to a message within a pipeline. Possible sources of the edit data include:

- Message data
- Line attributes
- Message attributes
- Literal data

With EDIT, messages can be created or reformatted in any desired fashion. In some cases modification of the message attributes and line attributes are also supported.

EDIT can be used to:

- Avoid creating a loop in REXX to manipulate messages.
- Improve performance. Editing within a pipeline flow is faster than driving a command to make the changes.
- Preserve message attributes while changing the message text.
- Improve programmer productivity when writing procedures to manipulate message data.
When used as a first stage, EDIT can be used to create multiline messages from literals.

Although appearing complex, EDIT is a simple stage consisting of global orders and edit phrases. Edit phrases define the action to be taken on the data flowing through the pipeline. Global orders define the overall environment for the subsequent edit phrases. Global orders are optional. Examples of global orders include:

- Defining padding characters.
- Defining how message data is parsed.
- Providing messages from the input stream to the edit phrase.
- Writing messages from the edit phrase to the output stream.

**Note:** Edit phrases operate on only one line at a time. Global orders control which line of a multiline message is processed by the edit phrase.

**Input Orders**
These orders define the source of data that will be processed by the conversion and output orders of the edit phrase. Examples of input orders include:

- Literal data
- Message attributes
- Line attributes
- All or part of the message data

**Conversion Orders**
These orders define how the data is to be manipulated. Conversion orders are optional. Examples of conversion orders include:

- Data conversion, such as from binary to character
- Date/time conversion
- Selecting a subset of the data

**Output Orders**
These orders define how the resulting data is to be placed in the output line and, subsequently, on the output data stream.

Together, global orders and edit phrases define an edit script. An edit script may be simple with one edit phrase, or a complex message processing program with hundreds of global orders and edit phrases. For an example of complex edit processing, refer to sample CNME2011 (SESMGET). All edit orders, new and old, are now accessible from the pipeline EDIT stage and from two automation table actions. To begin with simple examples, see “Example: Selecting a Word” on page 96 and “Example: Creating a Command” on page 96.

**Table 6. Global Order Summary**

<table>
<thead>
<tr>
<th>Global Order</th>
<th>Task Performed</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY</td>
<td>Copies one or more unread lines in a multiline message from input to output.</td>
<td>77</td>
</tr>
<tr>
<td>COPYREST</td>
<td>Copies all unread lines in a multiline message from input to output.</td>
<td>77</td>
</tr>
<tr>
<td>COPYREV</td>
<td>Copies one or more lines in a multiline message from input to output. The order of output is in descending order.</td>
<td>78</td>
</tr>
<tr>
<td>FINDLINE n</td>
<td>Changes the current line to the absolute line number indicated by the argument.</td>
<td>96</td>
</tr>
</tbody>
</table>
### Table 6. Global Order Summary (continued)

<table>
<thead>
<tr>
<th>Global Order</th>
<th>Task Performed</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINDLINE <em>string</em></td>
<td>Advances the current line to the line containing the specified target string.</td>
<td>78</td>
</tr>
<tr>
<td>FWDLINE <em>n</em></td>
<td>Moves the current line forward by the number specified.</td>
<td>78</td>
</tr>
<tr>
<td>LASTLINE</td>
<td>Resets the input to the last line of a multiline message.</td>
<td>78</td>
</tr>
<tr>
<td>NAMEBIND</td>
<td>Creates a name/value pair understandable by the alert and message adapters.</td>
<td>78</td>
</tr>
<tr>
<td>NOEXPOSE</td>
<td>Sets the IFRAUNEX automation flag.</td>
<td>78</td>
</tr>
<tr>
<td>ONTO</td>
<td>Redefines the logical end of the input line.</td>
<td>78</td>
</tr>
<tr>
<td>PAD</td>
<td>Defines the padding character to be used by all other orders.</td>
<td>78</td>
</tr>
<tr>
<td>PARSE</td>
<td>Defines how the WORD input order will count words.</td>
<td>78</td>
</tr>
<tr>
<td>READLINE</td>
<td>Provides the next line of a multiline message to the input orders.</td>
<td>80</td>
</tr>
<tr>
<td>RESET</td>
<td>Cancels all existing SKIPTO and UPTO orders.</td>
<td>80</td>
</tr>
<tr>
<td>RESETAUTO</td>
<td>Sets the INFRAUHND, IFRAUMTB, and IFRAUNEX automation flags to zero.</td>
<td>80</td>
</tr>
<tr>
<td>SETAUTO</td>
<td>Sets the IFRAUMTB automation flag to zero.</td>
<td>80</td>
</tr>
<tr>
<td>SKIPTO</td>
<td>Redefines the logical start of the input line.</td>
<td>81</td>
</tr>
<tr>
<td>TOPLINE</td>
<td>Resets the input to the first line of a multiline message.</td>
<td>82</td>
</tr>
<tr>
<td>UPTO</td>
<td>Redefines the logical end of the input line.</td>
<td>82</td>
</tr>
<tr>
<td>WRITELINE</td>
<td>Writes all text built by the output orders to the output message.</td>
<td>83</td>
</tr>
</tbody>
</table>

### Table 7. Input Order Summary

<table>
<thead>
<tr>
<th>Input Order</th>
<th>Task Performed</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIFR</td>
<td>Specifies that the input is the 256-byte AIFR body.</td>
<td>83</td>
</tr>
<tr>
<td>CURRGMT</td>
<td>Specifies an 8-byte store clock value generated at the time the order is executed.</td>
<td>83</td>
</tr>
<tr>
<td><em>hexstring</em></td>
<td>Specifies a hexadecimal string.</td>
<td>76</td>
</tr>
<tr>
<td>IFRAUHND</td>
<td>Use as input the automation action flag from the message.</td>
<td>83</td>
</tr>
<tr>
<td>IFRAUMTB</td>
<td>Use as input the automation submission flag from the message.</td>
<td>83</td>
</tr>
<tr>
<td>IFRAUNEX</td>
<td>Use as input the forbid exposure flag from the message.</td>
<td>83</td>
</tr>
<tr>
<td>IFRAUSRIB</td>
<td>Reads or sets a 2-byte user field.</td>
<td>84</td>
</tr>
<tr>
<td>IFRAUSRC</td>
<td>Reads or sets a 16-byte user field.</td>
<td>84</td>
</tr>
<tr>
<td>LEVEL</td>
<td>Specifies the data set concatenation level of the current line.</td>
<td>84</td>
</tr>
<tr>
<td><em>lineattr</em></td>
<td>Specifies that the input is one of the line attributes of the current line.</td>
<td>85</td>
</tr>
<tr>
<td>LINESENDER</td>
<td>Specifies the name of the sender.</td>
<td>85</td>
</tr>
</tbody>
</table>
Table 7. Input Order Summary (continued)

<table>
<thead>
<tr>
<th>Input Order</th>
<th>Task Performed</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgattr</td>
<td>Specifies that the input is one of the message attributes of the current message.</td>
<td>84</td>
</tr>
<tr>
<td>position.length</td>
<td>Specifies the subset of the input line to be processed. The subset is defined by specifying a starting character and the total number of characters.</td>
<td>77</td>
</tr>
<tr>
<td>SESSID</td>
<td>Specifies the TAF session ID for messages from TAF or SAF ID of messages received from the PPI.</td>
<td>85</td>
</tr>
<tr>
<td>/string/</td>
<td>Specifies a delimited character string.</td>
<td>77</td>
</tr>
<tr>
<td>WORD</td>
<td>Specifies the subset of the input line to be processed. The subset is defined by specifying a starting word and the total number of words.</td>
<td>85</td>
</tr>
</tbody>
</table>

Table 8. Conversion Order Summary

<table>
<thead>
<tr>
<th>Conversion Order</th>
<th>Task Performed</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B2C</td>
<td>Converts string of Boolean values to a character string.</td>
<td>88</td>
</tr>
<tr>
<td>C2B</td>
<td>Converts input to a string of Boolean values.</td>
<td>88</td>
</tr>
<tr>
<td>C2D</td>
<td>Converts input to a string representing a decimal number.</td>
<td>88</td>
</tr>
<tr>
<td>C2F</td>
<td>Converts input to a string representing a signed floating point number.</td>
<td>89</td>
</tr>
<tr>
<td>C2G</td>
<td>Converts fixed length string to double-byte (DBCS) string.</td>
<td>89</td>
</tr>
<tr>
<td>C2GV or C2VG</td>
<td>Converts varying length string to double-byte (DBCS) string.</td>
<td>89</td>
</tr>
<tr>
<td>C2P</td>
<td>Converts a packed-decimal number into a signed decimal.</td>
<td>89</td>
</tr>
<tr>
<td>C2S</td>
<td>Converts internal, floating point data into a 14-byte output string.</td>
<td>90</td>
</tr>
<tr>
<td>C2V</td>
<td>Converts a varying length string to a character string.</td>
<td>90</td>
</tr>
<tr>
<td>C2X</td>
<td>Converts input to a string representing its hexadecimal notation.</td>
<td>90</td>
</tr>
<tr>
<td>CNVDT</td>
<td>Converts the input from one date or time format to another. If the input cannot be converted, it is passed unchanged to the output.</td>
<td>91</td>
</tr>
<tr>
<td>CNVDT0</td>
<td>Converts the input from one date or time format to another. If the input cannot be converted, no data is output.</td>
<td>91</td>
</tr>
<tr>
<td>D2C</td>
<td>Converts a signed integer number into a full-word.</td>
<td>91</td>
</tr>
<tr>
<td>DT</td>
<td>Assumes the input text is a store clock (STCK) and converts the value to a readable 17-character string for the local time zone in the format MM/DD/YY HH:MM:SS.</td>
<td>91</td>
</tr>
<tr>
<td>DTS</td>
<td>Assumes the input text is a 17-character local time in the format MM/DD/YY HH:MM:SS and converts it to a store clock (STCK) value.</td>
<td>91</td>
</tr>
<tr>
<td>ETIME</td>
<td>Converts the store clock (STCK) to a decimal number indicating the elapsed time in microseconds since NetView startup.</td>
<td>91</td>
</tr>
</tbody>
</table>
### Table 8. Conversion Order Summary (continued)

<table>
<thead>
<tr>
<th>Conversion Order</th>
<th>Task Performed</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOUND</td>
<td>Translates a null string into No and any other string into Yes.</td>
<td>91</td>
</tr>
<tr>
<td>F2C</td>
<td>Converts a signed floating point number into a double-word.</td>
<td>91</td>
</tr>
<tr>
<td>G2C</td>
<td>Converts double-byte (DBCS) data to single-byte (SBCS) data.</td>
<td>91</td>
</tr>
<tr>
<td>GV2C or VG2C</td>
<td>Converts double-byte (DBCS) data into a varying length single-byte (SBCS) string.</td>
<td>92</td>
</tr>
<tr>
<td>LEFT</td>
<td>Truncates or pads the input to the length specified. Characters are counted from the beginning, or left, of the input.</td>
<td>92</td>
</tr>
<tr>
<td>ODDBYTES</td>
<td>Alternately, keeps and discards the input data.</td>
<td>92</td>
</tr>
<tr>
<td>OPDT</td>
<td>Assumes the input text is a store clock (STCK) and converts the value to a readable 17-character string representing the date and time in the format specified by the DEFAULTS command.</td>
<td>92</td>
</tr>
<tr>
<td>P2C</td>
<td>Converts a signed decimal number into packed-decimal.</td>
<td>92</td>
</tr>
<tr>
<td>RIGHT</td>
<td>Truncates or pads the input to the length specified. Characters are counted from the end, or right, of the input.</td>
<td>93</td>
</tr>
<tr>
<td>STRIP</td>
<td>Removes all padding characters from the beginning and end of the input.</td>
<td>93</td>
</tr>
<tr>
<td>STRIPL</td>
<td>Removes all padding characters from the beginning of the input.</td>
<td>93</td>
</tr>
<tr>
<td>STRIPR</td>
<td>Removes all padding characters from the end of the input.</td>
<td>93</td>
</tr>
<tr>
<td>SUBSTR</td>
<td>Selects a subset of the input data.</td>
<td>93</td>
</tr>
<tr>
<td>UPCASE</td>
<td>Translates the standard 26-character Latin letters to uppercase.</td>
<td>93</td>
</tr>
<tr>
<td>V2C</td>
<td>Converts input to a varying length string.</td>
<td>93</td>
</tr>
<tr>
<td>X2C</td>
<td>Converts character data to internal hexadecimal format.</td>
<td>93</td>
</tr>
<tr>
<td>YESNO</td>
<td>Converts a 1-byte field to the character string Yes or No.</td>
<td>93</td>
</tr>
<tr>
<td>ZDT</td>
<td>Assumes the input text is a store clock (STCK) and converts the value to a readable 17-character string for Greenwich mean time in the format MM/DD/YY HH:MM:SS.</td>
<td>94</td>
</tr>
</tbody>
</table>

### Table 9. Output Order Summary

<table>
<thead>
<tr>
<th>Output Order</th>
<th>Task Performed</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR</td>
<td>Sets presentation attributes for the output line.</td>
<td>94</td>
</tr>
<tr>
<td>LINETYPE</td>
<td>Defines the line type attribute of the output line.</td>
<td>94</td>
</tr>
<tr>
<td>NEXT</td>
<td>Specifies that the input is to be placed into the output without an intervening blank.</td>
<td>95</td>
</tr>
<tr>
<td>NEXTWORD</td>
<td>Specifies that the input is to be placed into the output with an intervening blank.</td>
<td>95</td>
</tr>
</tbody>
</table>
Table 9. Output Order Summary (continued)

<table>
<thead>
<tr>
<th>Output Order</th>
<th>Task Performed</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>Specifies that the data is to be placed in the output line beginning at the character indicated by position.</td>
<td>95</td>
</tr>
</tbody>
</table>

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

EDIT terminates when the input stream and the output streams are disconnected.

Operand Descriptions

Common Operands and Keywords

The following operands and keywords are common to a number of orders.

*hexstring*

Specifies a hexadecimal string. A *hexstring* can be in either of the following forms:

'nnnnnn'X
X'nnnnnn'

Where each *n* is a number 0–9 or character A–F. An even number of *n* values, up to 254, must be specified.

Before processing, the *hexstring* is converted to the corresponding character string.

When used as an input order operand, *hexstring* acts as a literal to be used as input.

*position.length*

The starting position and number of characters to be processed.

*Position* indicates the starting character within the line. By default, *position* is counted from the first character of the line. For input orders, the starting point for the count can be modified by the global orders.

*Position* can be any positive or negative number. A negative value for *position* indicates that the starting position is to be counted from the end of the line, rather than from the beginning.

*Length* is an unsigned positive number indicating the number of characters from *position* to be processed. An asterisk (*) can be specified for *length* indicating that all characters after *position* are to be used. *Position* without *length* and the period (.) separator will default *length* to 1.

If *length* is larger than the available characters, all available characters are used. The LEFT conversion order can be used to pad the resulting text if required.

Consider the following message:

**This...**

**Results in...**
PIPECANBEFUN!

+7.6 CAN BE

-7.* BE FUN!

9.20 N BE FUN!

8 A

-25.5 a null string is returned

-18.3 PI

number
An unsigned positive number. See the descriptions for the orders using number to determine the meaning of number.

/string/
A delimited character string. When used as an input order, /string/ defines a literal to be used as input. See the descriptions for other orders using /string/ to determine the meaning of /string/.

The plus sign (+), minus sign (−), and asterisk (*) have special meanings in edit scripts. To avoid confusion, these characters should not be used as /string/ delimiters in edit scripts.

Global Orders
Global orders specify common processing actions for all parts of the subsequent edit phrase.

COPY
COPY is used when the input is a multiline message. COPY copies one or more lines from the input to the output. Unlike COPYREST, COPY copies beginning with the current line, in ascending order.

One of the following parameters is required:

* Indicates all remaining lines in the input are to be copied.

number
A non-negative number indicating the number of lines to be copied.

COPY includes the current output line. If you want the first line of a multiline message to become the last line, you can code the following:

EDIT 1.* 1 /* get first line and output at position 1 */
READLINE /* Get next line */
COPY * /* copy lines 2 through end to output */
WRITELINE /* write first line */

The WRITELINE in this example is required because COPYREST cancels the implied WRITELINE created by the output order 1. See the description of WRITELINE on page 83 for more information.

COPYREST
COPYREST is used when the input is a multiline message. COPYREST copies all unread lines from the input to the output. It is the equivalent of coding the following for all input messages minus one:

READLINE
1.* 1
WRITELINE
COPYREST does not affect or write the current output line. If you want the first line of a multiline message to become the last line, you can code the following:

```
EDIT 1.* 1 /* get first line and output at position 1 */
COPYREST /* copy lines 2 through end to output */
WRITELINE /* write first line */
```

The WRITELINE in this example is required because COPYREST cancels the implied WRITELINE created by the output order 1. See the description of WRITELINE on page 83 for more information.

COPYREV
COPYREV is used when the input is a multiline message. COPYREV copies one or more lines from the input to the output. Unlike COPY, COPYREV begins with the current line, copies the number of lines requested, and displays the output in reverse order.

One of the following parameters is required:

- `*` Indicates all remaining lines in the input are to be copied.
- `number` A non-negative number indicating the number of lines to be copied.

COPY has an example. Do you want one for COPYREV?

FINDLINE `n`
Changes the current line to the absolute line number indicated by the argument. Note that FINDLINE 1 is equivalent to TOPLINE. If the number specified is greater than the number of lines in the current message, a null line is used for subsequent input orders.

If the number is negative, EDIT counts from the end of the message. For example, LINE -1 selects the last line of a multiline message and LINE -2 selects the next to last line. LINE 0 (zero) results in a null line.

FINDLINE `/string/`
Advances the current line forward to a line containing the specified target string, which then becomes the current line. If no line after the current line contains the target, a null line is used for subsequent input orders.

FWDLINE `n`
Moves the current line ahead by the number specified.

LASTLINE
Cancels all previous READLINE orders and performs a RESET. Input is set to the last line of the multiline message.

LASTLINE is a complementary function to TOPLINE. Where LASTLINE sets the input to the last line, TOPLINE sets the input to the first line of the multiline message.

See also TOPLINE on page 82.

NAMEBIND
NAMEBIND writes the value of the `/name/` and the text previously produced by other output orders, if any, to a new output line. Both the `/name/` and the output line data are preceded in the output line by a half-word length value. The message type (HDRMTYPE) and the line counter meet the name binding
requirements for events sent to the NetView Alert and Message Adapters. Lines created by NAMEBIND can be transferred to the adapter using the PPI stage. Examples of NAMEBIND can be found in "Example: Sending an Alert to the NetView Alert Adapter" on page 93, CNMEALUS, and CNMEMSUS. For more information about PPI, see "PIPE PPI" on page 154.

To create a valid adapter name/value pair binding, do the following:

- Copy the original contents of the automated alert or message to the output using a COPY * EDIT order.

  **Note:** This step is required for alert automation. Alerts can not be modified. For message automation messages can be modified using other EDIT orders prior to sending them to the output.

- Choose names and values consisting of displayable EBCDIC characters.

- Specify /name/ with a maximum 31 characters beginning with an alphabetic character.

  **Note:** The name specified in /name/ must also be defined in the Tivoli Event Adapter Profiles. IHSAACDS and IHSAMFMT are DSIPARM samples of adapter profiles.

- Choose names and values consisting of displayable EBCDIC characters.

  **Note:** Because NAMEBIND causes a line to be written, it cancels implicitly WRITELINE orders in effect.

**ONTO**
Sets the logical end of the line for input orders position.length and WORD to be a point other than the last character in the line.

/string/ Indicates that the input orders consider the line to end after the given /string/. Previous SKIPTO or UPTO orders are respected. ONTO is similar to UPTO except that the target string is included in the new logical line.

**NOEXPOSE**
Sets the IFRAUNEX automation flag. Subsequent routing of the message within a domain or cross-domain will not result in automation, message trapping, user exits, or logging.

**PAD**
Specifies the padding character to be used by subsequent orders. Examples of orders which use the padding character include the LEFT conversion order and the position output order.

PAD must be followed by one character value. This value can be specified as a delimited string, /char/, or as a hexstring.

The default PAD character is a blank.

**PARSE**
Specifies how the WORD input order counts words.

- **C** Indicates that all blank delimited tokens are counted as words.

- **Q** Indicates that tokens enclosed in single quotes are counted as words regardless of embedded blanks. The single quotes are removed from the parsed words. If the input data contains unbalanced quotes, only the data to the point where the error was discovered will be returned.

PARSE /string/
PARSE /string/ specifies how subsequent work/token orders will...
PIPE EDIT

count words. The characters in the specified string and only those characters are counted as token delimiters.

Note: A parse order always cancels the effect of any previous parse order.

For example, consider the following line:

'PIES ARE' 'REALLY' 'FUN'

If PARSE Q is specified, this line contains 3 words:
1. PIES ARE
2. REALLY
3. FUN

If PARSE C is specified, this line contains 4 words:
1. 'PIES'
2. 'ARE'
3. 'REALLY'
4. 'FUN'

The default PARSE value is C.

READLINE

READLINE is used with multiline input messages. READLINE makes the next line of the multiline message available to the input orders. In the following example, you issue MVS D A,L in your pipeline and want to retrieve only the time data and number of TS users from the resulting output.

The output from MVS D A,L would be similar to the following:

IEE114I 11.44.14 96.141 ACTIVITY 444
JOBS M/S TS USER SYBAS INITIS ACTIVE/MAX VTAM OAS
00000 00006 00001 00016 00002 00001/00300 00000

The following edit script would build a line containing the time data contained in the first line and the number of TS users from the third line:

WORD 2 1
READLINE
READLINE
WORD 3 NEXTWORD

The output from this edit script is 11.44.14 00001.

If required, READLINE performs a RESET.

Executing READLINE more times that the number of lines in the input message is not an error. If READLINE attempts to retrieve lines beyond the end of the message, a null line is passed to the input order.

RESET

Cancels all previous SKIPTO and UPTO orders. The original input line is made available to input orders specified subsequent to RESET.

RESETAUTO

Sets the INFRAUHND, IFRAUMTB, and IFRAUNEX automation flags to zero ('0'B). Take care not to create automation loops when using this function.
**SETAUTO**
Sets the IFRAUMTB automation flag to one ('1'). Subsequent routing of the message within a domain will not result in a submission to the automation table.

**SKIPTO**
Sets the logical start of the line for input orders position.length and WORD to be a point other than the first character in the line.

/\string/ Indicates that the input orders consider the line to start at the given /\string/. Previous SKIPTO or UPTO orders are respected. For example, if the current line is:

```
PIPES ARE FUN. PIPES USING EDIT ARE EVEN BETTER!
```

And, the edit script is:

```
SKIPTO /FUN/
SKIPTO /PIPES/
WORD 6
NEXT
```

The line is processed as follows:

```
PIPES ARE FUN. PIPES USING EDIT ARE EVEN BETTER!
```

And, the output will be:

```
BETTER!
```

\number\ Indicates that the input orders consider the line to start at character \number\. Number must be a positive number. Previous SKIPTO or UPTO orders are respected. For example, if the current line is:

```
PIPES ARE FUN. PIPES USING EDIT ARE EVEN BETTER!
```

And, the edit script is:

```
SKIPTO /USING/
SKIPTO 2
WORD 1
NEXT
```

The line is processed as follows:

```
PIPES ARE FUN. PIPES USING EDIT ARE EVEN BETTER!
```

And, the output will be:

```
SING
```
PIPE EDIT

If the /string/ is not in the input, or a number is specified which is larger than the length of the input, none of the input will be available to the input orders.

UPTO is a complementary function to SKIPTO. Where SKIPTO 1 returns the entire line, UPTO 1 returns none of the line.

See also UPTO on page 82 and RESET on page 81.

TOPLINE
Cancels all previous READLINE orders and performs a RESET. Input is set to the first line of the multiline message.

TOPLINE is a complementary function to LASTLINE. Where TOPLINE sets the input to the first line, LASTLINE sets the input to the last line of the multiline message.

See also LASTLINE on page 78.

UPTO
Sets the logical end of the line for input orders position.length and WORD to be a point other than the last character in the line.

/string/ Indicates that the input orders consider the line to end at the given /string/. Previous SKIPTO or UPTO orders are respected. For example, if the current line is:

PIPPES ARE FUN. PIPES USING EDIT ARE EVEN BETTER!

And, the edit script is:

UPTO /FUN/
WORD -1
NEXT

The line is processed as follows:

PIPPES ARE FUN. PIPES USING EDIT ARE EVEN BETTER!

UPTO /FUN/
WORD -1

And, the output will be:

ARE

number Indicates that the input orders consider the line to end at character number. Number must be an unsigned, positive number. Previous SKIPTO or UPTO orders are respected. For example, if the current line is:

PIPPES ARE FUN. PIPES USING EDIT ARE EVEN BETTER!

And, the edit script is:

UPTO 21
WORD -1
NEXT

The line is processed as follows:
Andrews the output is:

And, the output is:

If the /string/ is not in the input, or a number is specified which is larger than the length of the input, all of the input will be available to the input orders.

SKIPTO is a complementary function to UPTO. Where SKIPTO 1 returns the entire line, UPTO 1 returns none of the line.

See also SKIPTO on page 81 and RESET on page 80.

WRITELINE
Is used to build a multiline message. WRITELINE causes all text built so far by the output orders to be written to the output message. All text subsequently built by the output orders will be inserted as a new line in the multiline message.

Note: Output orders generate an implied WRITELINE at the end of the edit script unless WRITELINE is explicitly included. An implied WRITELINE remains in effect until an explicit WRITELINE or COPYREST is encountered.

Input Orders
Input order operands start an edit phrase. They define the data to be processed by the edit phrase. Possible sources for the data include:

- Literal text contained in the input order.
- Text received on the input data stream.
- Line attributes of the line received on the input data stream.
- Message attributes of the message received on the input data stream.

The orders, /string/, position.length, and hexstring can also be used as input orders.

AIFR
Specifies that the input is the 256-byte AIFR body. For additional information about the AIFR fields, refer to Tivoli NetView for z/OS Customization: Using Assembler.

Conversion orders, such as SUBSTR, can be used to obtain specific pieces of the AIFR.

Note: The position specified in SUBSTR must be the position described in Tivoli NetView for z/OS Customization: Using Assembler plus one (1).

CURRGMT
CURRGMT provides an 8-byte store clock value generated at the time the order is executed.

IFRAUHND
Use as input the automation action flag from the message. Bit value '1'B indicates the message matched a meaningful automation statement in the automation table. This is returned as the high-order bit in a 1-byte field.
IFRAUMTB
Use as input the automation submission flag from the message. Bit value '0'B indicates the message has not been submitted for automation. This is returned as the high-order bit in a 1-byte field.

IFRAUNEX
Use as input the forbid exposure flag from the message. Bit value '1'B indicates the message cannot be automated, trapped, or logged. This bit is set by output from CONSOLE ONLY and is returned as the high-order bit in a 1-byte field.

IFRAUSR
Reads or sets the 2-byte user field.

IFRAUSRC
IFRAUSRC reads or sets the 16-byte user field. The input order produces sixteen bytes; the output order is carried out only if the input available is exactly sixteen bytes.

LEVEL
Specifies that the input is the data set read by a previous < (from disk) stage containing the current line. The data set is indicated by the concatenation level of the data definition. The level is returned as a number preceded by a plus sign (+).

For example, if the following data sets are concatenated in this order under the DSIPARM DDNAME:
USER.INIT
USER2.INIT
NMPTLS.INIT
BNVE33E.PROCEED.DSIPARM
NETV.E120E.PROCEED.DSIPARM
NETV.E120E.PROCEED.CNMSAMP

And the current input line is contained in NMPTLS.INIT, the edit phrase input will be +3.

Notes:
1. If the data set is in-storage as a result of the INSTORE pipe stage, the LEVEL will be +0.
2. The EDIT stage containing LEVEL must be after a < (from disk) stage and can not have a NETVIEW or COUNT stage between < (from disk) and EDIT. The NETVIEW and COUNT stages reset the concatenation values.

\textit{lineattr}
Specifies that the edit phrase input is one of the line attributes of the current line processed by the edit phrase. Edit phrases operate on one line at a time. The \textit{lineattr} specifies attributes of the current line being processed by the edit phrase.

For example, the output from the following:
LINETYPE
NEXTWORD
READLINE
LINETYPE
NEXTWORD

Might be:
TC TD
Where TC was returned from the first line input and TD was returned from the second line. Lineattr can be one of the following:

**LINECOUNT**

LINECOUNT gets the line count from the current line as set by a previous COUNT EACHLINE, VET ROWS, or STEM (as a first stage). Any other source for LINECOUNT yields unpredictable results. See "PIPE COUNT" on page 54, "PIPE VET" on page 222, and "PIPE STEM and PIPE $STEM" on page 189 for more information.

LINECOUNT returns an EBCDIC number preceded by either a plus (+) or a minus (−) sign. This number is not padded unless the global order PAD /0/ is specified. If padded, LINECOUNT always returns a plus or minus sign followed by a 10-character number padded with leading zeros.

**LINEORIGIN**

Uses as input the domain ID where the line originated. This can differ for different lines in the same message. For example, if COLLECT is used to build the message with source messages from different domains, the message lines retain their origin domain IDs.

LINEORIGIN returns 8 characters.

**LINETYPE**

Produces as input 2 characters indicating whether the current line being processed is a control, label, data or end line. The lines returned are:

- TC: Current line is a control line.
- TL: Current line is a label line.
- TD: Current line is a data line.
- TE: Current line is an end line.

See also the LINETYPE output order.

**LINESENDER**

Specifies that the edit phrase input is the 8-character sender name of the current line.

`msgattr`

Specifies that the edit phrase input is one of the message attributes of the data received on the input data stream. For additional information on the attributes named, or with synonyms, beginning "IFRAU", refer to the assembler mapping of DSIIFR and the Tivoli NetView for z/OS Automation Guide.

`msgattr` can be one of the following:

- **ASID** Use as input the 2 hexadecimal character Address Space ID of the MVS originator of the message. If the message was not received from MVS, X’0000’ is returned.
  - Use the C2X conversion order to view or print this field.
  - ASID is synonymous with IFRAUWAS.

- **AUTOTOKEN** Use as input the 8-character MPF automation token.
  - AUTOTOKEN is synonymous with IFRAUTOK.

- **DESC** Use as input the 2-byte MVS Descriptor Code set by the originator of the message. If the message was not received from MVS, binary zeros are returned.
  - Use the C2B conversion order to view or print this field.
DESC is synonymous with IFRAUWDS.

**IFRAUTOK**
See AUTOTOKEN.

**IFRAUGMT**
Use as input the store clock value (STCK) at the time the message was created or received by NetView.

Use the OPDT or C2X conversion orders to view or print this field.

**IFRAUCON**
See SYSCONID.

**IFRAUCPY**
Use as input the copy flag from the message. This is returned as the high-order bit in a 1-byte field.

**IFRAUPRI**
Use as input the primary receiver flag from the message. This is returned as the high-order bit in a 1-byte field.

**IFRAUPPT**
Use as input the PPT origin flag from the message. This is returned as the high-order bit in a 1-byte field.

**IFRAUSDR**
Use as input the Task ID of the originator of the message.

**IFRAUSEC**
Use as input the secondary receiver flag from the message. This is returned as the high-order bit in a 1-byte field.

**IFRAUSRBB**
Use as input the 2-character User Field which may be set by various installation exits.

Usually the User Field contains binary data. Use the C2B conversion order to view or print this field if it contains binary data.

**IFRAUWAS**
See ASID.

**IFRAUWDS**
See DESC.

**IFRAUWJA**
See JOBNAME.

**IFRAUWJU**
See JOBNUM.

**IFRAUWRT**
See ROUTECODES.

**JOBNAME**
Use as input the 8-character JES job name of the originator of the message. JOBNAME will be 8 null characters if the message was not received from MVS.

JOBNAME is synonymous with IFRAUWJA.
JOBNUM
Use as input the 8-character JES job number of the originator of the message. JOBNUM will be 8 null characters if the message was not received from MVS.

JOBNUM is synonymous with IFRAUWJU.

MSGCOUNT
Use as input the results from a prior COUNT EACHMSG stage. MSGCOUNT should only be used if a stage preceding EDIT is COUNT EACHMSG. Any other source for MSGCOUNT yields unpredictable results. See "PIPE COUNT" on page 59 for more information.

MSGCOUNT returns an EBCDIC number preceded by either a plus (+) or a minus (−) sign. This number is not padded unless the global order PAD /0/ is specified. If padded, MSGCOUNT always returns a + or − sign followed by a 10-character number padded with leading zeros.

MSGORIGIN
Use as input the 8-character domain ID where the message originated.

ROUTECODES
Use as input the 16-character MVS route code data. If the message was not received from MVS, binary zeros are returned.
Use the C2B conversion order to view or print this field.
ROUTECODES is synonymous with IFRAUWRT.

SESSID
Specifies that the edit phrase input is the TAF session ID or SAF ID of the PPI sender.

SYSCONID
Use as input the 8-character MVS System Console ID. If the message was not received from MVS, blanks are returned.
SYSCONID is synonymous with IFRAUCON.

WORD
WORD is similar to position.length in that it specifies that a subset of the data received on the input data stream is used as input to EDIT. Unlike position.length, WORD counts blank delimited tokens or words within the input data. A word ends when a blank is encountered. The next word begins with the next nonblank character.

Startword.numwords must be specified.

Startword indicates the starting word within the current line. By default, startword is counted from the first word of the line.

Startword can be a positive or negative number. A negative value for startword indicates that the starting position is to be counted from the end of the current line, rather than from the beginning.

Numwords is an unsigned, positive number indicating the number of words from startword to be processed. An asterisk (*) can be specified for numwords indicating that all words after startword are to be used. Startword without numwords and the period (.) separator defaults numwords to 1.

If numwords is larger than the available words, all available words are used. The LEFT conversion order can be used to pad the resulting text if required.

Note: The PARSE global order can affect the way words are defined.
Consider the following message:

PIPES CAN BE FUN!

This ...

| WORD 1.* | Results in ...
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PIPES CAN BE FUN!</td>
<td></td>
</tr>
<tr>
<td>CAN BE</td>
<td></td>
</tr>
<tr>
<td>BE FUN!</td>
<td></td>
</tr>
<tr>
<td>CAN BE FUN!</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-25.5 a null string is returned</td>
<td></td>
</tr>
<tr>
<td>-6.3</td>
<td></td>
</tr>
</tbody>
</table>

### Conversion Orders

Conversion orders, if specified, must be in an edit phrase. That is, they must come after an input order and before an output order.

Multiple conversion orders can occur sequentially within the same edit phrase. Each subsequent conversion order operates on the results of the previous conversion order with the first conversion order operating on the text provided by the input order. Any number of sequential conversion orders can be included in a single edit phrase.

**B2C**

Specifies that the input data contains a text string. The text string is converted into its equivalent, internal, binary representation. For example, if the input was 1100000111000010 B2C would return AB.

The input data must be in exact multiples of eight characters. The converted data is one-eighth the length of the original.

B2C is the inverse of C2B.

**C2B**

Specifies that the input data is to be treated as a string of Boolean values. The input data is converted to a text string representing the individual bits. For example, if the input was AB, C2B would return 1100000111000010.

C2B is especially useful in converting bit string data such as that returned from DESC (IFRAUWDS) to a readable form.

Because C2B returns a character string 8 times longer than the original, you can easily generate a message which exceeds the 32 000 character limit for NetView messages. Use C2B to convert only the substring requiring conversion. See the SUBSTR conversion order on page 83 for more information. C2B is the inverse of B2C.

**C2D**

Specifies that the input data is to be treated as a two’s complement binary number. This input data will then be converted into a positive or a negative decimal number. For example, if the input was 1, C2D would return a result of -15. If the input was AB, C2D would return a result of -15934, as shown in the following example:

```
PIPE LIT /AB/
| EDIT 1.* C2D |
| CONS ONLY |
```
If the input is hexadecimal data and this data must be interpreted as a positive number, use PAD as the global order. The following example returns a result of 49602:

```plaintext
PIPE LIT /AB/
EDIT PAD '00'X 1.* RIGHT 3 C2D
CONS ONLY
```

C2D should be used with an input of 4 characters or less. The results of C2D are unpredictable with an input of more than 4 characters. Use C2D to convert only the substring requiring conversion.

C2D is the inverse of D2C.

**C2F**

Specifies that the input data is to be converted to a displayable floating point notation. The input can be a 2–8 byte floating point number. The converted value is a 22-byte, right justified, output string in the form \(-n.mmmmmE-dd\) where the exponent E-dd and the decimal point are only included if the converted number requires. When the exponent E-dd is not produced, the output is equivalent to packed decimal.

A maximum of 17 decimal digits are used in the conversion with leading and trailing zeros stripped. An 18th digit is calculated and used to round the results. For example, the repeating decimal number 1.9999999... is converted to 2.

See also the F2C conversion order on page 91 and the C2S conversion order on page 90.

**C2G**

Converts fixed-length strings to double-byte (DBCS) character strings by adding a shift-out character in front of the string and a shift-in after the string. C2G is the inverse of G2C.

**Note:** This conversion order is particularly useful when dealing with data interacting with the SQL pipe stage.

**C2GV or C2VG**

Converts varying-length strings to double-byte (DBCS) character strings by adding a shift-out character in front of the string and a shift-in after the string.

The input string must start with a 2-byte length field containing the number of DBCS characters. The number of data bytes after the length field must be twice the value of the length field since each DBCS character is represented by two bytes of data. The length field is not copied to the converted string.

C2GV is the inverse of GV2C.

**Note:** This conversion order is particularly useful when dealing with data interacting with the SQL pipe stage.

**C2P**

Specifies that the input data is to be converted to displayable floating point notation of a specified precision. Scale indicates the number of decimal digits retained.

- If \( \text{scale} \) is 0 (zero), the resulting number is an integer. For example, \( \text{C2P} 0 \) converts the input \( \text{X'123C'} \) to 123.
- If \( \text{scale} \) is positive, the resulting number will have \( \text{scale} \) number digits after the decimal point. For example, \( \text{C2P} 2 \) converts the input \( \text{X'123C'} \) to 123.45.
If scale is negative, the resulting number will have scale zeros added to the number. For example, \( C2P \, -1 \) converts the input X'123C' to 120.

\( C2P \) is the inverse of \( P2C \).

\( C2S \)

Specifies that the input data is to be converted to a displayable floating point notation. The input can be a 2–8 byte floating point number. The converted value is a 14-byte, right justified, output string in the form \(-n.mmmmE-dd\) where the exponent \( E-dd\) and the decimal point are included only if required by the converted number. When the exponent \( E-dd\) is not produced, the output is equivalent to packed decimal.

A maximum of 17 decimal digits are used in the conversion with leading and trailing zeros stripped. An 18th digit is calculated and used to round the results. For example, the repeating decimal number 1.9999999... is be converted to 2.

See also the \( F2C \) conversion order on page 51 and the \( C2F \) conversion order on page 84.

\( C2V \)

Specifies that the input data is a variable length string to be converted to a displayable string. The input data starts with a 2-byte, unsigned length value indicating the length of the string.

\( C2V \) is the inverse of \( V2C \).

Note: This conversion order is particularly useful when dealing with data interacting with the SQL pipe stage.

\( C2X \)

Specifies that the input data is to be converted to hexadecimal notation. For example, if the input was AB, \( C2X \) would return the EBCDIC hexadecimal value X'C1C2'. \( C2X \) is particularly useful when you need to display input data containing nondisplayable characters.

\( C2X \) is the inverse of \( X2C \).

\( CNVDT \)

Specifies that the input data is a date or time value and is to be changed to another date or time format. \( CNVDT \) must be followed by a parenthetical expression containing two entries. Each of these entries specifies a date or time format. The first entry is the format of the input data and the second is the format of the converted output.

Each entry can be the keyword DATE, the keyword TIME, or a string 4–8 characters in length. Specifying one entry for date and another for time is not supported.

DATE

Indicates the format is the date format specified by the DEFAULTS or OVERRIDE command.

TIME

Indicates the format is the time format specified by the DEFAULTS or OVERRIDE command.

\( from \) template or \( to \) template

Indicates that the conversion format is provided.
If a template is supplied, it must conform to the conditions specified for the date or time templates used with the DEFAULTS or OVERRIDE command. When using a template, both entries within the parenthetical expression must be for dates or times.

The input data is searched from the beginning of the data for a date format. For time conversion the input is searched from the end.

**CNVDT0**
Specifies that the same conversion is to be done as CNVDT. However, if the input data does not match the specified input format, no data is passed to the output.

**D2C**
Specifies that the input character string representing a signed or unsigned decimal number is to be converted to a 4-byte signed binary number. Use the RIGHT 2 conversion order to reduce the output to 2-bytes. For example, if the input was 49602, D2C returns AB.

Use D2C with an input resulting in 4 characters or less. The results of D2C are unpredictable with an input resulting in more than 4 characters. Use D2C to convert only the substring requiring conversion. See the SUBSTR conversion order on page 93 for more information.

D2C is the inverse of C2D.

**DT**
Specifies that the input data is a store clock (STCK) value, such as that obtained from the IFRAUGMT input order, and is to be converted to a 17-character date/time value. The data and time are for the local time zone and are in the converted to the form: MM/DD/YY HH:MM:SS.

To convert to Greenwich Mean Time, use ZDT.

**DTS**
Specifies that the input data is a 17-character local date/time value in the form MM/DD/YY HH:MM:SS, and is to be converted to a store clock (STCK) value which is based on Greenwich Mean Time.

**ETIME**
Specifies that the input data is a store clock (STCK) value and is to be converted to a decimal number representing the elapsed time in microseconds since NetView startup. The result is a decimal number that may be longer than 10 digits. The result may also be a negative number indicating that the message originated before NetView startup.

**FOUND**
FOUND is used after a SKIPTO or FINDLINE operation to translate a null string into No and any other string into Yes.

**F2C**
Specifies that the input character data represents a signed or unsigned floating point number and is to be converted to an 8-byte internal floating point representation. You can use the LEFT 4 conversion order to reduce the output to a short floating point internal number if desired.

F2C is the inverse of C2F.

**G2C**
Converts double-byte (DBCS) character strings to fixed-length strings by removing the shift-out character in front of the string and the shift-in after the string.
G2C is the inverse of C2G.

**Note:** This conversion order is particularly useful when dealing with data interacting with the SQL pipe stage.

**GV2C or VG2C**
Converts double-byte (DBCS) character strings to varying-length strings by removing the shift-out character in front of the string and the shift-in after the string. A 2-byte, unsigned length value precedes the converted string.

GV2C is the inverse of C2GV.

**Note:** This conversion order is particularly useful when dealing with data interacting with the SQL pipe stage.

**LEFT**
Specifies that the input data is to be truncated or padded to the length specified by `number`. Characters are counted from the beginning, or left, of the input. If padding is required, the character specified on the most recent PAD global order is used.

**ODDBYTES**
Specifies that the input text be alternately kept and discarded. The format of ODDBYTES is `ODDBYTES keep.discard`.

- `keep` is an unsigned, positive number specifying the number of characters to keep.
- `discard` is an unsigned, positive number specifying the number of characters to discard.

For example, if the input is 13:15:45 and ODDBYTES 2.1 is specified, 131545 is returned. That is, two characters were kept and one was discarded. Then, another two characters were kept, and one was discarded. And so on.

**OPDT**
Specifies that input text is to be treated as a store clock (STCK) value. OPDT converts the input into a 17-character string representing the date and time in readable form. The converted form is the one specified by DEFAULTS or OVERRIDE date and time formats for the system and operator where the conversion is done.

For example, X'ACE5652DC5690201' might be converted to 96/05/20 18:25:19. A typical use would be to take the input provided by the IFRAUGMT `msgattr` input order and convert it to readable form.

**Note:** The data representing a store clock is 8 characters in length. If the input data provided OPDT is not 8 characters, results are unpredictable.

**P2C**
Specifies that the input data is a character string representing a signed or unsigned floating point number. The input is converted to an internal packed decimal representation with `scale` decimal digits precision.

- If `scale` is 0, the integer portion of the number is converted to packed decimal. For example, C2P 0 converts the input 123.456 to X'123C' which is the packed decimal number representing 123.
- If `scale` is positive, the resulting number will have `scale` number digits included in the packed decimal. For example, C2P 2 converts the input 123.456 to X'12345C' which is the packed decimal representing 12345.
PIECE EDIT

- If \( scale \) is negative, the resulting number will have \( scale \) number of digits removed from the integer portion of the packed decimal. The decimal portion is ignored. For example, \( C2P -1 \) converts the input 123.456 to X'012C' which is the packed decimal number representing 12.

\( C2P \) is the inverse of \( P2C \).

**RIGHT**
Specifies that the input data is to be truncated or padded to the length specified by \( number \). Characters are counted from the end, or right, of the input. If padding is required, the character specified on the most recent PAD global order is used.

**STRIP**
Specifies that padding characters at the start or end of the data are to be removed. The padding character is defined by the most current PAD global order specification within the edit phrase.

**STRIPL**
Specifies that padding characters at the beginning of the data are to be removed. The padding character is defined by the most current PAD global order specification within the edit phrase.

**STRIPR**
Specifies that any padding characters at the end of the data are to be removed. The padding character is defined by the most current PAD global order specification within the edit phrase.

**SUBSTR**
Specifies that a subset of the input data is to be selected. \( Position.length \) indicates the starting position and length of data to be selected. For information about defining \( position.length \), see \( position.length \) on page 76.

If padding is required for the data to be the required \( length \), the characters specified by the most current PAD global order will be used.

**UPCASE**
UPCASE translates the standard 26-character Latin letters (as defined in code page 037) to uppercase.

**V2C**
Specifies that the input data is a displayable string and is to be converted to a variable length string prefixed with a 2-byte, unsigned length value.

\( V2C \) is the inverse of \( C2V \).

**Note:** This conversion order is particularly useful when dealing with data interacting with the SQL pipe stage.

**X2C**
Specifies that the input data is to be converted from displayable hexadecimal notation to internal binary representation. For example, if the input was X'C1C2', C2X would return the hexadecimal values X'AB'. The resulting hexadecimal value is half the length of the original.

\( X2C \) is the inverse of \( C2X \).

**YESNO**
Specifies that the first byte of the input data is to be converted from a bit-string to a value of Yes or No. If any bit in the first byte is a one (1), Yes is returned. If all bits in the first byte are zero (0), No is returned.
PIPE EDIT

Note: This conversion order is particularly useful when using the IFRAUCPY, IFRAUPPT, IFRAUPRI, and IFRAUSEC msgattr input orders.

ZDT
Specifies that the input data is a store clock (STCK) value, such as that obtained from the IFRAUGMT input order, and is to be converted to a 17-character date/time value. The date and time are for Greenwich mean time and are in the converted form: MM/DD/YY HH:MM:SS.

To convert to local time, use DT.

Output Orders
The output order ends the edit phrase and passes the resulting data to the output line. If number is negative, then EDIT counts from the end of the message.

COLOR
Specifies the presentation attributes of the resulting output line including color, highlighting, line type, and intensity. Multiple attributes must be enclosed in delimiters. Unknown attributes are ignored. Valid attributes are:

Color
CB  Blue
CR  Red
CP  Pink
CY  Yellow
CG  Green
CW  White
CT  Turquoise
CD  Default

Intensity
IN  Not intensified
IH  Intensified
ID  Output line is dark. Although the output line will be displayed, it will not be displayed.

Highlighting
HR  Reverse video
HU  Underlined
HB  Blinking
HD  Default

Line Type
TC  Control line
TL  Label line
TD  Data line
TE  End line

For example, if you want to create a blue, blinking data line of normal intensity, specify:
/CB IN HB TD/ COLOR

COLOUR, LINEATTR, and LINEATTRS are a synonyms for COLOR.

LINETYPE
Specifies the line type attribute of the resulting output line. The LINETYPE value is received from its input and must be one of the following:
TC  Output line is to be a control line.
TL  Output line is to be a label line.
TD  Output line is to be a data line.
Output line is to be an end line.

LINETYPE is not case-sensitive.

If the input to LINETYPE is not one of these four values, or if LINETYPE is not specified, the current line type attribute is retained.

**NEXT**
Specifies that the input to NEXT is to be inserted, without an intervening blank, into the output line after any text already in the output line.

**NEXTWORD**
Specifies that the input to NEXTWORD is to be inserted into the output line. If the output line already contains text, one blank is inserted into the output line prior to the data.

**position**
Specifies that the data be placed in the output line beginning at the character indicated by position. If position is larger than the current length of the output line, the existing output line is padded with the character defined by the PAD global order and the data added after the padding characters. If the output line created is already longer than position, the existing text beginning at position is overlaid.

For example consider the following message on the input stream to EDIT:
CAN BE FUN WITH EDIT!

With the following edit script:
/PIPES/ 1
1.* 7

PIPES is written to the output stream beginning at position 1. Then, the entire input stream is read using 1.* and written to the output stream beginning in position 7. The resulting output data is:
PIPES CAN BE FUN WITH EDIT!

Consider CAN BE FUN WITH EDIT! as the input stream to the following edit script:
PAD /*/
1.* 5

In this case the entire input stream is written to the output stream beginning at position 5. The first four positions are padded with asterisks (*) which was defined as the pad character. The resulting output data is:
****CAN BE FUN WITH EDIT!

Now consider the following edit script which receives CAN BE FUN WITH EDIT! on the input stream:
/MANIPULATING MESSAGES IS HARD/ 1
1.* 23

First MANIPULATING MESSAGES IS HARD is written to the output. Then, all the data received on the input stream is read by the input order 1.* and written to the output at position 23. Because MANIPULATING MESSAGES IS HARD is longer than 23 characters, the data read and the resulting output by 1.* 23 overlays the existing output data resulting in the following on the output stream:
MANIPULATING MESSAGES CAN BE FUN WITH EDIT!
**SETGMT**

SETGMT sets the IFRAUGMT value of the output message. The order is carried out only if the input available is exactly eight bytes.

**Usage Notes**

- Edit scripts consisting of many edit phrases can be difficult to read. It is recommended that you code one edit phrase or one global order on each source line. Together with appropriate commentary, your edit script will be as easy to understand. Refer to CNME2011 (SESMGET) for an example of this type of coding.

- When converting date and time values using CNVDT or CNVDT0, if the input data is longer than the specified input format, only a substring of the input data is compared and converted. The remainder remains as-is in the output.

  For example, the following converts the first 8 characters, the date portion, of the Greenwich mean time, to the date format specified by the DEFAULT command:

  ```
  PIPE ...|EDIT IFRAUGMT ZDT CNVDT ('MM/DD/YY' DATE) NEXT|...
  ```

  The last 9 characters remain in their original input format in the output.

  In date conversion, the first, or leading, characters of the input are converted. In time conversion, the last, or trailing characters of the input are converted. For example, the following converts the time portion of the GMT:

  ```
  PIPE ...|EDIT IFRAUGMT ZDT CNVDT ('HH:MM:SS' TIME) NEXT|...
  ```

**Examples**

Additional examples can be found in CNMS1101.

**Example: Selecting a Word**

The following edit script selects the fifth word in the input line and places it as the next entry in the output line:

```
WORD 5
NEXT
```

If the input line processed by this script is `DSI001I MESSAGE SENT TO NETOP2`, `NETOP2` is placed in the output line. If the output line currently contains text, `NETOP2` is added without an intervening blank.

**Example: Creating a Command**

In this example, the edit phrase changes the results from a `LIST STATUS=TASKS` command into commands to start all the reported resources. The LIST STATUS=TASKS command returns lines of the following format:

```
TYPE: OST TASKID: RESOURCE: A01A441 STATUS: NOT ACTIVE
```

Each LIST STATUS=TASKS line is processed by the following edit script:

```
/START TASK=/ 1
WORD 5
NEXT
```

/START TASK=/ is an input order. A single number can be either an input or output order. Because /START TASK= is the input order, the number 1 following /START TASK= must be an output order. So, START TASK= is written to the first position of the output line.
WORD 5 is also an input order. WORD requires a value, which in this case is 5. Because no global orders were specified for PARSE, parsing is done on blank delimited words. In the example line, the fifth blank delimited word, A01A441, is selected. The NEXT output order causes the selected word to be placed in the output line without an intervening blank.

So, the resulting output of this edit script on the example message would be:
START TASK=A01A441

If a NETVIEW stage follows the EDIT stage, this output is invoked as a command.

**Note:** Because status lines reported from LIST have a slightly variable format, it might be better to find the target word by counting from the end of the line, using a negative value for WORD, or by counting from a fixed word in the text. See the description of WORD on page 87 and SKIPTO on page 81 for more information.

**Example: Sending an Alert to the NetView Alert Adapter**
The following shows how to create a name/value pair. This name/value pair is bound with an automated alert and sent to the alert adapter. The command runs as a result of an automation statement containing TECROUTE.

```plaintext
/* construct a value from MYVAR and a unique identifier (GMT value */
/* set when alert received). Note: convert GMT to display-able chars */
'PIPE (NAME TECBIND)',
  'SAFE *',   /* copy complete automation alert into pipeline */
  'EDIT',     /* begin edit */
  'COPY *',   /* copy complete automation alert to EDIT output*/
  '/myvar'/ 1, /* start value one: variable value */
  'IFRAUGMT C2X NEXTWORD', /* add EBCDIC hex value */
  'NAMEBIND /EVENTID/', /* create output line for the TEC slot */
  '/MSUSEG('0000.31.30',3)'/ 1', /* start new line with text vec*/
  'NAMEBIND /ALERT31/', /* create another TEC slot */
  'PPI TECROUTE IHSATEC' /* transfer event to TEC */
/* Note use of virgule (/) to create a delimited string from the */
/* value of MSUSEG function assumes that no virgule (and no stage */
/* sep character) exists in the text. In actual practice, it would */
/* be wise to use non-printable characters for both delimiters. */
```
**Syntax**

```
ENVDATA:
```

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENVDATA</td>
<td>ENV</td>
</tr>
</tbody>
</table>

**Command Description**

The ENVDATA stage command outputs environment data, which consists of a multiline message in the following format:

- `SCREEN DEPTH` `nnn`
- `SCREEN WIDTH` `nnn`
- `COLOR COUNT` `n`
- `GENEALOGY command/name ...`

The data following the keyword `GENEALOGY` consists of blank delimited entries representing the REXX, PL/I, and C procedures in the calling sequence or procedure group which was active when ENVDATA was invoked.

Each entry consists of two names separated by a slash (/). The `command` is the command verb or synonym used to invoke the procedure. The `name` is one of the following:
- The module name if the procedure is PL/I or C
- The member name in DSICLD if the procedure is REXX.

Multiple entries following the `GENEALOGY` keyword show the calling sequence in reverse order. The command the operator entered is the last entry listed.

Currently, only four lines are produced. Additional data may be added in the future.

**Streams**

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

**Termination Conditions**

ENVDATA terminates when it finishes processing its output or when the output stream disconnects.
Usage Notes

- ENVDATA must be the first stage command.
- The numbers in output do not include leading zeros.

Examples

Example: Capturing Environment Data

The following example captures environment data in variables inside a REXX procedure.

Coded as a REXX example:

```plaintext
'PIPE ENVDATA',
'| NOT CHOP 13',
'| VAR ROWS COLS'
SAY ROWS
SAY COLS

--- 32
--- 80
```

Example: Capturing Genealogy Data

The following ENVDATA output shows the genealogy of a user-written command NDO:

SCREEN DEPTH 32
SCREEN WIDTH 80
COLOR COUNT 7
GENEALOGY HLLCMD/PLCMDMOD NDO/CNME9999

In this example, the NDO command was entered from an operator console. NDO is a CMDSYN of CNME9999, which is a REXX procedure. CNME9999 called HLLCMD as a command. HLLCMD resolves to the PL/I procedure PLCMDMOD. PLCMDMOD was the invoker of PIPE ENVDATA.
PIPE EXPOSE

**Syntax**

EXPOSE:

- RESPECT
- FORCE
- COMMAND
- NOLOG

**Command Description**

EXPOSE causes messages in the pipeline to be exposed and consists of the following actions:
- Passing a message to installation exit 02A
- TRAP (or &WAIT) processing
- Matching for action in automation table (message automation)
- Passing a message to installation exit 16
- ASSIGN COPY routing
- Logging to the network log, system log, or hardcopy log, as appropriate.

**Streams**

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

**Termination Conditions**

EXPOSE terminates when the input stream disconnects.

**Operand Descriptions**

**COMMAND**

Specifies that messages that are generated by the processing of a command in a previous CORRCMD, NETVIEW, MVS, or VTAM stage will be exposed before being absorbed into the pipeline. It is an error to specify COMMAND unless a previous CORRCMD, NETVIEW, or VTAM stage exists in the pipeline specification. If more than one previous CORRCMD, NETVIEW, or VTAM stage exists in the pipeline specification, then only the one nearest to the EXPOSE stage is affected.

Use of the COMMAND option allows a NETVIEW stage to successfully process command procedures that use TRAP MESSAGES (or &WAIT).

When COMMAND is specified, the effect of the EXPOSE stage occurs entirely before the messages are absorbed into the pipeline. There is no further action when the messages pass through EXPOSE at its position in the pipeline.

When COMMAND is specified, the command itself (command echo) is also exposed.

COMMAND implies RESPECT.
FORCE
Specifies that messages will be exposed to exit 02A, message automation, and exit 16 regardless of whether they have been previously exposed to those interfaces.

NOLOG
Specifies that messages are to be processed as indicated by other specified keywords, but no logging is to occur.

RESPECT
Specifies that messages that have already been exposed to exit 02A, message automation, or exit 16 will not be exposed to the same interfaces again. The default is RESPECT.

TOTRAP
Specifies that messages should only be exposed to TRAP processing. TOTRAP implies NOLOG.

Usage Notes
- EXPOSE cannot be a first stage command.
- Exposure can cause messages to be deleted. For example, if message automation matches the message and DISPLAY(NO) is the action, then that message is considered deleted and does not continue in the pipeline. Such deletion can affect the processing of CORRWAIT, TOSTRING, and TAKE that follow a CORRCMD, NETVIEW, MVS, or VTAM stage.
- Exposure causes null characters (X'00') in messages to be translated into blanks (X'40').
PIPE FANIN

Syntax

FANIN:

Command Description

The FANIN stage reads from multiple input streams. Unlike FANINANY, which reads multiple input streams simultaneously, FANIN reads from the first stream until that stream disconnects. FANIN then reads from the next input stream until it disconnects and so on. All data read by FANIN is passed to a single output stream.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>10</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

FANIN terminates when all input streams disconnect or the output stream disconnects.

Operand Descriptions

This stage command has no operands.

Usage Notes

- The primary input stream is the first to be processed. Additional input streams are delayed until processing is completed for the primary input stream.
- FANIN enables you to write output to a single stem variable from multiple places within a single pipeline.

Examples

Example: Process a List of Names

In this example, a list of names is read from a data set member. All individuals with the name SMITH are checked for the name TOMMY. All names containing SMITH and TOMMY will have TOMMY changed to TOM. Because the processing of records containing SMITH is the primary input to FANIN, all records containing SMITH are placed in the NAMES. stem, first.

Next, all records containing BAKER are truncated after 22 characters. The truncated records will be input to FANIN and subsequently placed in the NAMES. stem.

Finally, all other names that are not SMITH or BAKER will be processed by FANIN and placed in the NAMES. stem.
Unlike the FANIN ANY example, under "Example: Process a List of Names" on page 102, this example uses FANIN.
Syntax

FANINANY:

Command Description

The FANINANY stage reads from each connected input stream and passes the messages to a single output stream. Messages are passed in the order received without regard to their input stream. This is different from FANIN which passes all messages from a single input stream until it disconnects before passing messages from the next connected input stream.

For example, if FANINANY has two input streams, messages from both the primary and secondary input streams are passed to the output stream in the order received.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>10</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

FANINANY terminates when all input streams disconnect or the output stream disconnects.

Operand Descriptions

This stage command has no operands.

Usage Notes

- FANINANY delays the data stream flowing through the pipeline. If the output stream from FANINANY is used as an input stream to a second FANINANY, the messages passing through the first FANINANY will be delayed.
- To preserve the order of messages, code only one FANINANY stage in a complex pipeline.
- FANINANY enables you to write output to a single stem variable from multiple places within a single pipeline.

Examples

Example: Process a List of Names

In this example, a list of names is read from a data set member. All individuals with the name SMITH are checked for the name TOMMY. All names containing SMITH and TOMMY will have TOMMY changed to TOM. The modified names are input to FANINANY.
All records containing BAKER are truncated after 22 characters. The truncated records are input to FANINANY.

All other names, which are not SMITH or BAKER, are passed to FANINANY.

Unlike the FANIN example, under "Example: Process a List of Names" on page 102, in this FANINANY example, the names in the NAMES. stem will be in the same order as the records read from the data set.
PIPE FANOUT

Syntax

FANOUT:

Command Description

The FANOUT stage copies the messages received on its primary input stream to all connected output streams. Messages copied to the output streams are identical except:

- Messages written to the primary output stream retain the IFRAUPRI primary attribute of the original message. The copy attribute, defined by IFRAUCPY is set to 0.
- Messages written to all other output streams have the IFRAUPRI primary attribute setting set to 0 and the copy attribute, defined by IFRAUCPY set to 1.

For more information about IFRAUCPY and IFRAUPRI, refer to the Tivoli NetView for z/OS Customization: Using Assembler.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>10</td>
</tr>
</tbody>
</table>

Termination Conditions

FANOUT terminates when the input stream disconnects or all output streams disconnect.

Operand Descriptions

This stage command has no operands.

Usage Notes

- The messages are not written to the output streams in any predetermined order. Streams waiting on input from FANOUT may be delayed.
- If more than 10 output streams are required from FANOUT, one of the output streams can be used as an input stream to another FANOUT stage. The primary output stream from the second FANOUT stage should be passed to a HOLE stage because the primary output from the second FANOUT stage will not have a copy attribute.

Examples

Example: Driving Two Different Commands with the Same Message

In the following example, messages contained in the safe named MYSAFE are input to FANOUT. The primary output from FANOUT passes the message to
NETV MSGROUTE SAM HOLD(Y) to route the message to the operator SAM. A copy is passed by the secondary output stream to an EDIT stage that creates an ASEQLOG command that is passed to a NETVIEW stage. The ASEQLOG command logs the message to a sequential log.

For more information about ASEQLOG, refer to sample CNMS4275 and NetView for z/OS Customization: Using Assembler.

Additional commands can be driven off the same message by adding commands in stages following additional SEQLOG: connectors.

```
PIPE (NAME COPYNLOG END ¬)
| SAFE MYSAFE
| SEQLOG: FANOUT
| NETV MSGROUTE SAM HOLD(Y)
¬ SEQLOG:
| EDIT /ASEQLOG/ 1 1.* NEXTWORD
| NETVIEW
```
PIPE HELDMSG

Syntax

HELDMSG:

HELDMSG

Synonyms

Stage Command | Synonym
---------------|-------
HELDMSG        | HELD

Command Description

The HELDMSG stage command reads a copy of the held message queue of the task running the PIPE command into the pipeline. HELDMSG creates a snapshot of all the held messages at the instant this stage command runs. The messages remain in the held message queue for normal NetView processing. The held messages that are read into the pipeline are exact copies of the originals, including all time stamps and attributes.

Streams

Stream Type | Number Supported
-------------|-------------------
Input        | 0
Output       | 1

Termination Conditions

HELDMSG terminates when it finishes processing its output or when the output stream disconnects.

Usage Notes

- HELDMSG must be the first stage command.
- When you pull messages into the pipe with HELDMSG, those messages can potentially be logged a second time and held a second time. To redisplay messages without logging a second time, follow the HELDMSG stage with a CONSOLE ONLY stage.
- Held Message has a slightly different meaning at an autotask from that familiar for an attended operator. The purpose of holding a message at an autotask is to ensure proper routing of matching DOMs when they appear. Therefore a message is held at an autotask if and only if a DOM is expected for it. The message automation action HOLD(Y) is not meaningful at an autotask.

Examples

Example: Deleting Held Messages
To delete all the IEE123 messages on an operator’s screen, enter:
The HELDMSG stage copies all the operator’s held messages into the pipeline. The LOCATE stage passes only the IEE123 messages to its output stream. The CONSOLE DELETE stage issues a request to remove the held status on the operator’s screen for each message in its input stream.
PIPE HOLE

Syntax

HOLE:

HOLE

Synonyms

HOLE has no synonyms.

Command Description

The HOLE stage command discards the contents of the pipeline. Also, you can use it to determine whether a command has correlated output.

Use HOLE as a first stage to start a pipeline. Using HOLE in this way enables a stage that cannot normally be used as a first stage to start a pipeline by placing the stage immediately after HOLE.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

HOLE terminates when the input stream disconnects.

Operand Descriptions

This stage command has no operand descriptions.

Usage Notes

- Only commands that produce correlated output can be used effectively with the PIPE command. You can use the HOLE stage to determine whether a command produces correlated output. Code the command you want to test in a PIPE command and follow it with the HOLE stage:
  
  PIPE NETV your_command_name | HOLE
  
  If the command produces correlated output, the output from the command enters the pipeline and is discarded by the HOLE stage. If you do not see the usual output from the command, you know that the command produces correlated output and can be used effectively with the PIPE command.
- See "PIPE NETVIEW" on page 137 for a list of some commands for which command and response correlation is supported.

Examples

Example: Waiting Five Seconds

The following command waits five seconds.
The CORRWAIT stage does not insert any messages because there is no prior command to generate them, but CORRWAIT does not wait unless there is some stage connected to its output stream.

**Example: Discarding Pipeline Contents**
To run the LIST command, store the results in variables named MYVAR1, MYVAR2, and so forth, discard the pipeline contents, and add and display a text message, code the following instructions in a command list:

```
HOLETEST CLIST
&CONTROL ERR
* NETVIEW COMMAND LIST LANGUAGE
* PIPE NETVIEW LIST STATUS=TASKS +
  STEM MYVAR +
  HOLE +
  LITERAL ?HOLETEST IS RUNNING? +
  CONSOLE
* &EXIT
```
PIPE INSTORE

Syntax

```
INSTORE:  

INSTORE  ddname.membername
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLACE</td>
<td>REPL</td>
</tr>
<tr>
<td>NOREPLACE</td>
<td>NOREPL</td>
</tr>
</tbody>
</table>

Command Description

The INSTORE stage adds, deletes or replaces in-storage members. The members are then read from storage rather than the disk by DSIDKS disk services or any NetView process based on DSIDKS such as BROWSE or the < stage.

If INSTORE is the first stage, the named member is unloaded from storage and usage is reverted to reading it from disk. Otherwise, the input message lines are used to load the named member into storage. The member is not read from disk before loading. To read a member before loading it, place the < stage before INSTORE.

Note that the STRIP stage can save storage. For example, STRIP TRAILING results in the data being loaded without trailing blanks, which are then padded back to a minimum of 80 characters when the data is read. Null input message lines become blank lines when accessed. If no messages are input, the member is not found by DSIDKS FIND processes. Refer to the INSTORE examples section for coding samples.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Primary and secondary I/O streams can be defined. The primary input stream loads, unloads or replaces in-storage members and its output to the primary output stream is unchanged. A secondary input stream can be defined if the pipeline application requires the ability to monitor the data being read into INSTORE. If INSTORE detects a message on its secondary input stream, it terminates with return code 12 and the data is not loaded.

If a secondary output stream is defined, a signed, ten digit decimal return code is written to the secondary output.
Termination Conditions
INSTORE terminates when its function completes.

Operand Descriptions

ddname
Specifies the name of a standard NetView DDNAME, such as DSIPARM or DSICLD. Refer to the BROWSE command help for a list of valid DDNAMES.

membername
Specifies the name of the member being loaded or unloaded under the specified DDNAME.

COMMON
Specifies that the common in-storage member is loaded or unloaded.

LOCAL
Specifies that the loaded or unloaded member is local to this procedure family (LRCE group).

NOREPLACE
Specifies that the member is loaded only if it is not already in storage.

REPLACE
Specifies that the member, if already in storage, is to be replaced.

NOCRYPTO
Specifies that the member, when loaded into storage, is not encrypted.

CRYPTO
Specifies that the member, when loaded into storage, is to be encrypted such that its contents will not be visible in a storage dump. Determine the need for this against the CPU time to encrypt the data when this stage executes and decrypt the data when the member is read.

Usage Notes
When BROWSE is used to browse a member, the number of the data set containing the member, or included member, is displayed. If the number is zero, the member is an in-storage member.

The maximum supported line length is 255. Note that NetView record lengths for these datasets is typically 80. Use only larger lengths if you are sure the applications that read this member can handle them. Otherwise consider preceding this stage with CHOP 80. Records smaller than 80 are supported, but when DSIDKS returns them; they are padded out to 80 with blanks.

Security checking is done for the INSTORE stage, the LOCAL or COMMON keyword, and the member.

To prohibit loading member M1 in DSIPARM, code the following PROTECT statement:

PROTECT *.*.DSIPINS.*.DSIPARM/M1
To permit loading member M2 in DSICLD in the local procedure family but prohibit loading it in COMMON storage, code the following PROTECT statement:

```
PROTECT *.*.DSIPIINS.COMMON.DSICLD/M2
```

**Note:** Prohibiting LOCAL access to the DSIOPEN and CNMPNL1 DDs is not recommended.

### Examples

#### Example: Loading a Member from Disk into Storage, without Comments or Trailing Blanks

```
'PIPE < CNMPNL1.MEMXYZ',
  '  NLOCATE 1.1 /*/',
  '  STRIP TRAILING',
  '  INSTORE CNMPNL1.MEMXYZ COMMON REPLACE',
  '  APPEND NETV BR CNMPNL1.MEMXYZ'
```

**Attention:** NLOCATE 1.1 /*/ must be used with caution. An asterisk may not indicate a comment in every member and stripping these lines may invalidate the instore version.

#### Example: Unloading a Member from Storage (with a Return Code)

```
'PIPE {END ;} A: INSTORE CNMPNL1.MEMXYZ COMMON;',
  'A: | CONS'
```

#### Example: Hiding a Member within a Procedure Family

```
'PIPE HOLE|INSTORE DSICLD.C1 LOCAL|APPEND NETV BR DSICLD.C1|CONS'
```

#### Example: Managing Member Loading

Refer to CLIST CNME1054 (MEMSTORE).

#### Example: Ending the INSTORE if 'END' is Found at the Start of an Input Line

```
ARG MEM

'PIPE (END ;) < 'MEM, /* Read mem in */
  'A: NLOC 1.3 /END/', /* If not END line, send into INSTORE */
  'B: INSTORE DSIPARM.TEST7 COMMON REPLACE',/* Load in common */
  'HOLE', /* End pipe 1 */
  'A: ', /* END comes here... */
  'B: ', /* and back to INSTORE secondary-input*/
  'CONS' /* Output INSTORE's return code */
```
PIPE INTERPRT

Syntax

INTERPRT:

Stage Command Synonym

INTERPRT
INT

Command Description

INTERPRT builds stages from stage specifications that you supply to the PIPE command as its current message. You can use INTERPRT when creating your stage specifications dynamically, or when your PIPE command would otherwise be too long for the processing environment. For example, the NetView Command List Language restricts commands to 240 characters. If your pipeline specification would be longer than this, you can create a multiline message, each line of which is a complete stage specification. You provide this message to your PIPE command generally by using another, outer pipeline.

The stage command specifications coded should be considered to be complete stage specifications. The INTERPRT stage does not expect any stage separator or escape characters to be supplied, and any encountered is taken literally. However, it can be appropriate for the complete stage specification to contain a stage separator, or escape character within it, such as would be appropriate if a complete PIPE command was coded on the NETVIEW stage. See “PIPE (NCCF)” on page 21 for PIPE command options.

Often, the stage commands are read into the pipeline via the STEM stage. They are processed by the INTERPRT stage as an inner pipeline in a PIPE-within-a-PIPE structure.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

INTERPRT modifies the stages in the current message. Because it is a pipeline builder stage, INTERPRT does not have its own termination conditions. See the information on the stages INTERPRT is modifying for termination conditions.
Operand Descriptions

- Specifies that the current message is to be processed by INTERPRT. Asterisk (*) is the default.

Usage Notes

- INTERPRT can be used anywhere in the pipeline specification.
- Do not include an INTERPRT stage command among the stages being inserted by the INTERPRT stage command.
- All stage commands are assigned numbers based on their position in the pipeline specification. These numbers are shown in some error messages. Stage commands inserted by the INTERPRT stage command are assigned numbers according to the following formula:
  \[(pm) + ((10000)(sn))\]
  
  Where:

  \( pm \) Specifies the positions of the stage commands in the pipeline specification.
  \( sn \) Specifies the stage number of the INTERPRT stage command in the pipeline specification.

Examples

Example: Building Large Pipeline Specifications

Assume that the following variables are defined as input to the STEM stage of the pipeline specification below. OPER_CMD is a command of unknown size. Using the INTERPRT stage provides for the possibility that OPER_CMD can cause the pipeline specification to exceed the 240-character limit.

/* SAMPLE REXX COMMAND LIST */
X.0 = 5
X.1 = 'NETVIEW' OPER_CMD
X.2 = 'SEPARATE'
X.3 = 'TAKE 10'
X.4 = 'COLLECT'
X.5 = 'CONSOLE'
/* */
/* Collect the data records from the STEM X stage */
/* command (X is the name of the variables), and drive */
/* the INTERPRT stage command. */
/* */
'PIPE (NAME OUTER) STEM X.',
  'COLLECT',
  'NETVIEW PIPE (NAME INNER) INTERPRT *',
  'CONSOLE'
EXIT

During processing of the preceding command, the data records from the STEM stage command are formed into a multiline message. The NETVIEW stage uses the pipeline message it receives to drive the INTERPRT stage. The INTERPRT stage command interprets each of the data records in the message as a complete stage specification, then builds stage commands from the input data for the command. All such stages are substituted into the pipeline specification in place of the INTERPRT stage.

An equivalent pipeline created without using the INTERPRT stage is shown below. This pipeline is simpler but fails if the size of OPER_CMD forces the entire specification to become too large.
Example: Understanding Error Messages
Assume that the preceding example is changed so that the variable X.3 used to build the TAKE stage is misspelled as TAK. The following error messages are displayed.

DWO364E PIPELINE TERMINATED. NO STAGE TAK EXISTS.
DWO362E PIPELINE TERMINATED. ERROR IN STAGE 10003 IN PIPELINE "INNER"

The stage command number is:

\[3 + (10000)(1) = 10003\]

because TAKE is the third stage command inserted by the INTERPRT stage in the pipeline specification and INTERPRT is the first stage command in its (inner) pipeline specification.

Example: Running a Large Pipeline
In this example, there is a large pipeline specification saved in member LGPIPE of partitioned data set DSICLD. There should be one stage specification per record. To run the pipeline, enter:

```rexx
PIPE (NAME OUTPIPE) < DSICLD.LGPIPE
  | COLLECT
  | NETVIEW PIPE (NAME INPIPE) INTERPRT *
  | LOGTO HCYLOG
```

The stages are read from the member and collected. The NETVIEW stage makes this data the current message while running the command PIPE INTERPRT *. The INTERPRT stage reads the records and inserts them into the inner pipeline specification. The inner pipeline then runs. If there is any output from the inner pipeline from a CONSOLE stage, that output is trapped by the outer pipeline and passed to the next stage. In this case, the output is written to the hardcopy log.

You can add stages to the beginning or end of a pipeline specification that is to be interpreted. In the preceding example, you could replace PIPE (NAME INPIPE) INTERPRT * with this:

```rexx
PIPE (STAGESEP % NAME INPIPE) LITERAL /SOME INPUT/
  % INTERPRT *
  % COLLECT
  % CONSOLE
```

To the stages already defined in the member, you have added a first stage (LITERAL) and two end stages (COLLECT and CONSOLE). The definition of a new stage separator is necessary to avoid confusion with the stages of the outer pipeline, but has no effect on the stage specifications read from the member.

The new pipeline is shown below.

```rexx
PIPE (NAME OUTPIPE) < DSICLD.LGPIPE
  | COLLECT
  | NETVIEW PIPE (STAGESEP % NAME INPIPE) LITERAL /SOME INPUT/
    % INTERPRT *
    % COLLECT
    % CONSOLE
  | LOGTO HCYLOG
```
Syntax

IPLOG:

- IPLOG host port facility priority

Command Description

The IPLOG stage command sends a system log message to a remote host. The message is taken from the input stream.

Note: Most system log daemons only process single line messages. Unless you are sending to a daemon that supports multiline messages, such as the NetView system log daemon, ensure that single line messages are passed as input to IPLOG. This may require the SEPARATE stage command. See "PIPE SEPARATE" on page 174.

Operand Descriptions

host

Specifies the name of the remote host. It may be specified as a hostname or dotted IP address format.

port

Specifies the server port to use. In most cases, it should be 514.

facility

Specifies the source of the message. It accepts the same values as the facility portion of the -p operator command option.

priority

Specifies the message priority. It accepts the same values as the priority portion of the -p operator command option.

Examples

Example: Sending a System Log Message

```
PIPE LIT /My Message/
| IPLOG NMP119 512 USER ALERT
```
PIPE JOINCONT

Syntax

JOINCONT:

JOINCONT  TRAILING  LEADING  NOT

/string/

Synonyms

JOINCONT command has no synonyms.

Command Description

The PIPE JOINCONT stage joins consecutive messages in the stream when a match to a search string is found. A message is considered in its entirety and it could include blanks. For instance, an 80-byte record that is read from a file into the pipeline could contain trailing blanks or sequence numbers, which might cause a compare of a delimited string to the trailing part of the message to result in no match. You can use the STRIP stage to remove unwanted blanks or other characters before you use the JOINCONT stage.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

JOINCONT terminates when the input stream or the output stream disconnects.

Operand Descriptions

LEADING

Specifies that if there is a match to the comparison string at the beginning of the message, that message will be appended to the previous message.

NOT

Specifies that the absence of the specified strings will cause lines to be joined.

/string/

Specifies the character string as one of the following:

- Comparison string
- Substitution string

When one delimited string is specified, it is treated as a comparison string. If a message contains a match to the string, the string is removed from the message before it is joined.

When more than one delimited string is specified, the last such string is a substitution string while all others are comparison strings. When a message
PIPE JOINCONT

contains a match to a comparison string, in the appropriate leading or trailing position, the string that matches the search string is removed from the message and replaced by the substitution string between the messages being joined.

If multiple search strings are valid matches to the message, the longest matching string is replaced by the substitution string.

A line is considered a match, if any of the comparison strings are found in the appropriate leading or trailing position. A null string (/\/) is always a match.

At least one delimited string must be specified. You can specify /string/ p to 40 times.

The first nonblank character encountered after the keywords is the delimiter which establishes the boundary of the text string used by the stage. The delimited string ends when the same character is encountered a second time.

TRAILING
Specifies that if there is a match to the comparison string at the end of the message, the next message is appended to the message that contained the match. The default is TRAILING.

Usage Notes

- JOINCONT cannot be the first stage command.
- JOINCONT is used only with single-line messages. If the function is needed for multiline messages, use a SEPARATE stage preceding the JOINCONT stage.
- Processing a JOINCONT stage on messages, which are all multiline or a combination of single line and multiline, will yield unpredictable results.

Examples

Example: Joining Messages Ending in a '$'
For this example, you have established a file member named MYFILE in which some of the lines end with the character '$'. The file contains the following 80 byte records:

| YES |
| PIPE$ |
| LINES $ |
| ARE $ |
| GREAT |

Enter this command to eliminate each record’s trailing blanks, leave the YES message, join those that end in ‘$’ into a new message ‘PIPECINES ARE GREAT’, and write the results to the console:

```
PIPE < MYFILE
| STRIP TRAILING / /
| JOINCONT TRAILING /$/
| CONSOLE
```

Response

| YES |
| PIPELINES ARE GREAT |
Example: Joining Messages and Substituting a Character
Suppose you have established a file member named LETTERS in which you have some random vowels and consonants. The file contains the following 80 byte records:

A VOWEL
E VOWEL
I VOWEL
O VOWEL
U
M CONSONANT
T CONSONANT
X CONSONANT
Z

Enter this command to eliminate each record’s trailing blanks, join the records ending in VOWEL into a single message, substituting a comma for the word VOWEL, and join the records ending in CONSONANT into a single message substituting a comma for the word CONSONANT:

```
PIPE < LETTERS
| STRIP TRAILING / /
| JOINCONT TRAILING / VOWEL/ / CONSONANT/ /, / CONSOLE
```

Response

The output from the pipeline is in the form of two messages:

A, E, I, O, U
M, T, X, Z
PIPE KEEP

Syntax

```
KEEP keepname  (note)  APPEND SEIZE
   timeout   NOSPILL
```  

Command Description

The KEEP stage command enables you to define a task-global place to store messages. You can read from, or write to, storage defined by the `keepname`. The name and message are task global and endure beyond the life of the procedure that creates them. If PIPE KEEP is the first stage, it copies messages from the KEEP into the output stream. If PIPE KEEP is not the first stage, it copies messages from the input stream into the KEEP.

The PIPE KEEP stage command is similar to the PIPE SAFE stage command; PIPE KEEP enables you to define a task-global place to store messages and PIPE SAFE is a place to store one or more messages associated with a command procedure. For information about the PIPE SAFE stage command, see "PIPE SAFE" on page 170.

Operand Descriptions

**keepname**

A 1–8 character name of the KEEP. This name is case sensitive.

**timeout**

Specifies a timeout period. If no message is added for the specified period of time, the KEEP is freed. Timeout can be specified only when KEEP is not a first stage. The default value of the KEEP command is 1000 (16 minutes 40 seconds). The default timeout value of an existing KEEP command is the existing value. The maximum timeout value that can be specified is ten million seconds (nearly 4 months) or the value can be indicated by an asterisk, which means an indefinite timeout.

**Note:** When a message is input to the KEEP, the timeout period is refreshed, even if a new timeout is not specified.

**APPEND**

Used when KEEP is not a first stage. APPEND indicates that messages already stored should remain. By default, the KEEP is emptied if KEEP is not a first stage.

**SEIZE**

Used when KEEP is a first stage. It indicates that messages are taken, rather than copied from the KEEP. Use SEIZE to improve performance.

**SPILL**

Used when KEEP is not a first stage. When the KEEP expires, SPILL indicates to display messages in the KEEP. The messages are then subject to automation and message traps. If the KEEP expires because of a LOGOFF command or task termination, messages are routed to the authorized receiver.
NOSPILL

Used when KEEP is not a first stage. When the KEEP expires, NOSPILL indicates to discard messages in the KEEP.

Examples

Example: Accumulating Data

In the following examples, data is accumulated about a resource prior to an action:

IF MSGID='IST9501' & TEXT = . 'NCP44' . THEN
EXEC(CMD('PIPE SAFE * | KEEP NCP44 600 APPEND')) ROUTE(ONE NETOP1);
* Keep messages until no message has been received for ten minutes
* Note that all references to keep NCP44 must be invoked on the same opid

IF MSGID='IST6778A' & TEXT = . 'NCP44' . THEN
EXEC(CMD('CHKNCP NCP44'));
* Another message causes us to examine the history...

Where CHKNCP begins as follows:

```rexx
/* CHKNCP REXX */
parse arg keepName /* Keep names are case sensitive */
'PIPE (NAME CHKNCP1)',
'KEEP' keepName 'SEIZE', /* empty, not copy, the Keep */
'TAKE LAST 2', /* to examine PART of the history */
'EDIT IFRAUGMT OPDT 1', /* we need times of the two events*/
'STEM timeOfEvents.'
... examine times in stem var
```
PIPE LITERAL

Syntax

LITERAL:

---

LITERAL /string/

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>LITERAL</td>
<td>LIT</td>
</tr>
</tbody>
</table>

Command Description

The LITERAL stage command inserts a delimited text string into the pipeline.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

LITERAL terminates when the input stream or output stream is disconnected.

Operand Descriptions

/string/

Specifies the text.

The first nonblank character encountered after the stage name is the delimiter which establishes the boundary of the text string used by the stage. The delimited string ends when the same character is encountered a second time.

Usage Notes

- LITERAL can be used anywhere in the pipeline specification.
- If LITERAL is not the first stage command, messages remain in the pipeline and text added by this stage is inserted in front of the existing messages.

Examples

Example: Inserting Text Strings into a Pipeline

To display NetView Pipelines is powerful, enter:

```
PIPE LITERAL %NetView Pipelines is powerful% | CONSOLE
```
PIPE LOCATE

Syntax

LOCATE:

LOCATE ALL FIRST LAST (position.length) /string/

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE</td>
<td>LOC</td>
</tr>
</tbody>
</table>

Command Description

The LOCATE stage command selects messages that match a specified delimited character string to be passed to the primary output stream. Messages that do not contain the character string are passed to the secondary output stream, if connected.

LOCATE examines each input message for a match to the delimited string. A position and length pair can be supplied to limit the search to a particular column range.

If the delimited string is longer than the length specified on the LOCATE stage command, no matches occur, and no messages are passed to the primary output stream. Discarded messages are passed to the secondary output stream, if connected.

If the input message is a multiline message, all message lines are examined for the string specified. The entire multiline message is selected and passed to the primary output stream when any line of the message text matches the string specification.

A message is considered a match if any of the specified strings are located within it.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

LOCATE terminates when the input stream or both output streams are disconnected.
**PIPE LOCATE**

**Operand Descriptions**

**ALL|FIRST|LAST**

This keyword affects processing only for MLWTOs. Specifies whether comparisons are done on all, the first, or last line of multiline messages. The default is **ALL**.

**position.length**

Specifies the character position where searching begins in each message and the length of the search. If you specify a **length** of *, the remainder of the message is searched. If you do not specify a **position.length**, the entire message is searched.

**/string/**

Specifies the character string for which to search. The first nonblank character encountered after the stage name or **position.length** is the delimiter which establishes the boundary of the text string used by the stage. The delimited string ends when the same character is encountered a second time.

**Usage Notes**

- LOCATE cannot be the first stage command.
- You can specify the **position.length** and **/string/** pair up to 40 times.

**Examples**

**Example: Locating Messages by Content**

To issue the NetView command **LIST STATUS=TASKS**, trap the resulting messages, keep only the messages containing the phrase **NOT ACTIVE** in positions 55 through 64, and display them, enter:

```
PIPE NETVIEW LIST STATUS=TASKS
   LOCATE 55.10 /NOT ACTIVE/
CONSOLE
```
PIPE LOGTO

Syntax

LOGTO:

\[
\text{LOGTO:} \quad \text{ALL} \quad \text{HCYLOG} \quad \text{NETLOG} \quad \text{SYSLOG}
\]

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGTO</td>
<td>LOG</td>
</tr>
</tbody>
</table>

Command Description

The LOGTO stage command sends a copy of the contents of the pipeline to a specified log. The contents also remain in the pipeline for processing by the next stage.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

LOGTO terminates when the input stream is disconnected.

Operand Descriptions

ALL

Sends the message to all logs regardless of DEFAULTS or OVERRIDE command settings.

*  

Sends the message to the log or logs specified in the DEFAULTS or OVERRIDE commands. This is the default.

HCYLOG

Sends the message to the hardcopy log regardless of DEFAULTS or OVERRIDE command settings.

NETLOG

Sends the message to the network log regardless of DEFAULTS or OVERRIDE command settings.

SYSLOG

Sends the message to the system log regardless of DEFAULTS or OVERRIDE command settings.
PIPE LOGTO

Usage Notes

- LOGTO cannot be the first stage command.
- In the LOGTO stage command, do not specify * or ALL with any other keywords. You can specify any or all of the other logs in any combination.
- If you use NETLOG, SYSLOG, HCYLOG, or ALL, the settings for all logs in the DEFAULTS and OVERRIDE commands are ignored, not just the logs specified on the LOGTO stage.

Examples

**Example: Inserting a Text String and Logging It**
To insert a text string and log it in a NetView network log, enter:

```
PIPE LITERAL /TEST OF COMMAND XYZ STARTS HERE/ | LOGTO NETLOG
```
PIPE LOOKUP

Syntax

LOOKUP:

\[
\text{LOOKUP} \quad \text{detail_position\.length} \quad \text{reference_position\.length}
\]

APPEND  WILDCARD  /xyz/

Command Description

The LOOKUP stage compares two input streams and indicates on its output streams whether or not a match was found.

Data is read from the secondary input stream until it disconnects. The records read on the secondary input stream are called reference records. The data supplied on the primary input stream will be compared to these reference records.

After the secondary input stream disconnects, data is read from the primary input stream. The records read on the primary input stream are called detail records.

Each detail record is compared to the reference records. If the detail record matches a reference record, the input message containing the detail record is written to the primary output stream. Otherwise, it is written to the secondary output stream. The detail record data stream is not delayed.

If the detail record is a multiline message, only the first line of the message is compared to the reference records. If a match is found, the entire multiline message is written to the primary output stream. If a match is not found, the entire multiline message is written to the secondary output stream.

When all detail records have been processed and the primary input stream disconnects, LOOKUP writes all reference records that were not matched to any detail record to the third output stream.

One use for LOOKUP would be to compare a new data stream to an old data stream to determine the lines that were added, deleted, or unchanged.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>3</td>
</tr>
</tbody>
</table>

Termination Conditions

LOOKUP terminates when all input and output streams are disconnected.
**Operand Descriptions**

**APPEND**
Specifies that LOOKUP append the matching reference record to the detail record before writing the detail record to the primary output stream. If the detail record is a single line, the resulting output will be a multiline message consisting of the detail record and the reference record. If the detail record is a multiline message, the multiline message will be extended when the reference record is added to it.

*detail_position.length*
Specifies the starting position and length within the detail record to be compared to the reference records. The default is 1.*, which indicates that each detail record beginning at the first character for the entire length of the record will be compared to the reference records.

*Detail_position* can be any positive number.

*Length* can be any positive number or an *. Asterisk (*) indicates that all characters in the message beginning at the position indicated by *detail_position* should be compared. If *length* is not specified, *length* defaults to 1.

*reference_position.length*
Specifies the starting position and length of the reference records to be used in the comparison with each detail record. If *detail_position.length* is specified, the default for *reference_position.length* is the same value as the *detail_position.length* value.

*Reference_position* can be any positive number.

*Length* can be any positive number or an *. Asterisk (*) indicates that all characters in the message beginning at the position indicated by *reference_position* should be compared. If *length* is not specified, the *length* default is 1.

**WILDCARD**
The optional keyword **WILDCARD** must be followed by a delimited string of exactly three characters.

The first character, *x*, is the character contained in the reference record indicating that any single character found in the detail records at that point will be considered a match. A typical value for *x* is a question mark (?).

For example, if **WILDCARD */? /** is specified, and the reference record is ABCD?, then only detail record fields that are exactly 5 characters long with the first four characters being ABCD will be considered a match.

The second character, *y*, is the character contained in the reference record indicating that any number of characters found in the detail records at that point will be considered a match. A typical value for *y* is an asterisk (*).

For example, if **WILDCARD */?* /** is specified, and the reference record is ABCD*, then detail record fields at least 4 characters long beginning with ABCD will be considered a match.

The third character, *z*, indicates the exception to the *x* value. Any single character in the character position indicated by *x* will be considered a match unless that position contains the character indicated by *z*. Typically *z* is disabled by being specified as a blank.

For example, if **WILDCARD */?**/ is specified and the reference record is A?CD*, then the following will be considered a match:
**Usage Notes**

- **LOOKUP** requires two input streams and supports up to three output streams. If an output stream is not connected, the data that would have been passed on that output stream is discarded.
- **LOOKUP** must be able to read its secondary input until it disconnects. If it cannot read its secondary input, the pipeline will clog. For information about clogged pipelines, see "Chapter 8. Debugging NetView Pipelines" on page 297.
- When the `length` for the detail and reference records are different, and the WILDCARD option is not used, the characters specified for the reference record are searched for within the characters specified in the detail record. For example, if **LOOKUP** 1.6 5.2 is specified, two characters of each reference record beginning at character 5 will be compared to the first 6 characters in the detail record. Two consecutive characters within the first 6 characters of the detail record matching the 2 characters specified in the reference records cause a match. However, **LOOKUP** 1.2 6.4 searches for 4 characters of each reference record in 2 characters of the detail record.
- When the WILDCARD option is used, the pattern specified by the reference record is compared to the data in the detail records. For a match to occur, the data must match the pattern. All characters in the referenced data field are significant, including blanks. A wildcard pattern with trailing blanks implies that the detail record must have blanks in the same position. The pattern "AB*" is not the same as "AB+".
- Reference record line attributes such as color and line count are preserved. However, message attributes such as MLWTO structure and AUTOTOKE are not preserved.
- When **APPEND** is specified, **LOOKUP** appends both the data and line attributes of the reference record to the detail record.
- When the line attributes become inappropriately mixed within the output data stream, for example a label coming after a data line, insert a **COLLECT MAX 1** stage before writing the output data stream to the CONSOLE.
- Detail record line and message attributes are preserved in the output stream.
- Duplicate reference records should be avoided. **LOOKUP** processing cannot guarantee which of the duplicate records will be examined. Duplicate reference records should be particularly avoided if the **APPEND** keyword is specified since the record selected by **LOOKUP** to be appended is unknown.
- Reference records passed to the third output stream will not necessarily be in the same order as received on the secondary input stream.
Examples

Example: Comparing Values Contained in Two Stems
The following REXX fragment compares the stem old. to new. and creates three new stems:

**added.** Contains all the lines in new. which are not in old.

**deleted.** Contains all the lines which were in old. but are not in new.

**unchanged.** Contains all the lines which are in both old. and new.

/* REXX Fragment - Assume stems old. and new. already created */

```
PIPE (NAME LKUPXMP END %)',
'STEM new.',
'MIX: LOOKUP',
'STEM unchanged.', /* from new. that matched line from old. */
'MIX:', /* connect secondary output from lookup */
'STEM added.', /* from secondary--from new. unmatched */
'STEM old.', /* this STEM is a first stage! */
'MIX:', /* connect TWO streams of LOOKUP! */
'STEM deleted.' /* from old. with no match in new. */
```

Notes:
1. STEM old. immediately follows an end character making it the first stage of the subsequent simple pipeline.
2. The MIX: label between the STEM old. and STEM deleted. stages acts as a connector to the MIX: LOOKUP stage.
   - Being that MIX: has an input stream, MIX: connects it to the MIX: LOOKUP stage which defined the MIX: label. The input stream is connected as the next available input stream, which in this case is the secondary input stream.
   - Being that MIX: has an output stream, MIX: connects it to the MIX: LOOKUP stage which defined the MIX: label. The output stream is connected as the next available output stream, which in this case is the tertiary output stream.

So, the secondary input for LOOKUP is STEM old. and the tertiary output for LOOKUP is STEM deleted.

For a clearer understanding of the connections involved in this example, refer to the output generated after you change the second stage to MIX: (DEBUG) LOOKUP.

3. If you wanted STEM deleted. to contain line numbers from STEM old. instead of the data, you can add the following stage immediately before to the STEM deleted. stage:

```
EDIT LINECOUNT 1
```

**IDLEOFF Example**
Refer to CLIST CNME1057 (IDLEOFF) for an example of LOOKUP with WILDCARD. This example edits the detail records to facilitate wildcard pattern matching. The data is then edited to return the data to its original form.
PIPE MEMLIST

Syntax

```
MEMLIST
```

### Synonyms

Stage Command | Synonym
--- | ---
MEMLIST | MEML

### Command Description

The MEMLIST stage creates a list of members in one or more partitioned data sets (PDS) or data definitions (DD). For a DD, the members are listed for each data set in the concatenation. Members defined in an operator data set and members defined using INSTORE COMMON are also listed. For more information, refer to the OVERRIDE and PIPE INSTORE commands in the NetView online help.

The output is typically one multiline message for each input, one line for each member, as follows:

<table>
<thead>
<tr>
<th>Column</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–8</td>
<td>Member name</td>
</tr>
<tr>
<td>9</td>
<td>Blank</td>
</tr>
<tr>
<td>10</td>
<td>Relative data set number</td>
</tr>
</tbody>
</table>

For a PDS, the relative data set number is 1. For DD, the numbers match the concatenated data sets indicated by the LISTA command. For an operator data set member, the number is -1. For an INSTORE member, the number is 0 (zero).

### Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

### Termination Conditions

MEMLIST terminates when the input stream or both output streams are disconnected.

### Operand Descriptions

**(DD)**

Specifies that the specification is for a data definition.
PIPE MEMLIST

(DSN)
   Specifies that the specification is for a data set name.

dsndd
   The name of a DD or PDS. If an initial (DD) or (DSN) is not specified, the
   MEMLIST stage examines the specification. If the argument is a single 1-8
   character value without period delimiters or quotes, it is considered a DD.
   Otherwise, it is considered a DSN. The dsndd value may be specified on the
   stage, the input stream, or both.

Return Codes

If a secondary output stream is connected, each nonzero return code is sent as a
signed 10-digit decimal number, and includes the name of the data set that caused
the return code. For example, if XX does not exist, MEMLIST XX would produce
the following: '+0000000032 XX'.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>The user is not authorized to the data set.</td>
</tr>
<tr>
<td>28</td>
<td>The data set is unavailable. It may be in use by another user or task.</td>
</tr>
<tr>
<td>32</td>
<td>The data set does not exist.</td>
</tr>
<tr>
<td>69</td>
<td>Input or operand is not valid.</td>
</tr>
<tr>
<td>208</td>
<td>An RDJFCB error occurred. It may be a nonpartitioned DSORG.</td>
</tr>
<tr>
<td>212</td>
<td>An unspecified OPEN error occurred.</td>
</tr>
<tr>
<td>216</td>
<td>An unspecified READ error occurred.</td>
</tr>
<tr>
<td>300</td>
<td>A system problem occurred.</td>
</tr>
</tbody>
</table>

Examples

Example: Listing PDS Members
The following example illustrates how to list the members of a partitioned data set:
PIPE (END ;) a: MEML USER.INIT | CONS ONLY; a:| COLOR YEL | CONS

Example: Listing Members of Multiple DDs
The following example illustrates how to list members of multiple data definitions:
PIPE LIT /DSIPARM DSIVTAM/ | SPLIT | MEML | CONS ONLY
PIPE MVS

**Syntax**

```
MVS:
```

**Synonyms**

PIPE MVS has no synonyms.

**Command Description**

The MVS stage command specifies an MVS command to run. It can be used as a replacement to coding MVS commands within the NETVIEW stage. All rules that apply to an MVS command issued using the NETVIEW stage also apply here. See "PIPE NETVIEW" on page 137 for these rules.

**Streams**

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

**Termination Conditions**

The MVS stage terminates when it finishes processing its output or when the input stream is disconnected.

**Operand Descriptions**

*cmdtext*

Specifies the command.

If MVS is the first stage, `cmdtext` is required.

If MVS is not the first stage, `cmdtext` is optional. The command is run once for each message delivered by the previous stage. Every time the command runs, the input message becomes the current message during the processing.

**MOE**

Message on error (MOE) examines the return code from the command. If the return code is not zero, it inserts message DWO369I containing the return code into the stream after any messages the command could have returned. If you do not specify MOE, return codes from commands are ignored.

**Usage Notes**

Any restrictions that apply to running an MVS command under PIPE NETVIEW also applies to PIPE MVS.

**Note:** MVS system commands can be issued from the NetView program only if extended multiple console support (EMCS) consoles are being used.
Return Codes

The following return codes are valid only when the MOE operand is used.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>Installation exit 03 generated USERDROP.</td>
</tr>
<tr>
<td>-500 to -599</td>
<td>Failure attempting to call installation exit 03. Report specific return code to Tivoli Customer Support.</td>
</tr>
<tr>
<td>-108</td>
<td>Command is type=I or type=P.</td>
</tr>
<tr>
<td>-112</td>
<td>Command search failed. This is usually because the command is too long.</td>
</tr>
<tr>
<td>-116</td>
<td>ACCESS not authorized. Command authorization restrictions prevent processing.</td>
</tr>
<tr>
<td>-120</td>
<td>Command is type=D.</td>
</tr>
</tbody>
</table>

Note: The PIPE MVS stage could result in the return codes issued by the MVS command or various return codes indicating storage allocation failures. Message DSI124I will also be issued.

For information about the NetView MVS command, refer to the NetView online help facility.

For information about storage allocation failures, look for DSI124I at the system console. The code you receive depends upon the processing phase when storage failure was detected.

Examples

**Example: Issuing a Simple MVS Command**
To issue the command 'MVS D A,L', trap the resulting messages, and display them to the console, enter:

```
PIPE MVS D A,L
  CORRWAIT 10
  CONSOLE
```

**Example: Using a Message in the Pipeline to Trigger the MVS Stage**
To insert a literal string into the pipeline, use it to trigger the issuance of the command 'MVS D A,L', trap the resulting messages, and display them to the console, enter:

```
PIPE LITERAL /Issue a command to MVS/
  MVS D A,L
  CORRWAIT 10
  CONSOLE
```
PIPE NETVIEW

Syntax

NETVIEW: (CGI NOPANEL MOE) cmdtext

Synonyms

Stage Command       Synonym
NETVIEW             NETV

Command Description

The NETVIEW stage command specifies to run a NetView, MVS, or local VTAM command. The resulting messages are placed in the pipeline. The NETVIEW stage can be placed anywhere in the pipeline specification.

When NETVIEW is not a first stage, NETVIEW invokes a command once for each message on the input stream. Each time NETVIEW receives a message on its input stream, that message becomes the current message. The current message is the message to which NetView’s message information functions refer. Thus, GETMLINE, MSGORIGIN(), DSIGETDS, or other message-dependent commands and functions issued by the command invoked by the NETVIEW stage access the message that caused the command to be invoked, exactly as they would be if an automation table action or a MSGREAD operation had produced the current message. Also, NetView commands such as MSGROUTE that operate on messages have access to this current message.

Unlike many other stages, the NETVIEW stage does not require an output stream. This means that NETVIEW can be a last stage. Also, if a stage following NETVIEW were to be disconnected, NETVIEW would continue to process as long as it had an input stream.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

The NETVIEW stage terminates when it finishes processing its output or when the input stream disconnects.
Operand Descriptions

\textit{cmdtext}

Specifies the command.

If NETVIEW is the first stage, \textit{cmdtext} is required.

If NETVIEW is not the first stage, \textit{cmdtext} is optional. The command is run once for each message delivered by the previous stage. Every time the command runs, the input message becomes the current message during the processing.

If \textit{cmdtext} is specified, the first blank-delimited token is considered to be the command name. Any additional tokens are passed with the command and become arguments to it.

If \textit{cmdtext} is not specified, the NETVIEW stage extracts the first line of a message as the command and additional lines, if any, as data to be processed by that command. Any additional messages in the input stream would be treated the same way.

In all cases, the original message becomes the current message while the command runs and is then discarded.

If the command name is not a valid NetView command or if no command is found, a return code of 4 is generated and message DSI002I is inserted into the output stream.

\textbf{CGI}

Use the CGI option for a command that is able to produce either a 3270 display or HTML, to inform the command that HTML is preferred. The direct effect of the CGI option is on the REXX function, CGI(), and causes the function to return a value of 1. CGI cannot be specified with ECHO.

\textbf{ECHO}

When ECHO is specified, the text of the command itself is written to the pipeline before the command is executed. ECHO cannot be specified with CGI.

\textbf{MOE}

Message on error (MOE) examines the return code from the command. If the return code is not zero, it inserts message DWO369I containing the return code into the stream after any messages the command could have returned. If you do not specify MOE (message on error), return codes from commands are ignored.

\textbf{NOPANEL}

When NOPANEL is specified, the command will not be allowed to display a full-screen panel. If it attempts to do so, message BNH113W will be inserted into the pipeline and the command will receive an I/O error code from NetView presentation services.

NOPANEL has restrictions when used with the ATTACH command. Refer to the Tivoli NetView for z/OS Command Reference or the NetView online help for information about ATTACH.

\textbf{Usage Notes}

- It is important to distinguish between the output of a command and the results of the command. The NETVIEW stage causes the output of a command to be trapped within the pipeline, but the results, generally, are not. Output consists of
messages that are issued in the immediate environment. Results are messages or other actions that are caused by the command, but not immediately or not for the issuing environment.

For example, the MSG command causes two messages, one output and one result. The DSI039I MSG FROM ... is a result. It is not trapped in the pipeline, even if sent to the same operator where the MSG command was issued. The DSI001I MESSAGE SENT TO... message is output and is trapped by the CORR=CMD parameter to the DSIMQS invocation pipeline for further processing by subsequent stages. Also, read about the CORRWAIT stage; see "PIPE CORRWAIT" on page 48.

• The following are among those NetView commands supported:
  – Commands that run within the local VTAM.
    The VTAM stage command is useful for issuing VTAM commands.
  – User-written commands that use the CORR keyword on DSIPUSH, DSIFIND, or DSIMQS assembler macros. Refer to the Tivoli NetView for z/OS Customization: Using Assembler for more information about these macros.
  – MVS system commands issued from the NetView program, if extended multiple console support (EMCS) consoles are being used.

For more information about EMCS consoles, refer to the MVSPARM statement in the Tivoli NetView for z/OS Administration Reference.

When using MVS to address commands to another subsystem (such as JES2), correlation depends upon that subsystem’s proper use of the MVS CART message correlators and upon that subsystem’s proper routing of response messages. If messages from another subsystem do not appear to be properly processed by your pipelines, contact the support representative for the subsystem being addressed to see if CART support is available on the version of the subsystem you are using.

– ENDTASK. ENDTASK commands support correlation and can be used in cross-domain pipes for commands between NetView V3R1, or later, programs. ENDTASK commands to and from NetView programs before V3R1 can be executed in a NetView PIPE, but the response does not flow back through the PIPE.

– RUNCMD. Refer to the Tivoli NetView for z/OS Customization: Using REXX and the NetView Command List Language for more information about using RUNCMD.

– Other NetView commands that are correlated (those with displayable output). See "PIPE HOLE" on page 110 for information on determining if a command has correlated output.

• If you have your own commands (user-written) that produce messages asynchronously, you can modify them so that they are supported by NetView Pipelines. There are two ways to do this:
  – Correlation method.
    If your command solicits data from a DST or other NetView task by sending a command to that task by the DSIMQS macro, you can add an option. This option causes a correlator to be attached to the command that is sent. When the command runs, it can return correlated messages to the originating task by issuing DSIPSS TYPE=OUTPUT.
  – Long running command (LRC) method.
    You can use NetView’s long running command support to change asynchronous messages into synchronous ones. Use DSIPUSH to make your command an LRC. However your data is returned to the originating task, it must be made available to your resume routine. Usually, this is done by
causing a command to run at the originating task which can use DSIFIND to access rstorage that is accessible to the resume routine. The resume routine then issues DSIPSS TYPE=OUTPUT and the resulting messages are considered synchronous.

- If a blank is typed between the last character in the NETVIEW stage and the stage separator, the blank is appended to the stage data. NetView will ignore the blank, but passes it to another application where it can be processed as part of the stage data.

Return Codes
The following return codes are valid only when the MOE operand is used.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>Installation exit 03 generated USERDROP.</td>
</tr>
<tr>
<td>-500 to -599</td>
<td>Failure attempting to call installation exit 03. Please report specific return code to the Tivoli Customer Support.</td>
</tr>
<tr>
<td>-108</td>
<td>Command is type=I or type=P.</td>
</tr>
<tr>
<td>-112</td>
<td>Command search failure, usually because the command is too long.</td>
</tr>
<tr>
<td>-116</td>
<td>ACCESS not authorized. Command authorization restrictions prevent processing.</td>
</tr>
<tr>
<td>-120</td>
<td>Command is type=D.</td>
</tr>
</tbody>
</table>

There are other possible return codes indicating storage failure. The code you get depends upon the processing phase when storage failure was detected. Look for DSI124I at the system console for this condition.

Examples

Example: Issuing a Command
To issue the NetView command LIST STATUS=TASKS, trap the resulting messages, select and display messages containing the phrase NOT ACTIVE in positions 55 through 64, and discard the remaining messages, enter:

```
PIPE NETVIEW LIST STATUS=TASKS
   LOCATE 55.10 /NOT ACTIVE/
   CONSOLE
```
PIPE NLOCATE

Syntax

```
NLOCATE: 

NLOCATE position.length /string/
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLOCATE</td>
<td>NLOC</td>
</tr>
</tbody>
</table>

Command Description

The NLOCATE stage command discards messages from the primary output stream that match a specified delimited character string. Messages that do not contain the character string are passed to the primary output stream. Messages that contain the character string are passed to the secondary output stream, if connected.

NLOCATE examines each record for a match to the delimited string. A position and length can be supplied to limit the search to a particular column range. If the delimited string is longer than the length specified on the NLOCATE stage command, matches will never occur, and all messages are passed to the primary output stream.

If the input message is a multiline message, all message lines are examined for the string specified. The entire multiline message is selected and passed to the secondary output stream when any line of the message text matches the string specification.

A message is considered a match if any of the specified strings are located within it.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

NLOCATE terminates when the input stream or both output streams are disconnected.

Operand Descriptions

```
position.length
```

Specifies the character position where searching begins in each message and
PIPE NLOCATE

the length of the search. If you specify a length of *, the remainder of the
message is searched. If you do not specify a position.length, the entire message
is searched.

/string/

Specifies the string for which to search. The first nonblank character
encountered after the stage name or position.length is the delimiter establishing
the boundary of the text string used by the stage. The delimited string ends
when the same character is encountered a second time.

Usage Notes

• NLOCATE cannot be the first stage command.
• You can specify the position.length and /string/ pair up to 40 times.

Examples

Example: Discarding Messages by Content
To issue the NetView command LIST STATUS=TASKS, trap the resulting messages,
discard messages containing the phrase NOT ACTIVE in positions 55 through 64,
and display the remaining messages, enter:

PIPE NETVIEW LIST STATUS=TASKS
    NLOCATE 55.10 /NOT ACTIVE/
    CONSOLE
PIPE NLS

Syntax

NLS:

![Diagram of NLS syntax]

Command Description

NLS converts input messages to their translated versions as specified by the TRANSMSG command. Translated messages can then be displayed on VIEW panels.

Refer to [Tivoli NetView for z/OS Command Reference](#) for additional information about TRANSMSG.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

NLS terminates when both the input stream and output stream are disconnected.

Operand Descriptions

GLOBAL

Specifies that the input messages are to be translated using the TRANSMSG settings.

NONE

Specifies that the input messages are not to be translated. Messages are passed unchanged; however, they are marked so that they will not be translated.

Usage Notes

- After being translated by NLS, and as long as the message attributes remain unchanged, messages will not be translated again by a subsequent NLS stage or by Presentation Services.
- Translated messages should not be stored in a variable and reissued as new messages. The resulting messages will be unusable.
- Remote domain translation requirements will be ignored if a message translated by NLS is routed cross domain. Do not route translated messages cross domain.

Examples

**Example: Display a Translated Message**

The following displays the translated version of BNH054 on the view panel:

```bash
PIPE NLS
```
PIPE NLS

/* Write BNH054 to the IMMED message area */

"PIPE (NAME PlsWait)",
" NETVIEW MESSAGE BNH054",
" NLS",
" EDIT /MESSAGE IMMED DSI009 '/ 1",
"  1.* NEXT",
" '/ NEXT",
" NETVIEW"

Notes:
1. The IMMED option causes the MESSAGE command to display the message in the immediate message area. This option is supported by VIEW when CNMIMDL is used.
2. Message DSI009 is a special message that is issued without a message number.
Syntax

\[ \text{NOT:} \]

\[ \text{NOT stage_specification} \]

Synonyms

NOT has no synonyms.

Command Description

The NOT stage command changes the way output is passed to the primary and secondary output streams for those stages that pass part of their output to the primary output stream and part of their output to the secondary output stream. Such stages are called “selection stages” and examples are CHOP, TAKE, TOSTRING, and LOCATE. NOT causes the specified stage to execute as usual, except that input usually passed to the secondary output stream is passed to the primary output stream and the input usually passed to the primary output stream is passed to the secondary output stream. For example, specifying NOT TOSTRING /ABC/ passes all input up to, and including, the first message containing ABC to the secondary output stream, and passes all subsequent input to the primary output stream.

Termination Conditions

The NOT stage modifies the stage specified in \textit{stage_specification}. Because it is a modifier stage, NOT does not have its own termination conditions. See the information on the stage NOT is modifying for termination conditions.

Operand Descriptions

\begin{itemize}
  \item \textit{stage_specification}
  \begin{itemize}
    \item The stage specification being modified, including its operands.
  \end{itemize}
\end{itemize}

Usage Notes

\begin{itemize}
  \item The NOT stage command does not invert the function of the specified stage. NOT SEPARATE does not become COLLECT. NOT SEPARATE would discard all input, because SEPARATE keeps all input.
  \item The STRIP stage command is not considered a selection stage. NOT STRIP will discard all input.
  \item The stage NOT CHOP \textit{chop-operands} keeps the part of each line that CHOP would usually pass to the secondary output stream, if connected.
  \item Be careful in considering the inversion of output for stage options. For example, when the NOINCL (no include) option is used with TOSTRING, the TOSTRING stage passes the message containing the matching string to the secondary output stream. If NOT is used with TOSTRING NOINCL, the matching message is passed to the primary output stream.
  \item NOT is only supported for stages with two output streams.
\end{itemize}
Examples

Example: Passing Messages After One Containing a Certain String
The following example displays all data that follows the separator line, where TOSTRING normally stops.

```
PIPE < DSIOPEN.CNMNEWS
  | NOT TOSTR NOINCL /============/
  | CONSOLE

--->
--->
--->
--->
--->

--->
News for 8 April, 1998

--->
Our new product is up and running!

...
```

Example: Deleting Characters at Each Beginning Message Line
The following example displays all messages produced by the LIST "" command without the first five characters of each message.

```
PIPE NETVIEW LIST ""
  | NOT CHOP 5
  | CONSOLE
```
PIPE PERSIST

Syntax

PERSIST:

\[
\begin{array}{c}
\text{PERSIST} \, \text{minutes} \\
\text{DISPLAY} \\
\text{ROUTE} \, \text{label:} \\
\text{COMMAND} \, \text{cmd}
\end{array}
\]

Command Description

The PERSIST stage specifies the disposition for correlated output after a pipeline ends.

Operand Descriptions

minutes
Specifies the number of minutes that the PERSIST condition is to be active following the completion of the pipeline in which it is found. The range is 0–100000.

MINUTES
Optional parameter, used only for readability.

DISPLAY
Indicates that the correlated output is to be displayed on the console of the task where the pipeline ran. This is the default.

ROUTE
Indicates that the correlated output is to be sent to the specified task.

label:
Specifies a valid label that can be used by the ROUTE stage.

COMMAND
Indicates that the specified command is to be run under the task where the pipeline ran.

cmd
Specifies the command to be processed.

The command is invoked once for each message received by the PERSIST stage. When the command is invoked, the newly-received message becomes the current message and is available to the command through the use of pipe stage SAFE * or REXX functions such as MSGID.

Usage

Issue NCCF LIST PERSIST to get information about outstanding PERSIST elements.

Usage Notes

- The PERSIST stage defines the action to be taken for messages that arrive after termination of the correlation represented by a CORRWAIT stage. The conditions defined by PERSIST are enabled following termination of the preceding CORRWAIT stage (WAIT) when one of the following is true:
**PIPE PERSIST**

- WAIT times out
- WAIT responds to GO or RESET
- WAIT end prematurely because of a PIPEND (non-zero return code)
- DEBUG option is in effect for the PERSIST stage

- The command, routing, or display specified on your PERSIST stages is activated by the arrival of a message that would have been correlated to the CORRWAIT preceding your PERSIST, had that wait not ended prematurely.
- Unless DEBUG is in effect, the PERSIST condition is disabled when an affirmative end of processing is detected by PERSIST. All native NetView commands provide this affirmative end signal when complete. For VTAM in MVS release 10 and later, VTAM commands DISPLAY, MODIFY, and a few VARY commands also provide it. MVS commands and most VTAM VARY commands do not have an affirmative end.
- If you specify more than one PERSIST stage for a given command correlation environment, only the last specified PERSIST stage will take effect.
- Messages subject to DISPLAY action are exposed to user exits, trapping, automation, and logging just as for uncorrelated messages.
- Messages subject to ROUTE action are routed first, then are exposed as for other messages.
- A message subject to COMMAND action is provided as the current message when the indicated command runs. Any output from the command, including the original message, will be subject to exposure in the same was as the output of a command issued from the command facility command line.
- When PERSIST invokes a command, it does so with the same authority as was in effect for the pipeline which established the PERSIST action.

**Examples**

**Example: Issuing a VTAM Command**

To issue the VTAM command V NET,ACT,ID=X displaying initial messages from the VARY command for 10 seconds and then routing any further messages for an additional 30 minutes, enter:

```
EXCMD oper1 PIPE CORRCMD 10 V NET,ACT,ID=X
| PERSIST 30 MINUTES ROUTE AUTHRCVR
| CONSOLE
```

The default action for CORRCMD VARY is PERSIST DISPLAY. In this example, the default action is overridden by the specified PERSIST stage. Only the specified PERSIST stage is activated.
Pipe Pick

Syntax

PICK:

```
PICK position.length = position.length
  ¬= < <= > >=
```

PAD '00'X

PAD hexstring /string/

Synonyms

PICK has no synonyms.

Command Description

The PICK stage command selects messages satisfying a given criteria and passes them to the primary output stream. Messages that do not meet the specified criteria are passed to the secondary output stream, if connected.

If the input message is a multiline message, only the first line is examined. If selected, the entire multiline message is passed to the primary output stream.

The selection criteria is specified by giving a `position.length` within the message line to be compared against:

- Another `position.length` within the message line or
- A `/string/

If one of the two strings being compared is shorter than the other, the shorter string is padded with the PAD character before comparing the two strings.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

PICK terminates when the input stream or both output streams disconnect.

Operand Descriptions

- `=` Selects the message if the two strings being compared are equal.
- `¬=` Selects the message if the two strings being compared are not equal.
PIPE PICK

<    Selects the message if the first string being compared is less than the second string.

<=   Selects the message if the first string being compared is less than or equal to the second string.

>    Selects the message if the first string being compared is greater than the second string.

>=   Selects the message if the first string being compared is greater than or equal to the second string.

PAD
    Specifies a single character to be used to pad the shorter of the two strings before comparison. PAD is followed by:

hexstring
    Specifies a one character hexadecimal string. A hexstring can be in either of the following forms:
    'nn'X
    X'nn'

Where each n is a number 0–9 or character A–F. Two values for n must be specified.

The default PAD character is '00'X.

/string/
    Specifies the delimited character string to be compared to the string specified by position.length.

position.length
    Specifies the character position where comparison begins in each message and the length of the string being compared. If you specify a length of *, the remainder of the message is compared.

/string/
    Specifies the delimited character string to be compared to the string specified by position.length.

Usage Notes

• PICK cannot be the first stage command.
• PICK examines only the first line of the input message. Use SEPARATE to test each line or EDIT to rearrange lines should other tests be required.

Examples

Additional examples can be found in CNMS1101.

Example: Listing Procedures Used Less Than Six Times

You can use PICK to process the output from MAPCL to obtain a list of all procedures in storage that have been used less than six times.

MAPCL data lines are in the following format. The scale has been added to identify character positions.

|...+....1....+....2....+....3....+....4....+....5....+....6....+....7
|WINDOW  63    1446    71512  08/08/96  13:38:19  R
/* sample for REXX */
'PIPE (NAME LOWUSERX)',
'| NETVIEW MAPCL',   /* obtain display of all REXX in storage */
Notes:

1. Because blanks are less than all numbers in EBCDIC order, the comparison works when a number in the data line is longer than one digit.

2. Header and trailer lines could be restored to the output using FANINANY and the secondary outputs from the two DROP stages. However, the totals on the trailer lines would no longer accurately reflect the data lines above them.
PIPE PIPEND

Syntax

PIPEND:

\[ \text{PIPEND:} \quad * \quad \text{number} \]

Synonyms

Stage Command or Operand | Synonym
--- | ---
PIPEND | PIPEEND

Command Description

The PIPEND stage causes a complex pipeline to end immediately when it receives the first message on its input data stream. PIPEND can be used to:

- End a pipeline which is too complex to be terminated by normal end-of-stream conditions.
- End a pipeline and return a return code to the invoking program.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Termination Conditions

PIPEND terminates when it receives a message or when the input stream disconnects.

Operand Descriptions

- \( * \) Indicates that the data up to the first blank in the message should be used to determine the return code for the pipeline. If this data is a number, it will be returned as the return code from the PIPE command. If it is not a number, return code 100 will be returned from the PIPE command.

- \( \text{number} \)

  Specifies the return code to be returned from the PIPE command.

  \( \text{Number} \) may be any number up to \( 2^{31} - 1 \).

Usage Notes

Return codes returned by the PIPE command should not be used because of their predefined meaning to NetView. These return codes are shown in “PIPE (NCCF)” on page 21.
Note: For NetView use only: use of these return codes can yield unpredictable results.

-1 Is interpreted by pipeline processing as an error in a previous procedure. For example, an error in a command list called from a NETVIEW stage.

-5 Is interpreted by pipeline processing as a RESET request.

Also, any NetView documented return code that might be construed as an error in pipeline initiation should not be used with PIPEND.

Examples

**Example: Ending a Pipeline with a Return Code**

The pipe in the following example ends with two different return codes depending on whether or not the WAIT times out.

If you passed the command LIST STATUS=DSILOG as the first parameter to this REXX example and the log is not busy, the LIST request should easily complete within 2 seconds and the rc returned from the PIPE command will be zero. In this case the example returns Message was with the result of the LIST STATUS=DSILOG command.

If DSILOG is busy with traffic and the LIST command is delayed by more than 2 seconds, the example times out, the rc is set to 8, and Pipe failed with 8 is displayed on the console.

```
/* REXX example of ending a pipeline with a return code */

'PIPE (NAME SETRC END¬)
| NETVIEW' arg(1), /* do user's command */
| ATEND: WAIT 2', /* fail command if not returned in 2 seconds */
| VAR anmsg',     /* keep answer, if any */
| ATEND:',         /* end pipe and connector to wait */
| CONSOLE',        /* display message from CORRWAIT */
| PIPEND *'

IF rc = 0 THEN
  say 'Message was ' anmsg
ELSE
  say 'Pipe failed with ' rc
```

If in this same REXX example MVS D T was passed as an argument, the pipeline would always time out and rc would always be set to 8. Although this is a simple command which should be complete in 2 seconds or less, MVS and VTAM commands do not have the ability to inform a pipeline when they are complete. Because of this, NetView pipelines cannot determine when these commands end.

To prevent a timeout when using VTAM and MVS commands, add a TAKE or TOSTRING stage immediately following the WAIT stage. For example, if you add TAKE 1, PIPEND receives a message only if a timeout occurs. Otherwise, PIPEND does not receive a message and the return code will be zero.
Syntax

PPI Sender:

```
 PPI (DATAONLY)  receiver_name
   (TECROUTE)
   (TRAPROUTE)
   (MLWTO)
   (NV)
```

PPI Receiver:

```
 PPI (APONLY)  receiver_name
```

PPI Requestor:

```
 PPI (APONLY)  receiver_name /string/
```

Command Description

The program-to-program interface (PPI) stage communicates with another address space in the same host using the NetView program-to-program interface. PPI can be used in three ways:

- Sender
- Receiver
- Requestor

When PPI has an input stream, PPI acts as a sender. The data received on the input stream is passed to the receiver specified by `receiver_name`.

If PPI does not have an input stream, PPI acts as a receiver. The `receiver_name` specifies the name where data must be sent to be processed by the PPI stage.

When acting as a receiver, PPI should be followed by `CORRWAIT *` so data can be received continuously without deactivating the receiver. If the receiver is deactivated, even for a short time, senders may encounter errors.

Note: When PPI is used as a receiver, the `pipe option LOWQENAB` is in effect even if not explicitly specified. See "PIPE (NCCF)" on page 20 for more information about LOWQENAB.

When a `/string/` is specified, PPI acts as a requestor. The `/string/` is sent to the receiver specified by `receiver_name` and PPI waits for a response. A receiver name is automatically generated to receive the response.
When acting as a requestor, PPI should be followed by CORRWAIT with a sufficient wait time for the response to be returned. PPI will automatically end the wait when one message is received. This message may be a multiline message.

The PPI stage, unlike other NetView PPI receivers, can receive multiline messages. Multiline messages must be in a specific format to be recognized by NetView:

- The message must be prefixed with a seven character multiline identifier and a one character line descriptor. The multiline identifier must be X'0F0DC4E2C9FFE3'. The line descriptor indicates the line type desired along with whether or not line attributes are provided.
- Line type must be one of the following:
  
  - C  Control line
  - L  Label line
  - D  Data line
  - E  End line (may contain data)

  **Note:** The first line must be a control line.

### Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

### Termination Conditions

PPI terminates when the input stream disconnects or if the secondary output stream disconnects, if defined.

An input stream can only be specified when PPI is used as a sender. Input messages are copied to the primary output stream if the primary stream is connected.

When PPI is used as a receiver or requestor, received messages are written to the output stream. These messages are identified by a sender message attribute if a sender name is provided. This sender name is the sender field IFRAUSDR.

A signed 10-digit decimal return code is written to the secondary output stream if the secondary stream is connected.

### Operand Descriptions

```
receiver_name
```

A one-to-eight character name of the PPI receiver.

When PPI is a sender or requestor, `receiver_name` is the name of the program receiving the sent messages. The abbreviation *ALERT means the alert receiver name defined to NetView.

When PPI is a receiver, `receiver_name` is the name that will be used by other programs to communicate with the PPI stage.
PIPE PPI

When PPI is used as a sender, _receiver_name_ can also be an asterisk (*). Asterisk indicates that the message is to be returned to the program originating the message. The originating message is identified by the sender message attribute (IFRAUSDR).

**APONLY**

Specifies that messages will be accepted only from an APF authorized program.

**DATAONLY**

Specifies that only the data portion of each line will be sent. NetView buffer headers and structures will not be sent. DATAONLY is only valid when PPI is used as a sender.

**MLWTO**

Specifies that the message receiver can receive messages formatted to NetView multiline message standards. Message sent will be sent as a NetView multiline message unit. MLWTO is only valid when PPI is used as a sender.

**NV**

Specifies that the receiver is another NetView. Multiline messages and attributes are sent to the receiver who reconstructs them into a multiline message.

**TECROUTE**

Specifies that the message or alert is to be formatted and transferred to the event/automation service associated with the named PPI receiver. The message will be converted by the event/automation service into a Tivoli Enterprise Console event and sent to the server.

**TRAPROUT**

Specifies that the alert is to be formatted and transferred to the event/automation service associated with the named PPI receiver. The alert will be converted by the event/automation service into a trap and sent to the SNMP manager. Note that text messages are not supported by TRAPROUT.

_/string_/ 

A delimited character string to be sent to the receiver specified by _receiver_name_.

**Usage Notes**

- It is imperative, when passing an alert to the alert adapters, that the entire original alert be sent. Additions to the alert can be made using the NAMEBIND EDIT order. Deletions or other changes to the alert may cause the message to be unrecognized as an alert by the alert adapters.
- When sending messages to another NetView, the NV option will preserve all message attributes except the cross domain sender name.
- When PPI is used as a requestor, two return codes can be output to the secondary output stream: the first results from sending the request, the second from receiving the response. When the send fails, the receive is cancelled and only the return code from the send is passed to the secondary output. When the receiving session cannot be established, only the receive initialization failure code is returned. Refer to the _Tivoli NetView for z/OS Application Programmer’s Guide_ for more information about PPI return codes.
- The PPI stage is not supported under the PPT task.
- When PPI is used as a requestor, the PPI stage chooses a receiver name that will be used to receive a reply. The name chosen is in the form _aa#xxxxx_ where _aa_ is the system default defined by the PPIPREFX keyword on the DEFAULT.
command and xxxx is a value dynamically chosen at run time. For more information about PPIPREFIX, refer to the *Tivoli NetView for z/OS Command Reference.*

If an error is detected in the PPI prefix, a return code is passed to the secondary stream. If a secondary stream is not defined, message DWO411I is issued with the invalid PPI prefix.

- Access to PPI functions can be controlled using SAF or the NetView Security Table.

Security checking is done for the pseudo-keywords RECEIVE and SEND on a DSIPPI command. The SEND pseudo-keyword controls both the sender and requestor functions of PPI.

The SEND and RECEIVE pseudo-keywords correspond to the PPI receiver_name specified on the PPI stage.

To prohibit using:

```plaintext
PIPE PPI GEORGE | WAIT ...
```

Code the following PROTECT statement:

```plaintext
PROTECT *.*.DSIPPI.RECEIVE.GEORGE
```

To prohibit using:

```plaintext
PIPE LITERAL /STUFF/ | PPI SAM ...
```

Code the following PROTECT statement:

```plaintext
PROTECT *.*.DSIPPI.SEND.SAM
```

### Return Codes

The following return codes are returned by the PPI stage as signed, 10-digit decimal numbers:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PPI completed successfully.</td>
</tr>
<tr>
<td>100</td>
<td>A system abend occurred during a PPI service send or receive. This may be a forced closure of the PPI address space.</td>
</tr>
<tr>
<td>104</td>
<td>A user abend occurred during a PPI service send or receive. Refer to the <em>Tivoli NetView for z/OS Application Programmer’s Guide</em> for more information about user abends.</td>
</tr>
<tr>
<td>1001</td>
<td>The AIFR or the input length was not valid.</td>
</tr>
<tr>
<td>1002</td>
<td>Could not identify the data as a message or MSU.</td>
</tr>
<tr>
<td>1003</td>
<td>An incomplete multiline message was discarded by the PPI stage due to the receipt of an unrelated message from the same sender.</td>
</tr>
<tr>
<td>1004</td>
<td>An illegal alert forwarding loop was detected. NetView attempts to avoid loops by not forwarding alerts back to their source. Specifically, PPI (TECROUTE) is not valid for a generic alert whose subvector 92 flag bit 7 is on. PPI (TRAPROUT) is not valid for a generic alert whose subvector 92 flag bit 4 is on. For more information, refer to the SNA library.</td>
</tr>
<tr>
<td>1005</td>
<td>The specified target type does not support the data. For example, (TRAPROUT) was specified for a message which is not an alert.</td>
</tr>
<tr>
<td>1012</td>
<td>The user is not authorized.</td>
</tr>
</tbody>
</table>
Other Any valid return code returned by a PPI request type 4 (init), 14 (send), or 22 (receive). Refer to the [Tivoli NetView for z/OS Application Programmer’s Guide](#) for information about these return codes.

Examples

Example: Generating an Alert from Hex Data
The following REXX example produces an alert similar to the following on the NPDA ALD screen:

```
NTV7E GENAL3 COMC 13:43 EQUIPMENT MALFUNCTION:COMM SUBSYST CTRL
```

The text ‘Here is my subvector 31 stuff.’ is seen by selecting the alert and entering 'D' to view the event detail.

Note: Vector lengths in alerts must be correct or the alert may not be recognized.

```rexx
/* Make an alert */
altxt = '41038D000000000000780000'X
altxt = altxt||'089200001100012345678'X
altxt = altxt||'1010000011000000F2F3F4F5F6F7F8'X
altxt = altxt||'069304032012'X
altxt = altxt||'0E950601150213E1068101011504'X
altxt = altxt||'1103030109C7C5D5C1D3F34040C30D3C3'X
altxt = altxt||'04931001'X
altxt = altxt||'30310602046E01F40512'X||'ENU'||'032111'X
altxt = altxt||'2030'X||'Here is my subvector 31 stuff.'
'pipe (end =) varaltxt',
'a: PPI *ALERT',
'| cons dump',
'= a:',
'|color whi',
'|cons'
```

Example: Responding to Requests
The following simple example responds to the COUNT request with the number of requests processed so far. Other requests will receive the response ERROR 1.

Note: The receiver is not shut down when the pipeline ends so the response can be generated.

```rexx
/* responding to a request */
'PIPE (NAME CNMCOUNT END -)',
  | PPI CNMCOUNT', /* start receive for receiver name "CNMCOUNT"*/
  | WAIT', /* CORRWAIT forever (til RESET or STOP FORCE)*/
  | COUNT EACHLINE', /* using line count for requested data */
  | X: LOC 1.5 /COUNT/ ', /* valid requests... */
  | EDIT LINECOUNT 1', /* constructing the simple response */
  | PPI *', /* sending the response to requestor */
  | 'X:', /* invalid requests come here. */
  | EDIT /ERROR 1/ 1', /* error message... */
  | PPI *' /* sending error message to requestor */
```

Notes:

1. The first stage, PPI CNMCOUNT, records the sender’s ID as a message attribute in each message. The attribute is used by the sixth stage, PPI *, to send the response back to the originator.
2. The EDIT stage in this example could have been written with the WRITELINE order to create a multiline message. Because this example does not assume that the requestor is another NetView, it cannot assume that the requestor can handle a multiline response.

Adding the NV option to the PPI send stage, stage 6, would add the appropriate multiline identifiers to the data before it is sent to the requestor.

**Example: Receiving a Response**

In this example a request is sent to a remote PPI receiver running in another address space. The pipeline then waits for a response.

This example assumes that the remote receiver is another NetView. Because it is another NetView, a multiline response is possible.

```plaintext
/*** issuing a request ********/
address NETVASIS,
  'PIPE (NAME PPIOPS)',
  'PPI OEXXX /egrep "NetView" set.log/'; /* send request 'egrep...' */
  'WAIT 180'; /* wait up to 3 min for resp*/
  'STEM NIVnSET.' /* store response */
```

**Notes:**

1. The wait automatically ends when one message is received. This message may be a multiline message.
2. Multiline identifiers and line descriptors, if any, are removed when the multiline response is built by PPI processing.

For additional PPI examples, see the following:

- [PIPE EDIT](#) on page 68
- Example CNMEALUS in DSICLD
- Example CNMEMSUS in DSICLD
- Sample CNMS1101
Syntax

\[
\text{PRESATTR:}
\]

- \text{PRESATTR [asis color asis highlighting asis intensity]}

Synonyms

<table>
<thead>
<tr>
<th>Stage Command or Operand</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESATTR</td>
<td>COLOR, COLOUR</td>
</tr>
<tr>
<td>BLUE</td>
<td>BLU</td>
</tr>
<tr>
<td>PINK</td>
<td>PIN</td>
</tr>
<tr>
<td>GREEN</td>
<td>GRE</td>
</tr>
<tr>
<td>YELLOW</td>
<td>YEL</td>
</tr>
<tr>
<td>WHITE</td>
<td>WHI</td>
</tr>
<tr>
<td>TURQUOIS</td>
<td>TUR</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>DEF</td>
</tr>
<tr>
<td>REVERSE</td>
<td>REV</td>
</tr>
<tr>
<td>UNDERSCO</td>
<td>UNDER, UND</td>
</tr>
<tr>
<td>BLINK</td>
<td>BLI</td>
</tr>
<tr>
<td>NONE</td>
<td>NON</td>
</tr>
<tr>
<td>NORMAL</td>
<td>NOR</td>
</tr>
<tr>
<td>BRIGHT</td>
<td>BRI</td>
</tr>
<tr>
<td>DARK</td>
<td>DAR</td>
</tr>
</tbody>
</table>

Command Description

The PRESATTR stage command changes how messages are to be displayed at the NetView console. The categories of presentation are:
- Color
- Highlighting
- Intensity

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

PRESATTR terminates when the input stream or the output stream is disconnected.
Operand Descriptions

asis

When a value is not specified in an attribute category, the PRESATTR stage preserves the current value.

color

Specifies the color to display the message. You can specify DEFAULT to display a message in the default color of the terminal on which it is displayed. Color can be one of the following:

- BLUE
- RED
- PINK
- YELLOW
- GREEN
- WHITE
- TURQUOIS
- DEFAULT

A color specification has no effect on monochrome displays.

highlighting

Specifies how the message is highlighted when displayed. Valid values are:

- REVERSE
  Reverse video
- UNDERSCO
  Underlined
- BLINK
  Blinking
- NONE
  Default

You can specify NONE to change a previously highlighted message to the default presentation.

A highlighting specification has no effect on terminals that do not support extended highlighting.

intensity

Specifies the intensity of the message to be displayed. Valid values are:

- NORMAL
  Normal
- BRIGHT
  Bright
- DARK
  Invisible

A specification of BRIGHT has no affect on terminals that do not support bright display.

Usage Notes

- The synonym COLOR is used for PRESATTR in examples and descriptions for ease-of-use.
- The color and highlighting settings may be overridden by message automation. To ensure you get the settings you have specified, use CONSOLE ONLY or WINDOW to display your messages.
Examples

Example: Changing Color of Selected Data
In this example, data is read from a file called NAMES. LOCATE selects all the lines containing the string /BOB/. The selected lines are output to the console in red. All other names are output to the console in blue.

PIPE (END %)
< NAMES
| A: LOCATE /BOB/
| COLOR RED
| CONSOLE
%A:
| COLOR BLUE
| CONSOLE
PIPE QSAM

Syntax

QSAM:

\[
\text{QSAM} \quad \text{(DD)} \quad \text{data_set_name} \quad \text{(DSN)} \quad \text{data_definition_name}
\]

Synonyms

Stage Command or Operand          Synonym
QSAM (read)                       <
QSAM (write)                      >

Note: The < and > synonyms for QSAM read and write can only be used when < and > are immediately followed by a data set name enclosed in quotes. The < synonym has additional functionality as the < (from disk) stage. See "PIPE < (From Disk)" on page 232 for more information.

Command Description

The QSAM stage command reads and writes from dynamically allocated data definition names or data sets. Other devices are also supported when allocated for Physical Sequential access.

The QSAM stage command can be used with either a data definition name defined by the ALLOCATE command, or a fully-qualified data set name. If desired, a data set name can be enclosed in single quotes. The quotes are ignored.

When specified as a first stage, QSAM will read from the data definition name or data set. When not specified as a first stage, QSAM writes to the data definition name or data set. The messages received on the input stream are passed to the output stream.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

When specified as a first stage to read a file, the QSAM stage terminates when end-of-file is reached or when the output stream is disconnected. When not a first stage, QSAM terminates when the input stream or primary output stream is disconnected.

A signed 10-digit decimal return code is written to the secondary output stream if the secondary stream is connected.
Operand Descriptions

\textit{data definition name}

Specifies what data definition name is to be used. If an initial (DD) or (DSN) is not specified, the QSAM stage examines the specification. If the argument is a single 1–8 character value without period delimiters or quotes, then it is considered a \textit{data definition name}. Otherwise, the argument is considered a \textit{data set name}.

You can use a data definition name with the QSAM stage by executing an ALLOCATE command. A data set allocated in this manner may include a member name as part of the specification. A data definition name that is specified to QSAM with a member name will be rejected. When using this method for a device or medium that is not a data set, it must be suitable for Physical Sequential access. Specify DSORG(PS) and a suitable block size.

\textit{data set name}

Specifies what data set name is to be used. If an initial (DD) or (DSN) is not specified, the QSAM stage examines the specification. If the argument is a single 1–8 character value without period delimiters or quotes, it is considered a \textit{data definition name}. Otherwise, the argument is considered a \textit{data set name}.

(DD)

Specifies that the specification is for a data definition.

(DSN)

Specifies that the specification is for a data set name.

Usage Notes

- QSAM cannot access a data definition name and member name combination except through a DDNAME allocated with a member name.
- When neither DD or DSN is specified, the QSAM stage examines the name specification to determine whether it is a data set name or a data definition name. This is the default.
- You can read and write to the same data definition or data set name within a single pipe.
- Access security for the QSAM stage is provided through the READSEC and WRITESEC commands. Refer to the \textit{Tivoli NetView for z/OS Administration Reference} for information on the READSEC and WRITESEC commands. Other errors can stop processing before the security check can be done.

Return Codes

The following return codes are returned by the QSAM stage, on the secondary output stream, as a signed, 10-digit decimal numbers:

\begin{tabular}{|l|l|}
\hline
\textbf{Return Code} & \textbf{Meaning} \\
\hline
0 & QSAM completed successfully. \\
12 & The user is not authorized to the data set. \\
28 & The data set is unavailable. It may be in use by another user or task. \\
32 & The data set does not exist. \\
36 & NetView does not support writing to this data set. For example, the data set may have been defined with RECFM(U). \\
100 & An internal failure or abend occurred. \\
\hline
\end{tabular}
Examples

Example: Reading from a Data Definition
The following will read data from a data set specified by a data definition:

```
PIPE QSAM (DD) allocddd
```

`allocddd` is the 'FILE' value from the ALLOCATE command.

Example: Reading from a Data Set Name
The following will read data from a data set specified by a data set name:

```
PIPE QSAM (DSN) hiqual.midqual.lowqual(member)
```

Note: If the DSN is not partitioned, omit '(member)'.

Example: When (DD) or (DSN) Is Not Specified
When neither (DD) or (DSN) is specified, QSAM examines the argument for periods or parentheses. The presence of these delimiters cause the argument to be considered a data set name. A single 1- to 8-character name is considered a data definition name.

```
MVS VARY 00A,ONLINE
ALLOCATE FILE(RDR) UNIT(00A) OLD RECFM(F) LRECL(80) DSORG(PS) BLKSIZE(80)
PIPE QSAM RDR
```

Example: Writing to an Existing Data Definition
The following will write data from the input stream to a data set specified by a data definition:

```
PIPE ...
| QSAM (DD) allocddd
```

`allocddd` is the 'FILE' value from the ALLOCATE command.

Example: Writing to an Existing Data Set Name
The following will write data from the input stream to a data set specified by a data set name:

```
PIPE ...
| QSAM (DSN) hiqual.midqual.lowqual(member)
```

Note: If the DSN is not partitioned, omit (member).
PIPE REVERSE

Syntax

REVERSE:

```
  REVERSE
     LINE
  MESSAGE
     STREAM
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Command or Operand</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>REVERSE</td>
<td>REV</td>
</tr>
<tr>
<td>LINE</td>
<td>L</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>M</td>
</tr>
<tr>
<td>STREAM</td>
<td>S</td>
</tr>
</tbody>
</table>

Command Description

The REVERSE stage command can be used to reverse message text, multiple line write-to-operator (MLWTO) buffer sequence or change the order of messages in the pipeline.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

REVERSE terminates when the input stream or the output stream is disconnected.

Operand Descriptions

- **LINE**
  Specifies that each line of output is to be reversed, character by character. For example, DS1069I SYNTAX ERROR is changed into RORRE XATNYS I960ISD. LINE is the default.

- **MESSAGE**
  Specifies that each multiline message is to be reversed, line by line. Only multiline messages are affected when specifying MESSAGE.

- **STREAM**
  Specifies that the next stage receives messages in reverse order. The structure of multiline messages is not affected.
Usage Notes

REVERSE cannot be the first stage command.

Examples

Example: Changing the Order of Message Lines
The following example shows how to retrieve the bottom section of an archived net log and reverses the output from descending order to ascending order.

ALLOCATE DDNAME(OLDLOG) DSN('archive-name')
PIPE CORRCMD DSIVSMX GETREV OLDLOG 10 X'FF' X'00'
   REVERSE STREAM
   NOT CHOP 50
   CONSOLE

----> TYPE: OST TASKID: RESOURCE: A01A443 STATUS: NOT ACTIVE
----> TYPE: OST TASKID: RESOURCE: A01A444 STATUS: NOT ACTIVE
----> TYPE: OST TASKID: RESOURCE: NT7ED002 STATUS: ACTIVE
----> TYPE: OST TASKID: TOM RESOURCE: NT7ED002 STATUS: ACTIVE
----> TYPE: OST TASKID: AUTO1 RESOURCE: AUTO1 STATUS: ACTIVE
----> TYPE: OST TASKID: AUTO2 RESOURCE: AUTO2 STATUS: ACTIVE
----> TYPE: OST TASKID: NETOP2 RESOURCE: NETOP2 STATUS: ACTIVE
----> TYPE: OST TASKID: DSILCOPR RESOURCE: DSILCOPR STATUS: ACTIVE
----> END OF STATUS DISPLAY
----> SWITCH DSILog,S

In this example, the last command before archiving was SWITCH.

Attention: Do not use DSIVSMX to access any data set defined to a NetView Optional Task.
PIPE ROUTE

Syntax

ROUTE:

- ROUTE
  - label: [standard NetView label or AUTHRCVR]

Command Description

The ROUTE stage sends messages to another task. The target task is identified by a standard NetView label or by the authorized receiver (AUTHRCVR), as specified by the previous ASSIGN command.

If you use the label syntax, the target task can be local (in the same NetView program) or remote (in another NetView program). For a remote target, the message is routed similarly to a RMTCMD response by SNA or by TCP/IP, depending on the domain name. For more information on the domain name specification, refer to the RMTSYN statement in DSIPARM member CNMSTYLE.

If an argument is not specified, ROUTE reads target specifications from the secondary input. One message is read from the secondary input for each message routed. If the message read from the secondary has multiple lines, a single message from the primary stream is routed to each target specified.

The message is also written to an output stream under the following conditions:
- If the routing is successful, the message is written to the primary output stream, if connected.
- If the routing is not successful, the message is written to the secondary output, if defined and connected. If no secondary output was defined, the message is written to the primary output, if connected.

Note: When multiple targets are specified, the operation is regarded as successful if any one of the message routings is successful. For a remote target, the routing is successful if SNA or IP routing methods accept the message for routing. A subsequent failure (for example the domain is inactive or security prevents the session from being established), is reported by messages to the authorized receiver.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

ROUTE terminates when the primary input stream disconnects.
If an argument is not specified and the target is from the secondary input stream, ROUTE ends when either the primary or secondary input stream disconnects.

Operand Descriptions

*label:* Specifies a valid label. A label can be supplied to the ROUTE stage as a parameter or as input from the secondary input stream. The three part syntax (netid.luname/oper_id:) is defined and used the same way as command labels used with CORRCMD stage with the following exceptions:

- An asterisk (*) supplied for the operator name denotes the current operator ID when the ROUTE takes place.
- A percent sign (%) can be used in place of the oper_id to indicate the message should be sent to the authorized receiver at the target domain, as indicated by the netid and luname.

**AUTHRCVR** Indicates the messages should be sent to the authorized receiver in the local domain.

Usage Notes

- To send messages to a remote domain over IP, the domain’s IP address must be defined using the RMTSYN statement in DSIPARM member CNMSTYLE.
- The authorized receiver is determined by the individual message. For more information, refer to the online help for ASSIGN command (PRI and SEC keywords).
- You can route a message to a specified operator over SNA for any target domain. To route a message over IP or to an authorized receiver, you must have Tivoli NetView for z/OS V5R1.

Examples

**Example: Sending a Message to OPER1**
To send a message to OPER1, enter:

```
PIPE LITERAL /Hello/ | ROUTE /OPER1:
```

**Example: Sending Multiple Messages**
To send multiple messages to operators OPER1, OPER2, and NETOP1 from a REXX Exec:

```
dest.1 = '/OPER1:'
dest.2 = '/OPER2:'
dest.3 = '/NETOP1:'
dest.0 = 3
'PIPE (END &),
| 'NETVIEW LIST TASK', /* generate "a few" messages */
| 'A: ROUTE ', /* route to destination read from below*/
| '/* end of main pipeline */
| '& STEM dest.', /* read in the three labels */
| '/* MLWTO -> "route one message to all" */
| 'COLLECT', /* make copies until next stg disc */
| 'DUP *', /* feed msgs with labels up to ROUTE */
| 'A:'
```

**Example: Sending a Message to the Authorized Receiver**
To send a message the authorized receiver at CNM02, enter:

```
PIPE LITERAL /ABC123I more and more/ | ROUTE CNM02/%:
```
PIPE SAFE

PIPE SAFE

Syntax

SAFE

Synonyms

SAFE has no synonyms.

Command Description

A SAFE is a place to store one or more messages associated with a command procedure. The SAFE stage allows the user to read from or write to a default or named SAFE. The messages in a SAFE retain their full message structure and attributes.

If a multiline message created at 08:16:55 and colored red is stored in a safe, then retrieved and displayed, the displayed message still has the multiline structure, same time stamp, and will be red. Moreover, a DOM that matched the original message will also match the retrieved copy.

The PIPE SAFE stage command is similar to the PIPE KEEP stage command; PIPE KEEP enables you to define a task-global place to store messages and PIPE SAFE is a place to store one or more messages associated with a command procedure. For information about PIPE SAFE stage command, see “PIPE KEEP” on page 123.

The types of SAFE are defined as follows:

default SAFE
The current message associated with a command procedure. For example, when the automation table invokes a command procedure, the default message is the automated message.
• The default SAFE is preserved as long as the command procedure runs.
• A default SAFE can contain at most one message.

named SAFE
A named area for a queue of messages associated with a command procedure group. For example, a REXX command list can write messages to a named SAFE, then call a second REXX command list which can read from, or write to, that named SAFE using the PIPE command.
• A named SAFE is preserved as long as the command procedure group runs.
• A named SAFE can contain any number of messages.
• A command procedure group could have any number of named SAFEs at a given time.

When SAFE is a first stage, the specified SAFE is read into the pipeline. For a named SAFE, this can cause multiple messages to be read into the pipeline.
When SAFE is not a first stage, the input messages are written to the specified SAFE. For the default SAFE just one message is written and all messages are copied to the output stream. For a named SAFE, each input message is written to the SAFE and to the output stream.

### Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

### Termination Conditions

SAFE terminates when the input stream disconnects.

### Operand Descriptions

**APPEND**

Specifies that data should be added after data that already exists in the named SAFE. APPEND is valid only when using a named SAFE (not used with the default SAFE because it can contain no more than one message). The APPEND option is not valid when SAFE is a first stage.

* Specifies that the default SAFE is to be used.

**name**

Specifies the 1–8 character name of a named SAFE. When the command procedure group ends, all named SAFEs created by the group go away and the associated storage is freed.

**SEIZE**

Use SEIZE for performance improvement when you do not need the contents of the safe to remain in the safe after a read operation.

### Usage Notes

- The only access to a named SAFE is through the SAFE stage.
- Any REXX command list that is called by the REXX CALL instruction or is invoked as a REXX function uses the same default SAFE as its caller.
- Since named safes are shared among command procedures, verify that the names you select are not already in use for another purpose. For example, if your command list invokes the WINDOW command (CNME1505), do not use the safe names that are already used by the WINDOW command, such as MSGS.

### Examples

**Example: Determining Whether a Named SAFE Exists**

Use the following test to determine if a given named SAFE (MYSAFE) exists.

```bash
/* REXX sample command list */
'PIPE SAFE MYSAFE',
'| VAR X|
IF SYMBOL('X') = 'LIT' THEN
  Say 'MYSAFE was not found.'
ELSE
  Say 'MYSAFE was found.'
```
**Example: Creating a Named SAFE That Contains a NULL Message**

A named SAFE can exist and can contain a 'NULL' message. The following shows how to create a named SAFE which contains a NULL message.

```/* REXX sample command list */
'PIPE LITERAL //',
'  | SAFE ABC'
```

The SAFE named ABC exists and contains one message. The message in the SAFE has no associated message text.

**Example: Passing Messages to a Second PIPE Command**

The following example shows how a PIPE command can pass messages to a second PIPE command using the default SAFE.

```PIPE LITERAL /Message created by outer PIPE/
   | NETVIEW PIPE (STAGESEP %) SAFE *
   % LITERAL /Message created by inner pipe/
   % COLLECT
   % CONSOLE
   | CONSOLE ONLY
```

The outer pipe generates a message and invokes the inner pipe. The inner pipe reads the outer pipe’s message, adds another message to it, COLLECTs both messages into a multiline message and sends it to the outer pipe. The outer pipe displays the multiline message with the CONSOLE ONLY stage.

**Example: Passing Messages to a REXX command list**

Issue a PIPE command, which invokes a REXX command. The REXX command reads its default SAFE into the pipeline and displays it to the console.

```PIPE LITERAL /Message created by PIPE/
   | NETVIEW SHOWDFLT
   | CONSOLE ONLY
   /* SHOWDFLT REXX COMMAND LIST */
   'PIPE SAFE *',
   '  | LITERAL /Message created by SHOWDFLT/',
   '  | COLLECT',
   '  | CONSOLE'
```

In this example the PIPE command creates a message and calls the SHOWDFLT command list. The SHOWDFLT command list reads the current message (passed by invoking PIPE) into its pipeline. A second message is added and a multiline message is created. The CONSOLE stage within the SHOWDFLT command list passes the multiline message to the CONSOLE ONLY stage of the invoking PIPE and the message is displayed.

**Example: Using a Named SAFE as a Message Queue**

A named SAFE (THISSAFE) is used as a message queue for command procedures in a command procedure group. To begin, invoke the following SAFEEX1 command list:

```/* SAFEEX1 REXX COMMAND LIST */
'PIPE LITERAL /Added by SAFEEX1/',
  | SAFE THISSAFE
'SAFEEX2'
'PIPE SAFE THISSAFE APPEND'
```
When SAFEEX1 is invoked, the THISSAFE named SAFE is created and contains a single message Added by SAFEEX1. Then, SAFEEX2 is invoked. SAFEEX2 adds a second message into the THISSAFE named SAFE and calls SAFEEX3. The SAFEEX3 command list adds a third message then returns to SAFEEX2 which returns to SAFEEX1. SAFEEX1 runs its last PIPE command and reads the THISSAFE messages into the pipeline. Three messages are collected and displayed on a cleared console. When SAFEEX1 completes, the THISSAFE SAFE is freed.
Syntax

SEPARATE:

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEPARATE</td>
<td>SEP</td>
</tr>
</tbody>
</table>

Command Description

The SEPARATE stage command transforms input multiline messages into multiple single-line messages. Input single-line messages are passed without being changed. Output single-line messages inherit all of the attributes of the input messages that created them.

The output of SEPARATE consists of single-line messages. Generally, the number of output messages will be more than the number of input messages. When SEPARATE generates many single-line messages from an input multiline message, all of the output messages have the same message attributes as the message from which they are derived. For example, a 10-line message received from JOB STC00040 exactly at noon and passed through SEPARATE yields 10 distinct single-line messages, each with JOBNAME STC00040 and all apparently received at the same microsecond. If you later display these 10 messages with a CONSOLE stage and subsequently receive a DOM that would have matched the original multiline message, then that DOM matches all 10 of the separated messages.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2 (DATA)</td>
</tr>
<tr>
<td></td>
<td>10 (SEQUENCE)</td>
</tr>
</tbody>
</table>

Termination Conditions

SEPARATE terminates when the input stream or all output streams disconnect.

Operand Descriptions

DATA

Specifies that the lines labeled as data lines, or data-end lines, are passed to the primary output stream. All other lines are passed to the secondary output stream. If a secondary output stream is not defined, all lines which are neither data nor data-end lines are discarded.
SEQUENCE

Specifies that the lines are output to the output streams in sequence. The first line is output to the primary output stream. The second line is output to the secondary output stream, and so on, for as many output streams as are defined. All remaining lines are passed as single-line messages to the last defined output stream.

If only the primary output stream is defined, all lines are output to that stream. If only a secondary stream is defined, for example, when specifying NOT SEPARATE, all lines, except the first line, are output to the secondary stream.

Usage Notes

- SEPARATE cannot be the first stage command.
- SEPARATE directly affects the way that messages in the pipeline are displayed, logged, and searched by other stages.
- SEPARATE can be useful preceding stages that search for matches to a delimited string within a record.
- SEPARATE has no effect on single-line messages.

Examples

Example: Breaking a Multiline Message into Single-Line Messages
To issue the D NET,CDRMS command, allow time for asynchronous messages to return from VTAM, and break the multiline messages into single-line messages, enter:

```plaintext
PIPE NETVIEW D NET,CDRMS
    | CORRWAIT 10
    | SEPARATE
    | CONSOLE
```

Example: Breaking a Multiline Message, Selecting from and Displaying the Results
This example issues a TASKUTIL command, separates multiline messages into single lines, selects messages with occurrences of OPER1 or OPER2, collects them into a multiline message, and displays them.

```plaintext
PIPE NETVIEW TASKUTIL
    | SEPARATE
    | LOCATE /OPER1/ /OPER2/
    | COLLECT
    | CONSOLE
```

Example: Separating Data Lines from Control and Label Lines
This example shows how to separate data lines from control and label lines into two pipeline streams.

```plaintext
PIPE (END %)
    | ... /* stages creating input stream */
    | A: SEPARATE DATA
    | ... /* stages processing data lines */
    | B:
    | ... /* stages processing control/label lines */
```
PIPE SORT

Syntax

SORT:

PAD '00'X
A
PAD 'nn'X
D

(position.length)

Synonyms

SORT has no synonyms.

Command Description

The SORT stage command reads messages from the input stream and writes them to the output stream in a specified order. Only the first line of each message is examined. To sort lines within a message, the SEPARATE stage must be included prior to SORT. If messages contain identical sort fields, they will retain their input stream order when passed to the output stream.

Before any data is written to the output stream, all data is read from the input stream until the input stream disconnects. This causes the stream to be delayed.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

SORT terminates when both the input stream and output stream disconnect.

Operand Descriptions

A  Specifies that the messages will be sorted in ascending order. That is, messages where the sort fields are lower EBCDIC values will be output before those with higher EBCDIC values.

D  Specifies that the messages will be sorted in descending order. That is, messages where the sort fields are higher EBCDIC values will be output before those with lower EBCDIC values.

PAD  Specifies the character, in hex, to be used to pad sort fields when the fields specified extend beyond the end of the message being sorted. The default is to pad with the null character (X'00'). The message itself if not modified.

For example, if the following two messages were sorted with SORT PAD 'C1'X A 17.4:

This is message one
This is message number two
The two sort fields will be:

oneA
numb

And, the messages will be passed to the output stream in the following order:

This is message number two
This is message one

PAD can be in either of the following forms:

'nn'X
X'm'

Each n is a number 0–9 or character A–F.

position.length
The starting position and number of characters defining the sort field.

Position indicates the starting character within the message. By default, position is counted from the first character of the message.

Position can be any positive number.

Length is an unsigned positive number indicating the number of characters from position to be included in the sort field. An asterisk (*) can be specified for length indicating that all characters after position are to be used. Position without length and the period (.) separator will default length to 1.

If length is larger than the available characters, all available characters are used and the PAD value is used to pad the sort field to the required length.

Consider the following message:

PIPS CAN BE FUN!

This ... Results in ...

7.6 CAN BE
9.20 N BE FUN!
8 A

Up to eight sort fields may be specified. Sorting comparison proceeds from left to right order with the later fields only being considered if the previous are equal. Fields can overlap, but doing so causes additional processing time.

Usage Notes

- SORT cannot be the first stage command.
- CASEI cannot be used with SORT.
- RESET during SORT processing can yield unpredictable results.
- SORT is stable. If identical sort fields are specified, sorted messages are kept in the same order.

Examples

Example: Sorting Messages
This example shows three messages sorted by SORT A 1.3 5.1 10.2:

One message
One more message
Another message
The three sort fields for each of these message are:

One m ge
One m me
Another message

Processing proceeds from left to right, so the first fields are examined. For two messages these fields are the same (One). For those two messages, the second fields are examined. They too are the same (m). So, the third field is examined. Since they are different (ge and me), the two records can be sorted appropriately. The messages will be passed to the output stream as follows:

Another message
One message
One more message
PIPE SPLIT

Syntax

```
SPLIT:
```

![Diagram of SPLIT syntax]

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANYOF</td>
<td>ANY</td>
</tr>
<tr>
<td>STRING</td>
<td>STR</td>
</tr>
</tbody>
</table>

**Command Description**

SPLIT divides a line of text into multiple lines.

*Note:* SPLIT acts only on the first line of a multiline message. If all lines should be split, SEPARATE should be used prior to SPLIT.

**Streams**

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

**Termination Conditions**

SPLIT terminates when either the input stream and output stream is disconnected.

**Operand Descriptions**

**AFTER**

The input line will be split just after the specified string or character. Nothing is deleted from the output. The point where the split occurs can be adjusted using `charcnt`.

See also the BEFORE keyword.

**ANYOF**

Indicates that the `/string/` is a list of characters. Any character contained in the list will be a match.

See also the STRING keyword.

**AT**

The line will be split where the specified string or character is found. The matching string or character is deleted from the output.
PIPE SPLIT

BEFORE
The line will be split just before the specified string or character. Nothing is
deleted from the output. The point at which the split occurs can be adjusted
using charcnt.

See also the AFTER keyword.

charcnt
Indicates an offset to the split point. For example, if /string/ is specified, the
value of /string/ is found first, then the split is made charcnt characters before
or after that point. The value of charcnt must be a positive number, negative
number, or zero (0). The default value is zero (0).

Valid values for charcnt are in the range of −10 000 000 — +10 000 000.

See also the BEFORE and AFTER keywords.

STRING
Indicates that the /string/ is a single string. A match occurs only when the
complete string is found.

See also the ANYOF keyword.

/string/
A delimited character string containing a character list or string. SPLIT will
search for /string/ as indicated by the ANYOF or STRING keyword and split
the input line at each occurrence. The default value for /string/ is a single
blank.

Note: One or more characters must be enclosed within the delimiters. /String/
can not be a null string (/ /).

Examples

Example: Splitting at Blanks
The following splits the literal string /HERE IS SOMETHING TO SPLIT/ at each
blank:

PIPE LITERAL /HERE IS SOMETHING TO SPLIT. /
            | SPLIT
            | CONSOLE

The output displayed on the console is:
HERE
IS
SOMETHING
TO
SPLIT.

Example: Splitting Following a String
The following splits the literal string /BUY IT, YOU’LL SPLIT AND LIKE IT BETTER./ three characters after each occurrence of the string /IT/:

PIPE LITERAL /BUY IT, YOU’LL SPLIT AND LIKE IT BETTER. /
            | SPLIT 3 AFTER STRING /IT/
            | CONSOLE

The output displayed on the console is:
BUY IT, Y
OU’LL SPLIT AN
D LIKE IT BE
TTER.
PIPE SQL

Chapter 2. Pipeline Stages and Syntax
Command Description

The SQL stage command queries DB2 tables, inserts rows into DB2 tables, and issues DB2 commands.

For additional information about interacting with SQL databases, see "Chapter 6. Using NetView SQL Stages for Access to DB2" on page 283.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>3</td>
</tr>
</tbody>
</table>

Termination Conditions

SQL terminates when it discovers that a primary output stream is not connected. It also terminates if a negative return code is received from DB2. If this happens, the unit of work is rolled back unless DB2 indicates it has already done this.

Operand Descriptions

Optional keywords are followed by a function keyword. The EXECUTE function does not require additional arguments; SELECT requires a SELECT statement; INSERT requires at least three words of an INSERT statement.

**COMMIT**

At stage completion, commit the unit of work or roll back in the event of an error. This is the default.

**NOCOMMIT**

Do not commit the unit of work when processing is completed without errors. Roll back in the event of an error. Use this option when processing with multiple cursors or to issue DB2 statements from multiple invocations of SQL as a single unit of work.

**NOCLOSE**

Keeps the current plan open after the pipe ends.

**INDICATORS**

The data streams used by SQL SELECT and SQL INSERT include indicator half words in front of the data field. INDICATORS is the default for SQL SELECT.

**NOINDICATORS**

The data streams used by SQL SELECT and SQL INSERT do not include indicator half words in front of the data field. For SQL SELECT, indicator words are read and discarded; thus errors are not reported when null fields are
selected. Null fields contain blanks or zeros as appropriate to the data field format. NOINDICATORS is the default for SQL INSERT.

**PLAN** *word*

The *word* specifies the plan to use.

The CNMSJSQL sample contains the plan name for your level of NetView. The plan name is in the form DSISQLnn, where the *nn* suffix is the level of NetView SQL. This suffix enables you to run different levels on a single system. For NetView Version 1 Release 4, the plan name is DSISQL05. For Version 1 Release 3, the plan name is DSISQL04.

**DIAGNOSE**

The DIAGNOSE options provides additional messages describing the SQL functions as they are run. The output of DIAGNOSE is written to the secondary output stream in yellow, using a message type of ‘apostrophe’ (HDRTYPE). Use this option when debugging applications or at the request of Netview service and support personnel.

**TEST**

The TEST option enables you to run the SQL and SQSELECT stages in a testing mode. DB2 databases are not accessed by the TEST option. Diagnostic messages are issued to describe the SQL services. For DESCRIBE SELECT and SELECT requests, a constant set of all DB2 field types is generated. The TEST mode can be operated without the DSIDB2MT task or DB2 being active. For example, the command:

```
PIPE SQL TEST DESCRIBE SELECT anything
```

Produces a description of all of the fields generated by the test data. The command:

```
WINDOW SQSELECT (TEST) anything
```

Produces a window with a header line and a single record of data under each column.

**Note:** The TEST option is best suited for understanding the conversion of internal and external formats, whereas DIAGNOSE may be more helpful when you are testing an application being developed.

**SSID**

The SSID option enables you to access different DB2 subsystems on the local system. If the *ssidname* is not specified, the last subsystem specified for your task is used. If no subsystem was specified for the task, the subsystem defined by the DSIDB2MT task is used, if DSIDB2MT is active. If you specify SSID*, the last subsystem name used by the task is reset, making the DSIDB2MT defined subsystem again the default. If you use multiple DB2 subsystems with NetView, consider specifying SSID (either with a name or *) in the beginning of each procedure used by that task. Another way to organize your DB2 access is to start one AutoTask for each DB2 subsystem and run requests to the AutoTasks using the NetView labeled command technique.

**Usage**

Tables are loaded using SQL INSERT, queried with SQL SELECT, and maintained with SQL EXECUTE.

**Performing a Query with SQL SELECT**

The argument specifies a complete SELECT statement. One record is written for each row of the result of the query. By default, each column is preceded by a
2-byte indicator word specifying whether the column has a null value or contains data. Use NOINDICATORS to suppress this field in the output record.

In an indicator word, binary zero indicates that the column has a value; a negative indicator word indicates that the column is null. A positive value in the indicator word means that the column is truncated; this should not occur, because each column has as many positions reserved as SQL DESCRIBE reports for the table. Blanks or zeros, as appropriate to the field format, are stored in the unfilled positions of columns that contain a null value and columns that have variable length. When the last field has variable length, the record is truncated to the end of the data present.

/* Query a table, save results in a stem variable */
'pipe SQL select * from jtest |$stem results.'
'pipe $stem results. | cons'

The above example shows how $STEM saves the message colors in the "results." array as well as saving the data.

The results of the queries are written to the primary output stream, and are color-coded green. Output data is written using NetView message type double quote (HDRTYPEK).

If the secondary output stream is defined and connected, any error messages are written there. If a secondary stream is not defined, error output is not sent to the primary stream, but escapes to the parent pipe or is displayed instead. Error messages are also color-coded in red. Error messages are written with the NetView message type apostrophe (HDRTYPEJ).

If the third output stream is defined and connected, additional return code information is provided. The output consists of the following:

- A plus (+) sign or minus (-) sign
- A 10-digit decimal SQL value, including leading zeros
- A blank space
- The input text of the SQL statement that ran.

This stream is useful for monitoring end of file conditions, reported by SQLCODE +0000000100. The stream is color-coded in green and the NetView message type is double quote (HDRTYPEK).

Diagnostic messages (from the SQL TEST or DIAGNOSE options) are color-coded yellow, and are written to the secondary stream. Diagnostic messages are written with the NetView message type apostrophe (HDRTYPEJ).

**Performing a Query with DESCRIBE SELECT**

The argument is a query to be described. One record is written for each field of the query. Refer to the description of the SQLDA in your DB2 documentation for more information.

Each record has five blank-delimited fields of fixed length:

3 The decimal number defining the field type.

16 The field type is decoded or Unknown if the field type is not recognized by NetView SQL Stages. The first four positions have the word LONG if the field is a long character or graphics field.

5 The field length as reported by DB2. This is a single number except for
decimal fields where the precision and scale are reported with a comma between them. For graphic fields, the length is the number of DBCS characters and does not include shift-in or shift-out characters.

5 The maximum length of the field in characters, including a halfword length field if required, computed from the length and field type. This is the number of bytes SQL SELECT reserves for the field in the output record from a query, and the number of bytes required in the input record to SQL INSERT. The length does not include the indicator word. For graphic fields, the length is the number of bytes of DBCS characters and does not include shift-in or shift-out characters, but does include 2 bytes for the length field if the field is variable length.

30 The field name. The record is truncated at the end of the name; the name field is in the range of 1–30 bytes.

Sample DESCRIBE SELECT

/* Describe the result of a query */
'pipe SQL describe select * from jtest | console'

Loading Tables with SQL INSERT
An insert statement with a values() clause or a subquery is executed immediately without reference to an input stream. A values() clause cannot refer to host variables. Either a values() clause or a subquery must be used. DB2 does not provide the ability to insert on a cursor.

Release statement, Set Connection, Set Current Degree, Set Current Package Set, Set Current Rules, or Send Current SQLID
Definition of parameters match the SQL language. Since these statements cannot be executed using dynamic SQL, NetView provides explicit code support for them. These statements may be used in a "PIPE SQL EXECUTE" on the primary input stream as is done for other functions.

Using SQL LISTREGS
This a NetView provided function that lists the values in the DB2 special registers. Output appears as follows:

```
* NTV98 PIPE SQL SSID DB2 LISTREGS | CONS
* NTV98
CURRENT DEGREE=1
CURRENT PACKAGESET=
CURRENT RULES=DB2
CURRENT SERVER=DB2L01
CURRENT SQLID=IBMUSER
USER=IBMUSER
CURRENT TIMEZONE=-50000
CURRENT TIMESTAMP=1998-11-09-09.31.02.048057
```

Using SQL EXECUTE
A statement after EXECUTE is issued first; the primary input stream is then read and each record is performed. All DB2 statements are performed as a single unit of work. Most DB2 statements are supported; refer to the description of the PREPARE statement in your DB2 documentation for a list of unsupported statements. SQL processes COMMIT, CONNECT, and ROLLBACK directly; thus, they are also supported. Unsupported statements are rejected by DB2 with return code -515. Processing stops as soon as an error is reported by DB2.

/* Drop a table */
'pipe lit /drop table jtest/| SQL execute|console'
**Using Multiple Concurrent SQL Stages**

Up to 10 SQL stages can run concurrently in all active pipelines. The option NOCOMMIT should be used in concurrent operations. DB2 considers all SQL stages to be part of one unit of work; an implied commit by a stage causes errors when other stages resume. Explicit commit or rollback is done with SQL COMMIT and SQL ROLLBACK.

```sql
/* Merge two tables */
'PIPE (end ?)',
'SQL nocommit noclose select * from table1 |SEP| p: FANIN | CONS ?',
'SQL nocommit select * from table2 |SEP| p:'
'PIPE SQL execute commit work'
```

Use NOCLOSE to leave the plan open from one pipe to another on the same NetView task. The plan closes when all concurrent SQL stages have terminated. A NOCLOSE option used in any (concurrent) stage of a pipe makes the plan stay open when the pipe ends.

When accessing multiple DB2 subsystems from NetView, you cannot directly access multiple DB2 subsystems on a single task without having the SQL close and reopen the plan. Consider using multiple autotasks which interface to a different DB2 subsystem as servers for other tasks. You can use labeled commands and pipes to correlate the SQL requests running on the separate tasks.

The plan will close when none of the SQL stages within the pipe specify NOCLOSE. For example, PIPE SQL COMMIT WORK | CONSOLE will commit the unit of work and close the active plan.

**Note:** If a plan is left open and the REXX procedure ends, the plan remains open until a subsequent pipe closes it or the task ends. A REXX procedure might use PIPE SQL COMMIT WORK | CONS at the start of SQL processing to ensure any previous plan is closed. Alternatively, use PIPE SQL NOCLOSE CONNECT RESET | CONS to ensure the local database is being used and the plan is open.

Use NOCLOSE:

- When using SQL CONNECT to a remote database, a NOCLOSE enables you to keep the remote connection between two pipes. You may find it convenient to open a remote connection in one pipe, do some processing in REXX, and then finish working with the remote connection in a second pipe. You specify NOCLOSE in the first pipe and omit the operand in the second pipe.
- When using database locks in SQL, use NOCLOSE to keep the locks from one PIPE to the next.
- When using applications requiring two pipes to implement one function, typically, with other (REXX) processing between the two pipes.

**Other Considerations When Using SQL**

When using SQL, consider that:

- DB2 statements are read from the primary input stream when EXECUTE is used. The query results are written to the primary output stream. Error and diagnostic messages are written to the secondary output stream.
- SQL will terminate when the primary output stream is disconnected. It will also terminate if a negative return code is received from DB2.
  - When a negative return code is received from DB2, the unit of work is rolled back, unless DB2 indicates that it has already rolled back.
- EDIT is often used to insert indicator words for columns that are always present.
• **EDIT conversion orders** can convert SQL data from one format to another. See "PIPE EDIT" on page 68 for more information.

• The result of a query can be a single 4-byte binary integer; use EDIT to convert it to decimal, if desired.

```c
/* Determine query size */
"PIPE",
    " SQL select count(*) from table1 where KWD < 'C'",
    "   EDIT 3.4 C2D 1",
    " | CONS"
```

• The NetView packages and plans must be bound before you can access DB2 tables with NetView SQL stages. CNMSJSQL is the input to the preparation process; it is shipped with the NetView JCL samples. Your database administrator gives privileges to the NetView using the GRANT statement.

• Refer to your DB2 documentation for more information about preparing the NetView SQL stages plan (DSISQLnn in sample CNMSJSQL).

• Use the definition member DSIDB2DF to specify the DB2 subsystem you want to use. When DSIDB2MT is started it connects to that DB2. Stopping the DSIDB2MT task causes it to disconnect from the DB2 subsystem.

• An SQL INSERT must have a values() clause specifying literals on MVS. Use EDIT to construct an insert statement from data in the record.

• To access an MVS database through distributed relational access, export the plan.
PIPE SQLCODES

**Syntax**

```
SQLCODES:
```

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLCODES</td>
<td>SQLC</td>
</tr>
</tbody>
</table>

**Command Description**

SQLCODES writes a 44-byte record with the last 11 nonzero SQL codes received. This stage is primarily used to diagnose problems when using the SQL stage.

**Streams**

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

**Termination Conditions**

SQLCODES terminates when the output stream is disconnected.

**Usage Notes**

SQLCODES must be a first stage.
PIPE STEM and PIPE $STEM

Syntax

STEM

Command Description

The STEM stage command can be used anywhere in the pipeline.

When STEM is the first stage, it reads records from an array of stemmed command procedure variables. Each record is passed as a single-line message to the pipeline output stream.

When STEM is not the first stage, it writes each line of each message to a variable within a stemmed array of command procedure variables and to the output stream. In addition, an integer is appended to the given variable name (for example, VARNAME1). The number represents the position of the message line being processed. For instance VARNAME1 is the first line, VARNAME2 is the second line. When all lines are processed, a variable with a zero appended (VARNAME0) is written to the variable pool, but not to the output stream. This variable contains the total number of all lines that were processed.

Thus, if VARNAME1, VARNAME2, and VARNAME3 were created containing messages 1, 2, and 3 respectively, VARNAME0 would contain the number 3.

The use of the STEM stage command is limited to the command procedure environments (REXX, NetView command list language, and HLL). However, if the (COMMON) or (TASK) option is used, STEM can be invoked from message automation, by command originating at the PPT task or an optional task, or by using a labeled command originating in a command procedure. Use of the STEM stage outside of these environments results in message DSI290I and termination of the pipeline.

By contrast, the VAR stage reads and writes to uniquely named variables that do not represent an array.

The $STEM stage command is the same as STEM, except that it also reads or writes the VIEW attribute variables (which start with $) that are associated with the specified array of stemmed data variables. When $STEM is the first stage, the color and highlighting specified in the attribute variables are translated to the output messages. When $STEM is not the first stage, the color and highlighting attributes specified in the input messages are translated to the attribute variables.
Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

If specified as a first stage, STEM and $STEM terminate when the output stream disconnects or when the end of the data stored is reached. If specified as a subsequent stage, STEM and $STEM terminate when the input stream disconnects.

Operand Descriptions

(COMMON)
Specifies that the common global variable dictionary is accessed instead of your personal variable dictionary.

(TASK)
Specifies that the task global variable dictionary is accessed instead of your personal variable dictionary.

APPEND
Specifies that new data should be appended as additional STEM variables following STEM data that already exists as determined by the count in element zero. APPEND can only be used on a stage that is not first.

The APPEND option is not enabled if STEM is the first stage command. For processing of the APPEND option, the record count of the STEM variable must be zero or positive.

COLLECT
Causes STEM to build one multiline message instead of many single-line messages. COLLECT is allowed only when STEM is the first stage in a pipeline. Using the COLLECT operand on the STEM stage is the functional equivalent of using the STEM stage followed by the COLLECT stage, but it is faster and uses less storage. Collect can only be used on a first stage.

stemroot
Specifies the name of the STEM variable to read from or write to. It should end with a period (.) if you are using a REXX command procedure. Do not include an ampersand (&) in the name (an ampersand is implied in the NetView command list language). The name length (name plus appended STEM count) can be up to 11 characters in the NetView command list language and up to 31 characters in REXX and HLL except when $STEM is used, in which case, the limits are 10 and 30, respectively. Lowercase characters in the name are changed to uppercase before being processed. The &l- to &31-variables as used in the NetView command list language are not supported for use in the STEM stage command. However, you can assign these values to or from other named variables, which you can use in the STEM stage command.

The record count for the STEM variable is name with a zero appended to it. The count indicates how many records the STEM variable contains. The STEM records are composed of name with a numeric value appended.

(number)
Specifies the number invocations (generations) to refer back when setting the variables. The number of generations refers to the current nesting level within the REXX, PL/I, or C calling sequence.
(Number) must be zero (0) or greater, and less-than or equal-to the existing number of generations. If (number) is greater than (0), the variables are in a generation preceding the current generation. The specified generation can precede the generation from which the PIPE command is issued if such a generation exists.

The default for (number) is zero (0).

FROM
Indicates a starting point for access to the stem variables. If FROM is specified, frNumber must also be specified. FROM can be used on stages that are first or not first.

frNumber
A positive number. When STEM is a first stage this is the number of the first stem variable written to the output stream. When STEM is not a first stage, this is the number of the first variable stored. Do not specify both FROM and APPEND.

Usage Notes

• When STEM is the first stage command of a pipeline specification, the following conditions apply:
  – The maximum size of a message buffer output from the STEM stage is 32000 bytes. Message buffers exceeding 32000 bytes are truncated to 32000 bytes.
  – If the variable specified on the STEM stage command has a record count that is not valid, the pipeline is rejected with message DWO206I, and the pipeline ends. The record count of the STEM variable must be zero, or positive, and less than 10000000.

• When STEM is not the first stage command of a pipeline specification, the following conditions apply:
  – The value of the count variable (name with '0' appended) is initialized to zero very early in pipeline initialization (unless APPEND is specified). If your pipeline fails to run because of an error or a RESET condition, this variable might have a value of zero, even though your pipeline was not processed. Likewise, if messages are not processed by STEM, the value of the count variable is zero.
  – Input messages to the STEM stage are inspected for message lines as follows:
    - An input stream containing a single message line causes two STEM records to be saved: one STEM variable containing the message, and one STEM0 containing the record count.
    - An input stream containing a 10-line MLWTO causes 11 STEM records to be saved: one for each line of the MLWTO, plus one for STEM0 containing the record count.
    - An input stream consisting of a single-line message and a 10-line MLWTO causes 12 STEM records to be saved and so forth.

• A much more efficient and predictable behavior is obtained when using COLLECT with the COMMON option. If STEM is a first stage, COLLECT should be specified on the STEM stage. If STEM is not a first stage, a COLLECT stage should precede the STEM stage.

For example, if one task is updating the common global stem X. and your task is reading it, the following might get some of the updated X. values and some of the older values:

PIPE STEM (COMMON) X. | CONSOLE
PIPE STEM and PIPE $STEM

However, the following command gets all of the older values or all of the updated values.

```
PIPE STEM (COMMON) X. COLLECT | CONSOLE
```

- STEM ignores the structure of the messages it receives. Thus, 10 one-line messages sets 10 stem records and one 10-line message also sets 10 stem records.
- Multiple streams cannot be input to a single STEM stage. The FANIN stage can be used to collapse multiple streams into a single output stream, which can be used as input to STEM. See "PIPE FANIN" on page 102 for more information about FANIN.
- If the value of the count variable (name with '0' appended) was not set for TASK and COMMON variables, the value of the count variable will be null. However, a null value is handled the same way as if the count variable had been set to zero (0).

Examples

**Example: Writing to Stemmed Variables**

If a NetView command list named PRIME runs as a result of NetView automation, the command list drives a second command list named SECND. You can give SECND the same access to the message that called PRIME, and save all output data to PRIME’s variable SSLT (in REXX), by entering:

```
PIPE SAFE *
   | NETVIEW SECND
   | STEM SSLT.
```

**Example: Saving the Count of Records Processed**

In this example, the DSICMD file is read into the pipeline and saved to the STEM variable named A. When the pipeline completes, the record count in A0 indicates the number of lines read from DSICMD.

```
/* REXX COMMAND LIST */
'PIPE < DSICMD INCL',
   'STEM A'
SAY 'THERE ARE ' A0 ' LINES IN DSICMD.'
```

For examples of using $STEM, refer to the WINDOW command list (CNME1505).
PIPE STRIP

Syntax

STRIP:

Command Description

The PIPE STRIP stage removes blanks or other specified characters from the beginning or end of message data. Alternately, STRIP will remove all characters up to a blank or other specified characters.

You can use the STRIP stage to remove unwanted blanks or other characters before you use the JOINCONT stage.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

STRIP terminates when the input stream or the output stream is disconnected.

Operand Descriptions

BLANK
The default is to remove blanks.

BOTH
Removes blanks or other specified characters from both the beginning and the end of the text in the message lines. This is the default.

/charset/
Specifies the set of characters to be stripped. Order and duplicate characters are ignored. The delimited set must be specified.

The first nonblank character encountered after the keywords is the delimiter which establishes the boundary of the character set used by the stage. The delimited set ends when the same character is encountered a second time. // is interpreted as a null set.

LEADING
Removes blanks or other specified characters from only the beginning of the text in the message lines.

limit
The maximum number of characters to be removed by STRIP. If you use BOTH, the limit applies separately to the leading and trailing strip operation.
PIPE STRIP

TO/NOT
Removes blanks or other specified characters that are not blank (or not specified). TO and NOT have exactly the same function.

TRAILING
Removes blanks or other specified characters from only the end of the text in the message lines.

Usage Notes
- STRIP cannot be the first stage command.
- A delimited character set is not recognized as a sequence of characters. Each character is considered individually. If you specified the delimited set /CAT/ with TRAILING, any message ending with an A, C, T, or any combination of those characters would be considered a match.

Attention: Be cautious when using NOT to strip non-null characters from a message. If you omit nn to limit the strip action, the entire message might be stripped.

Examples

Example: Stripping Leading Characters
For this example, you have established a file member named AFILE in which records begin either with the characters 'A' or 'T' as shown:

A
TAME
ARTFUL
AARDVARK
ATE
THE
APPLE

To read the records into a pipeline, strip leading characters 'A', 'T', 'AT', or 'TA', and write the results to the console, enter:

PIPE < AFILE
| STRIP LEADING /TA/
| CONSOLE

Response
(blank)
ME
RTFUL
RDVARK
E
HE
APPLE

Example: Stripping Sequence Numbers from the End of a Message
For this example, you have established a file member named THISFILE. The records are 80 bytes long and end in eight character sequence numbers.

To read the records into a pipeline, strip any character that is not null for eight characters from the end of the record, write the resulting messages to a stem variable named 'OUTLINE.', and process in a command list as shown.
/* REXX command list */
'PIPE < THISFILE',
   | STRIP TRAILING NOT // 8',
   | STEM OUTLINE.'
Syntax

SUBSYM:

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSYM</td>
<td>SUBS</td>
</tr>
</tbody>
</table>

Command Description

The SUBSYM stage command takes messages in the pipeline and substitutes any MVS or user-defined system symbolics, including the NetView-supplied &DOMAIN symbolic, found in those messages for their system values.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

SUBSYM terminates when the input stream or the output stream disconnects.

Usage Notes

- SUBSYM cannot be the first or last stage command.
- Substitution is performed on the &DOMAIN symbolic, unless substitution was disabled when NetView was started. For MVS-defined and user-defined system symbolics, substitution is not performed under the following conditions:
  - If you are not running on an MVS system
  - If you are running on an MVS system before MVS Version 5 Release 2
  - If substitution was disabled when NetView was started
  - If you have not defined an MVS system symbolic on your MVS system
PIPE TAKE

Syntax

TAKE:

Command Description

The TAKE stage command enables you to specify the number of messages or lines that are passed to the primary output stream (if any). All messages or lines in excess of this number are passed to the secondary output stream. If either primary or secondary stream is disconnected, messages that would have been passed to the particular stream are discarded.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

TAKE terminates when the input stream or both output streams are disconnected.

Operand Descriptions

- **count**
  Specifies the maximum number of messages or lines to be passed to the primary output stream. Valid values are in the range of 0–1000000. The default is 1.

- **FIRST|LAST**
  Specifies whether the messages or lines to be passed to the next stage are the first count messages or the last count messages. The default is FIRST.

- **LINES**
  Specifies that the count is made of individual lines without regard to their grouping as multiline messages. If the indicated count is satisfied during processing of a multiline message and a secondary output is connected, then the AIFR data for the multiline message is replicated and passed to the secondary output, along with the remaining lines for that message.

- **MSGS**
  Specifies that the count indicated applies to whole messages, which can consist of zero or more lines. This is the default.

Usage Notes

- TAKE cannot be the first stage command.
**PIPE TAKE**

- TAKE LAST delays the stream. This means that TAKE LAST produces no output until the previous stage disconnects its output stream. This causes delayed processing by stages following TAKE LAST.
- TAKE LAST can affect performance because it must process the entire input stream, before it is able to send the LAST messages selected to the output stream.

**Examples**

**Example: Selecting and Displaying the Last Message**
To issue the VTAM command DISPLAY NET,APPLS, allow 10 seconds for each asynchronous message to return to the pipeline from VTAM, terminate the wait early (TAKE 2), select the last message (TAKE LAST 1), and display the message, enter:

```plaintext
PIPE VTAM DISPLAY NET,APPLS
   | CORRWAIT 10
   | TAKE 2
   | TAKE LAST 1
   | CONSOLE
```
PIPE TOSTRING

Syntax

TOSTRING:

- TOSTRING
- ALL
- FIRST
- LAST
- INCL
- NOINCL
- position.length
- /string/

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOSTRING</td>
<td>TOS</td>
</tr>
</tbody>
</table>

Command Description

The TOSTRING stage enables you to select messages in the input stream up to, and including, the message containing the text that matches the delimited string that you specified. Selected messages are passed to the primary output stream. Those not selected are passed to the secondary output stream, if connected. You can specify up to 40 delimited strings, each with an optional position and length pair, to limit the column range of the search.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

TOSTRING terminates when the input stream or both output streams disconnect. If a secondary output stream is not defined, TOSTRING terminates when it matches its target.

Operand Descriptions

- **ALL | FIRST | LAST**
  This keyword affects processing only for MLWTOs. Specifies whether the first, the last or all lines of multiline messages are compared. The default is ALL.

- **INCL**
  Include the matched message in the primary output stream. This is the default.

- **NOINCL**
  Do not include the matched message in the primary output stream.

- **position.length**
  Specifies the character position in each message where searching begins, and
**PIPE TOSTRING**

the length of the search. If you specify a *length* of *, the remainder of the message is searched. If you do not specify a *position.length*, the entire message is searched.

/*string*/

Specifies a string for which to search. A message is considered a match if any of the specified strings are found within it. The first nonblank character encountered after the stage name or *position.length* is the delimiter which establishes the boundary of the text string used by the stage. The delimited string ends when the same character is encountered a second time.

**Usage Notes**

- TOSTRING cannot be the first stage command.
- If the delimited string is longer than the length specified on the TOSTRING stage command, no matches occur and all messages are discarded from the pipeline.
- TOSTRING is a terminating stage, meaning that once a match is found, processing for this stage terminates along with any outstanding CORRWAIT.
- You can specify the *position.length* and /*string*/ pair up to 40 times.

**Examples**

**Example: Using TOSTRING to End a Wait**

Because VTAM commands do not return an end of response indication, the text of the response must be examined instead. In this REXX example, TOSTRING will terminate after it has processed the message containing the string */IST314I/.* This termination disconnects the output stream of CORRWAIT and causes the wait to end.

```rexx
PIPE (NAME ENDSOON)',
| VTAM D NET,LINES',       /* Issue VTAM display command */
| CORRWAIT 50',           /* Wait up to 50 seconds for response*/
| TOSTRING */IST314I/'    /* Last line expected: IST314I END */
| SEPARATE',              /* (for example) */
| ... ',                  /* processing as required. */
```

The following stage, in this case SEPARATE, does not have a disconnected input stream until after it consumes the last message. The remainder of the pipeline can process the message containing */IST314I/ before it ends.
## Syntax

```
TSO:  
     TSO    (ECHO)    tso_command
```

## Command Description

TSO transfers a command to a NetView TSO server, which is a batch job submitted by NetView or a started task. The command will be executed and the results returned. CORRWAIT should follow TSO to enable the return of command responses.

Commands can be passed on the stage or on the input stream. If `tso_command` is not specified and a multiline message is received on the primary input stream, the first line of the message is considered the TSO command and all other message lines are passed as the data of that command. If `tso_command` is specified, that command will be executed for each input message or only once if there is no primary input stream.

When TSO has a primary input stream and no `tso_command` is specified, the command to be executed is read from the primary input stream. The message input to the TSO stage at the time the TSO command is scheduled will be passed to the TSO server for execution. The message data is contained in a sequential file `userid.NVCMDIN` that is allocated as DD NVCMDIN. NVCMDIN is not allocated if there is no current message for the command.

A secondary input stream can be connected to TSO. The secondary input stream must contain records that contain the TSO user ID optionally followed by the TSO server job member name. The user ID and server job member name must be separated by at least one blank. If the server job member name is not specified, it defaults to CNMSJTSO. The default server is the server specified for this operator by the START command.

If a secondary input stream is not connected, the TSO server started for the issuing operator is used. The default operator is specified on the START command.

The TSO server job member name is specified on the START command with the TSOSERV and MEM keywords.

Each primary input stream message is associated one-for-one to each secondary input stream message. That is, for each primary input stream message there may be a secondary input stream message indicating the TSO server where the primary input stream messages are to be sent. If either the primary or secondary input stream is exhausted before the other, the remaining input on the active input stream will be discarded. If no secondary input stream is defined, all messages on the primary input stream are sent to the TSO server started for the issuing operator.
Figure 20 shows an example execution of a TSO command. Within a pipeline a NETSTAT command is duplicated using the DUP stage. Multiple NETSTAT commands become the primary input stream to the TSO command. From elsewhere in the pipeline, a secondary input stream is connected to the TSO stage. This secondary input stream contains records with the TSO user ID followed by a TSO server job member name.

The TSO stage sends the NETSTAT command to all TSO server jobs on the secondary input stream. Responses are returned from TSO to the CORRWAIT stage following the TSO stage.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

TSO terminates when the primary input stream and the primary output stream is disconnected. If followed by CORRWAIT, CORRWAIT will automatically end when the TSO command is completed and the response has been received.

Operand Descriptions

ECHO

Write the command to the primary output stream prior to the execution of the command.
**tso_command**

A TSO line mode command to be invoked at the TSO server. *tso_command* is required when TSO has no input stream and is otherwise optional.

If *tso_command* is specified, the command will be invoked once if the TSO stage has an input stream. Otherwise, the command will be invoked when an input message is passed to the TSO stage.

If *tso_command* is not specified, the commands to be invoked are read from the input stream.

**Note:** Long running commands, such as IPTRACE, and commands requiring a dialog, such as ACCOUNT, are not supported.

**Usage Notes**

- Command responses are delayed until command completion. For example, a TSO command that issued a message every 2 seconds and runs for 10 seconds will appear to delay 10 seconds before issuing 5 messages together.
- Command execution is single threaded for each server.
- Commands requiring a dialog are not supported. However, data for a dialog can be assembled into a multiline message. This multiline message can be used to drive a TSO REXX procedure managing the dialog within TSO.
- Command authority is checked prior to and after submitting the command.

Prior to submission authority is checked for the pipe stage and for the verb of the TSO command. The verb of the TSO command is the first blank delimited token. To permit only BOSS to use the TSO stage, you would code the following:

```
PROTECT *.*.DSIPIITSO
PERMIT BOSS *.*.DSIPIITSO
```

The verb of the TSO command is treated as the value of the PROTECT keyword VERB for authority checking purposes. For example, if you want to prevent operators from issuing a PIPE TSO IPTRACE command, you would code:

```
PROTECT *.*.DSIPIITSO.VERB.IPTRACE
```

The specified server is treated as the value of the PROTECT keyword TSOSERV for authority checking purposes. For example, to prevent operators from using server USER1 in combination with the started job SERVJOB1, you would code:

```
PROTECT *.*.DSIPIITSO.TSOSERV.USER1/SERVJOB1
```

Refer to the START command in the NetView online help for additional information on started jobs.

**Note:** The TSO stage cannot resolve TSO synonyms. All command synonyms must be protected.

TSO performs command authority checking using the same rules that apply when the TSO user name is directly logged-on.

**Return Codes**

A secondary output stream can be connected to receive command response codes. Each code begins with a 10-digit, 0-padded, signed number. Nonzero codes indicate an error and are followed by a space and keyword indicating the source of the error such as +0000000100 PPI.
The keyword can be one of the following:

**AUTHCHK**
An error occurred during authorization checking. AUTHCHK will be followed by a DSIKVS return code indicating the error. Refer to Tivoli NetView for z/OS Customization: Using Assembler for DSIKVS return codes.

**COMMAND**
Indicates that the preceding is the return code resulting from the execution of the TSO command. Refer to the appropriate publication for the TSO command being executed for further information.

**INITERR**
An error occurred during TSO stage initialization.

+0000000122
Indicates that the TSO server is not started or that a TSO server name was not specified on the stage.

**PPI**
An error occurred connecting to the NetView Program-to-Program Interface or when sending the command to the destination.

+0000000032
Indicates that an out of storage error occurred.

+0000000100
Indicates a system abend occurred within PPI processing. This can occur when the Program-to-Program interface is cancelled.

+0000000104
Indicates that a user abend occurred in PPI processing.

Refer to the Tivoli NetView for z/OS Application Programmer’s Guide for information on additional Program-to-Program Interface send, receive, transaction, and initialization return codes.

**TSO**
An error occurred in TSO operations supporting the command invocation.

−0000000003
Indicates that the TSO command does not exist.

**TSOSERV**
An error occurred when trying to identify the TSO server.

+0000000002
Indicates that an invalid TSO server name was specified.

+0000000004
Indicates that the TSO server was not found.

+0000000008
Indicates that TSO user was not found.

+0000000012
Indicates that communications could not be established with the PPI.

+0000000016
Indicates that the SAF product denied the requester access to the TSO server.

+0000000020
Indicates that a storage problem occurred during TSO server identification.
Examples

Example: Discover TSO Stacks Serving a User
In the following example, the TSO stacks serving a TCP user are listed:

/* REXX Example usage of TSO stage. */
/* Purpose: to discover which of several TCP stacks is serving a */
/* given user. */
/* */
/* Input: TCP User ID */
/* */
/* Output: stack name and current state */
/* (multiple lines are shown for user ids using multiple */
/* ports) */
/* */
/* Assumptions: */
/* */
/* 1. the name of the TCP stack or other mnemonic is used */
/* as a member name (copy of CNMSJTSO) for starting the */
/* TSO servers. See help for START TSOSERV. */
/* */
/* 2. authority has been granted to use any server found. */
/* (Otherwise, further filtering can be done. See note 1, */
/* below.) */
/* */
/* 3. TSO server has PROFILE MSGID in effect. */
/* */
ADDRESS NETVASIS
arg theUser
IF words(theUser) <> 1 THEN
  DO;
    say 'User ID required'
    EXIT 12
  END;
ELSE theUser = left(theUser,8)

'PIPE (NAME SRVRLIST)', /* Obtain list of all servers. */
  'NETV LIST STATUS=TSOSERV', /* sep & discard label lines. */
  'SEPARATE DATA',
  'SORT 21.8', /* sort on member (= stack name). */
  'LOCATE? See NOTE 1, below */
  'DELPURES KEEPFIRST 21.8', /* keep one of each. */
  'EDIT WORD 2 NEXT', /* build record: tso userid and */
  'WORD 3 NEXTWORD', /* member name from each line. */
  'STEM STACKS.' /* save these to feed TSO stage */
  'PIPE (NAME MULTSTAT END %)', /* on its secondary input, below. */

'PIPE (NAME MULTSTAT END %)', /* creating a command */
  'LITERAL /NETSTAT/;', /* make copies indefinitely; NOTE 2 */
  'DUP *', /* read destinations from "A" below */
  'A: TSO', /* wait plenty */
  'WAIT 90', /* can't use SEP DATA */
  'SEPARATE', /* get just the data lines */
  'LOCATE 1.10 /EZA0185I/', /* and lines with our user's name */
  'LOCATE 10.10 /*theUser*/', /* Now that we have data about our user */
  '/* where did it come from? The answer */
  '/* is in the attributes: JOBID */
  '/* Build msg with member name, */
  'EDIT JOBID 1', /* user's name, and */
  'word 2 NW ', /* current state. */
  'word 6 NW ', /* */
  'CONS', /* */
  'TAKE LAST 1', /* IS there a last one? */
  'PIPEND 2', /* IF so, make rc = 2 */
  '/* ------- end of main pipeline ------- */
  '/STEM STACKS.', /* first stage: read server list */
  'A:' /* feed this to TSO secondary */
IF rc <> 2 THEN say theUser 'not found.'

/* NOTE 1 If desired, a LOCATE stage could be inserted at this point to select one or more TSO userids.
   You might want to do this, if security requires that a general user be limited in choice of servers to use. */

/* NOTE 2 Infinitely many copies?!? Not really, since the TSO stage has a secondary input stream (servers to use), it will only accept as many commands as there are servers feed to it. After the secondary input stream disconnects, TSO stage disconnects and the copying ends. */
Syntax

TSROUTE:

Synonyms

Stage Command Synonym

TSROUTE TSR

Command Description

TSROUTE sends a copy of each message line to CNMTAMEL. CNMTAMEL formats the message into an instrumentation event. This event is then sent to each Topology Display Server with an existing NETCONV session. The message lines are also written to the primary output stream.

Messages produced by CNMTAMEL are BNH351I through BNJ354I. All other messages are ignored.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

TSROUTE terminates when the input stream is disconnected.

Usage Notes

- TSROUTE cannot be the first stage command.
- TSROUTE requires exactly one input stream.

Return Codes

A secondary output stream can be connected to receive a signed, 10-digit return code:

+0000000000 Indicates that at least one message line was successfully sent to CNMTAMEL.

+0000000104 Indicates that the line written was longer than 32000 characters.

+0000000204 Indicates that the line could not be written to CNMTAMEL.
PIPE TSROUTE

Examples

Example: Automation Table Sample
Refer to sample DSIAPML for an example of TSROUTE.
PIPE UNIX

Syntax

UNIX:

```
UNIX: \(\text{(ECHO)}\) \(\text{unix\_command}\)
```

Command Description

The PIPE UNIX stage command transfers a command to a NetView UNIX server where the command will be executed and the results returned. CORRWAIT should follow UNIX to enable the return of command responses.

Commands can be passed on the stage or on the input stream.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>3</td>
</tr>
</tbody>
</table>

Termination Conditions

UNIX terminates when the input stream and the primary output stream is disconnected. If UNIX is followed by CORRWAIT, CORRWAIT will automatically end when the UNIX command completes and the response has been received.

Operand Descriptions

**ECHO**

Write the command to the primary output stream prior to the execution of the command.

**unix\_command**

A UNIX line mode command to be invoked at the UNIX server. *Unix\_command* is required when UNIX has no input stream and is otherwise optional.

If *unix\_command* is specified, the command will be invoked once if the UNIX stage has an input stream. Otherwise, the command will be invoked when an input message is passed to the UNIX stage.

If *unix\_command* is not specified, the commands to be invoked are read from the input stream.

Usage Notes

- Command authority is checked prior to and after submitting the command.

Prior to submission authority is checked for the pipe stage, but no checking is done of the UNIX command. To permit only BOSS to use the UNIX stage, you would code the following:

```
PROTECT *.*.DSIPIUNIX
PERMIT BOSS *.*.DSIPIUNIX
```
UNIX commands are submitted under the UNIX user name equivalent to the NetView operator ID. The command will be checked for authorization by UNIX according to the same rules that apply when the UNIX user name is directly logged-on.

- When UNIX has an input stream, the command to be executed is read from the input stream. The message input to the UNIX stage at the time the UNIX command is scheduled will be passed to the UNIX server for execution. When the input is a multiline message, the input data is available to the target command on its primary input.
- A secondary output stream can be connected to receive command response codes. See "Return Codes" for information on codes passed to this stream. If a tertiary stream is connected, message DSI037I is written to the tertiary stream. Message DSI037I contains the UNIX process ID created for each command.

Note: DSI037I is logged even if a tertiary stream is not defined.

Return Codes
A secondary output stream can be connected to receive command response codes. Each code begins with a 10-digit, 0-padded, signed number. Nonzero codes indicate an error and are followed by a space and keyword indicating the source of the error such as +0000000100 PPI.

The keyword can be one of the following:

PPI
- An error occurred connecting to the NetView program-to-program interface or when sending the command to the destination.
- +0000000100 Indicates a system abend occurred within PPI processing. This can occur when the Program-to-Program interface is cancelled.
- +0000000104 Indicates that a user abend occurred in PPI processing.

For information about other codes, refer to the Tivoli NetView for z/OS Application Programmer’s Guide.

UNIX
- An error occurred in UNIX operations supporting the command invocation. Immediately following the response code are the rc, retval, errno, and errnojr specific to the UNIX error. The format of the UNIX response code is −0000000000 UNIX rc, retval, errno, errnojr. For more information about these codes, refer to the MVS OpenEdition® library.
- −0000000001 Indicates that an attempt was made to run the UNIX server in a non-UNIX/390 REXX environment.
- −0000000002 Indicates that an unsuccessful call was made to DSIPHONE.
- −0000000003 Indicates that the UNIX command failed.
- −0000000004 Indicates that the server is unable to spawn child processes. The UNIX server terminates.
Indicates that an internal error during open pipe (open) processing caused UNIX command processing to terminate.

Indicates that an internal error during spawn (spawn) processing caused UNIX command processing to terminate.

Indicates that an internal error during write to pipe (write) processing caused UNIX command processing to terminate.

Indicates that an internal error during read from pipe (read) processing caused UNIX command processing to terminate.

Indicates that an incorrect UNIX command was sent.

Indicates that an internal error during get user information (getuid) processing caused UNIX command processing to terminate.

Indicates that an internal error during set user identity (setuid) processing caused UNIX command processing to terminate.

Indicates that an internal error during set user’s group (setgid) processing caused UNIX command processing to terminate.

Indicates that an attempt to send outback back to the PPI failed for unexpected reasons.

Indicates that the PPI has become inactive and the server had an unexpected error when pausing between attempts to reestablish its PPI receiver.

Indicates that permission was denied. The specified user does not have the authority required to run the submitted UNIX command.

Indicates that an internal error during set home directory (cwd) processing caused UNIX command processing to terminate.

Indicates that the CNMEUNIX PPI receiver is still active from a previous invocation of the server.

Indicates that an internal error during close pipes processing (close) caused UNIX command processing to terminate.

Indicates that an internal error occurred while reading a file.

Indicates that an internal error during get supplementary group ID information (getgroupsbyname) processing caused UNIX command processing to terminate.
Indicates that an internal error during set user’s supplementary groups (setgroups) processing caused UNIX command processing to terminate.

Examples

Example: List the Current Working Directory
The following code changes to the current working directory to /usr/lpp and then lists the contents of that directory. The directory listing is displayed in green followed by the response code displayed in yellow.

```
NetVAsIs PIPE (END +)
  A: UNIX cd /usr/lpp; ls -al
    WAIT 19
    COLOR GREEN
    CONSOLE ONLY+
  A: COLOR YELLOW
    CONSOLE ONLY
```

Example: Execute Commands Contained In DSIPARM
The following code sends the UNIX script file contained in DSIPARM member LW to UNIX for execution. The command results are displayed in green followed by the UNIX response code displayed in yellow.

```
NetVAsIs PIPE (END +)
  < DSIPARM.LW
    COLLECT
    A: UNIX cat > script.cmd; chmod 777 script.cmd;./script.cmd
    WAIT 19
    COLOR GREEN
    CONSOLE ONLY+
  A: COLOR YELLOW
    CONSOLE ONLY
```

Example: Compile and Execute a Java™ Sample
The following code sends the Java sample HelloWorld from the NetView data set to UNIX where it is compiled and executed. The results are returned to the invoker. Results are displayed in green followed by the UNIX response code displayed in yellow.

For simplicity, the example is broken into three separate operations:

1. Send the source code to UNIX:

```
NetVAsIs PIPE (END ;)
  < CNMJSHW
    STRIP TRAILING
    COLLECT
    A: UNIX cat > HelloWorld.java
    WAIT 99
    COLOR GREEN
    CONSOLE ONLY;
  A: COLOR YELLOW
    CONSOLE ONLY
```

2. Compile the HelloWorld Java source program:

```
NetVAsIs PIPE (END +)
  A: UNIX javac HelloWorld.java
```
3. Run the HelloWorld executable:

```
NetVAsIs PIPE (END +)
A: UNIX java HelloWorld
  WAIT 99
  COLOR GREEN
  CONSOLE ONLY+
A: COLOR YELLOW
A: CONSOLE ONLY
```
PIPE VAR and PIPE $VAR

Syntax

VAR and $VAR:

```
VAR (0) (COMMON) (TASK) (number) name
$VAR
```

Command Description

The VAR stage command can be used anywhere in the pipeline specification.

When VAR is the first stage command, records are read from the variable specified. Each record is passed as a single-line message to the pipeline output stream.

When VAR is specified as a subsequent stage, messages are read from its input stream and writes them to both the specified variables and to its output stream. Data from the first input message is placed in the first variable and written to the output stream, data from the second message goes in the second variable and also to the output stream, and so on.

When all specified variables have been assigned, VAR writes subsequent messages directly to its output stream.

The use of the VAR stage command is limited to the command procedure environments (REXX, NetView command list language, and HLL). However, if the (COMMON) or (TASK) option is used, VAR can be invoked from message automation, by command originating at the PPT task or an optional task, or by using a labeled command originating in a command procedure. Use of the VAR stage outside of these environments results in message DSI290I and termination of the pipeline.

By contrast, the STEM stage reads and writes to variables within a stemmed array.

The $VAR stage command is the same as VAR, except that it also reads or writes the VIEW attribute variables (which start with $) that are associated with the specified data variables. When $VAR is the first stage, the color and highlighting specified in the attribute variables are translated to the output messages. When $VAR is not the first stage, the color and highlighting attributes specified in the input messages are translated to the attribute variables.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>
Termination Conditions

If specified as a first stage, VAR and $VAR terminate when the output stream is disconnected or when it finishes processing. If specified as a subsequent stage, VAR and $VAR terminate when the input stream is disconnected or when all variables are set and the output stream is disconnected.

Operand Descriptions

(COMMON)

Specifies that the common global variable dictionary is accessed instead of your personal variable dictionary.

(TASK)

Specifies that the task global variable dictionary is accessed instead of your personal variable dictionary.

name

Specifies the name of the variable to read-from or write-to. Do not include an ampersand (&) in the name (the ampersand is implied in the NetView command list language). The name length can be up to 11 characters in the command list environment and up to 31 characters in REXX and HLL except when $VAR is used. When $VAR is used, the limits are 10 and 30, respectively. Lowercase characters in the name are changed to uppercase before being processed. The &1–&31 variables as used in the NetView command list language are not supported for use in the VAR stage command. However, you can assign these values to, or from, other named variables, that you can use in the VAR stage command.

The amount of variable names is unlimited.

(number)

Specifies the number invocations (generations) to refer back when setting the variables. The number of generations refers to the current nesting level within the REXX, PL/I, or C calling sequence.

(Number) must be zero (0) or greater, and less than or equal to the existing number of generations. If (number) is greater than (0), the variables are in a generation preceding the current generation. The specified generation may precede the generation from which the PIPE command is issued, if such a generation exists.

The default for (number) is zero (0).

Usage Notes

The following applies to both VAR and $VAR stages:

• If (COMMON) or (TASK) is specified, VAR does not require the PIPE to be issued from a procedure.

• When VAR is the first stage command of a pipeline specification, the following conditions apply:
  – The maximum size of a message buffer output from the VAR stage is 32000 bytes. Message buffers exceeding 32000 bytes are truncated to 32000 bytes.
  – In the REXX environment, if the variable specified on the VAR stage command has not been initialized, the output value is the variable name.
  – For the NetView command list language and HLL environments, if the variable specified on the VAR stage command has not been initialized, the output value is a null message (an automation internal function request with a single, zero-length message buffer).
PIPE VAR and PIPE $VAR

- When VAR is not the first stage command of a pipeline specification, the following conditions apply:
  - The variable specified on the VAR stage command is initially dropped. Under REXX, this is equivalent to the REXX DROP function. Under REXX, the SYMBOL function indicates that the variable is of type LIT. In the NetView command list language and HLL, the variable is set to null and has a zero length. Therefore, if VAR is not the first stage command and is never called to process message buffers for a given pipeline, the value is dropped when the pipeline completes.
  - If the first input buffer to the VAR stage command is an MLWTO, only the first message line of the MLWTO is saved to the named variable, and the entire MLWTO is sent to the output stream.

Examples

Example: Writing to Named Variables
To select the first five data elements from the 'DATA.' stem and save them into variables A, B, C, D, and E respectively, run the following REXX COMMAND LIST.

```rexx
/* REXX COMMAND LIST */
'PIPE STEM DATA.'
 ' | VAR A B C D E'
```
PIPE VARLOAD

Syntax

VARLOAD:

Command Description

The VARLOAD stage is used to set values for variables that are passed in the input stream. The names and values of the variables set by VARLOAD are specified by the records passed on the primary input stream. VARLOAD sets one variable for each input message containing a character other than a blank or asterisk in the first position of the record. Messages beginning with a blank or asterisk are ignored. All other messages are treated as delimited strings.

The use of the VARLOAD stage command is limited to the command procedure environments (REXX, NetView command list language, and HLL). However, if the (COMMON) or (TASK) option is used, VARLOAD can be invoked from message automation, by command originating at the PPT task or an optional task, or by using a labeled command originating in a command procedure. Use of the VARLOAD stage outside of these environments results in message DSI290I and termination of the pipeline.

Data passed to VARLOAD on the input stream can be in one of two formats:

- `/variable1/value`
- `/variable1=variable2/value`

If `/variable1/value` is specified, the variable name following the delimiter is set to the value after the second delimiter. In the `/variable1=variable2/value` case, the current value of variable1 is compared to the value of variable2. If they are equal, variable1 is set to the value following the second delimiter. This is equivalent to the compare and swap OS/390 function.

Notes:

1. Variable1 is read from the dictionary specified by (COMMON), (TASK), or (number).
2. Variable2 is read from the local dictionary.
3. If variable1=variable2 is specified and is contained in a multiline message, all multiline message comparisons are done first. If any comparison fails, no variables are updated from the message data.

You can control which comparisons are grouped together using COLLECT and SEPARATE.

All messages from the input stream are also written to an output stream. If a secondary output stream is defined, the following is written to the secondary output stream:

- All input messages with errors in the variable name
PIPE VARLOAD

- If data is in the form \texttt{variable1=variable2}, all messages where \texttt{variable1} does not equal \texttt{variable2}

If no secondary output stream is defined, all messages are written unchanged to the primary output stream.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

VARLOAD terminates when the input stream is disconnected or when all variables are set and all defined output streams are disconnected.

Operand Descriptions

\textbf{(COMMON)}

\texttt{variable1} is accessed from the common global variable dictionary instead of the local variable dictionary.

\textbf{(TASK)}

\texttt{variable1} is accessed from the task global variable dictionary instead of the local variable dictionary.

\textbf{(number)}

Specifies the number invocations (generations) to refer back to when accessing \texttt{variable1}. The number of generations refers to the current nesting level within the REXX, PL/I, or C calling sequence. \texttt{ENVDATA} can be used to determine the number of generations.

\texttt{(Number)} must be zero (0) or greater and less than, or equal to, the existing number of generations. If \texttt{(number)} is greater than (0), the variables are in a generation preceding the current generation. The specified generation can precede the generation from which the PIPE command is issued.

The default for \texttt{(number)} is zero (0).

Usage Notes

- VARLOAD does not delay the stream.
- \texttt{Variable1} and \texttt{variable2} can be any valid REXX variable. (COMMON) and (TASK) variables can be a maximum of 32 characters in length.
- VARLOAD translates variable names to uppercase. Stem variable names will be translated to uppercase up to the first period.
- VARLOAD does not substitute symbols in the stem of a variable name specified as a compound symbol.
- To remove unwanted trailing blanks from input records before executing VARLOAD, use \texttt{STRIP} with the TRAILING keyword prior to VARLOAD.
- All data after the second delimiter will be assigned to the variable regardless of intervening blanks or subsequent delimiters.
Examples

Example: Setting Variables
To set the first five data elements in the 'DATA.' stem to the values A, B, C, D, and E, run the following REXX COMMAND LIST:

```
/* REXX COMMAND LIST */
'PIPE < MYDATA'
  '| VARLOAD'
```

Where MYDATA contains:

```
/DATA.1/A
/DATA.2/B
/DATA.3/C
/DATA.4/D
/DATA.5/E
```

Example: Comparing and Setting Variables
To set the first five data elements in the 'DATA.' stem to the values A, B, C, D, and E only when they currently contain the value contained in the variable STATUS, run the following REXX COMMAND LIST:

```
/* REXX COMMAND LIST */
'PIPE < MYDATA'
  '| VARLOAD'
```

Where MYDATA contains:

```
/DATA.1=STATUS/A
/DATA.2=STATUS/B
/DATA.3=STATUS/C
/DATA.4=STATUS/D
/DATA.5=STATUS/E
```

Example: Copying Task Globals
In the following example, task global values are copied from the target task to the local dictionary. The copied values are then output to the console.

```
/* VARLOAD Example: Copy task globals from target task */
arg opid
  IF opid = '' THEN opid = 'TOM'

  'PIPE (NAME COPYTGLB)',
  '| CORRCMD /"opid": QRYGLOBL TASK VARS=TCP*', /* get data from "opid"*/
  'SEPARATE',
  'LOCATE 1.7 /BNH039I/", /* just the DATA please*/
  'EDIT "/" NEXT', /* change format to suit*/
    'WORD 2 NEXT',
    "/*/" NEXT',
    "45.* NEXT',
  'CONSOLE', /* show the records to the nice folks*/
  'VARLOAD TASK' /* put them in my dictionary*/

  'QRYGLOBL TASK VARS=TCP*' /* confirm update*/
```

exit

The console output might look like this:

```
* NTV7E COPYTCP TOM
  NTV7E /TCPLSOCK/*..2206
  NTV7E /TCPADDR/9.67.50.1
  NTV7E /TCPUSER/NV65
  NTV7E /TCPSTACK/TCP32
  NTV7E
```
Example: Update Current Group Members

The following will update the current group (currGrp) members if the current group has not been updated by another task.

/* VARLOAD example: Update members for "current group"

Problem: Value of "currGrp" can be changed at any time by some other task. */

'GLOBALV GETC currGrp'
say 'Current group name is' currGrp

'PIPE (NAME BLDGRP)', /* Find the members of this group */
' | NETV LIST ASSIGN=GROUP, GROUP="currGrp", /* LIST members... */
' | LOCATE 1.6 /DSI640/ 1.6 /DSI641/ ', /* isolate data lines*/
' | EDIT 22.* 1', /* remove "headers" */
' | JOINCONT //', /* make into one line*/
' | VAR MEMBERS' /* ...and save */

UpDate.1 = '/GRPop/' || members /* format for VARLOAD*/
UpDate.2 = '/currGrp=currGrp/' || currGrp

/* Update.2 contains multiple references to currGrp. CurrGrp is being used in */
/* three ways: the first references the common global dictionary (see */
/* option on VARLOAD below), the second references the local dictionary */
/* and the third is resolved immediately by REXX. */

/* Since this is a COMMON GLOBAL, we check to be sure that other */
/* task did not change CURRGRP while we were working... */

/* COLLECT is important in the following PIPE. */
'PIPE (NAME SETGRP END %)', /* */
' | STEM UpDate.', /* */
' | COLLECT', /* COLLECT IMPORTANT! */
' | A: VARLOAD COMMON', /* */
' | EDIT "Update successful:" 1 WRITELINE COPY ', /* */
' | CONSOLE', /* */
' | % A:', /* */
' | EDIT "Entire update failed:" 1 WRITELINE COPY', /* */
' | CONSOLE' /* */

'GLOBALV GETC currGrp'
say 'After process currGrp is' currGrp

If the update failed because the current group was updated by another task, the output is similar to:
Current group name is +FIRST
ENTIRE UPDATE FAILED:
/GRPopS/ TOM MARK NETOP1 OPER1 NETOP2
/currGrp=currGrp/+FIRST
After process currGrp is' +SECOND

When the comparison at line 2 fails, the entire multiline message is not updated.
PIPE VET

Syntax

VET (first stage):

VET  NEXT  ROWS  NAME only one
     CURRENT  FIELDS  NAME attach name

VET (stage other than first):

VET  cursor position  ENTER  NAME only one
     ROW.COL  action key  NAME attach name

VET (command):

VET  cursor position  ENTER
     ROW.COL  /string/  action key

     NAME only one

     NAME attach name

Synonyms

<table>
<thead>
<tr>
<th>Stage Command</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>VET</td>
<td>VOSTIO</td>
</tr>
<tr>
<td>CURRENT</td>
<td>C</td>
</tr>
<tr>
<td>FIELDS</td>
<td>FIELD, F</td>
</tr>
<tr>
<td>NEXT</td>
<td>N</td>
</tr>
<tr>
<td>ROWS</td>
<td>ROW, R</td>
</tr>
</tbody>
</table>

Command Description

The VET stage is used to read data from, and subsequently write data to, a virtual screen belonging to a virtual OST (VOST).

When used as a first stage, VET obtains data from the VOST in one of the following forms:
- Row
- Field
- Message

Row and field form data is returned in message BNH150I. If a command issued on the VOST does not result in a full-screen being presented on the virtual screen, the
message displayed on the VOST is returned to VET in message form. If the
application running on the VOST returns a message and a full screen, both the
message and full-screen data are returned to VET.

When VET is used as a command or as a subsequent stage, VET writes data to the
virtual screen belonging to the VOST. Data is written in the form of simple text
where one line is written for each input-capable field on the VOST.

For more information about the VET stage, see "Chapter 5. Full-Screen
Automation" on page 267.

### Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

### Termination Conditions

If specified as a first stage, VET terminates when the output stream is disconnected
or when it finishes processing its output. If specified as a subsequent stage, VET
terminates when the input stream or the output stream is disconnected.

### Operand Descriptions

**action key**

Specifies the action key to be sent with the data. Any of the following keys are valid:

- PF1
- PF2
- PF3
- PF4
- PF5
- PF6
- PF7
- PF8
- PF9
- PF10
- PF11
- PF12
- PF13
- PF14
- PF15
- PF16
- PF17
- PF18
- PF19
- PF20
- PF21
- PF22
- PF23
- PF24
- PA1
- PA2
- PA3
- ENTER
- CLEAR
- NOKEY

After the data specified in the input stream and */string/* are written to the
virtual screen, the action key is passed to the application running on the VOST.
The application responds as if the */string/* data was entered and pressed the
designated action key on a terminal was pressed.

Unless another action key is specified, the default ENTER is sent to the
application with the data.

NOKEY is a special action key. NOKEY indicates that the data specified in
*string/* is to be written to the virtual screen, but an action key is not to be
pressed. This is as though a user enters data on a panel and does not press
Enter, a PF, or PA key.

**CURRENT**

Specifies that the virtual screen image on the VOST at the time of the call is to
be returned to the stage. CORRWAIT is not required with VET CURRENT. Any
pending I/O requests sent by the application running on the VOST are applied
to the virtual screen before returning the screen image to the stage.
PIPE VET

Note: The entire screen image is returned to the stage in message BNH150I.

FIELDS
Specifies that one line for each field on the virtual screen follows the BNH150I message header.

NAME
Specifies the name of the VOST. NAME must correspond to the NAME on the ATTACH command that created the VOST or attach name. VET NAME indicates that the VET stage is to interact with the named VOST.

If a VOST was created by the ATTACH command without a NAME, the VOST is dependent on the invoking procedure. In the case of a dependent ATTACH, if you code VET without specifying a NAME, VET interacts with only one VOST; the VOST created by the ATTACH command within the same procedure as the VET stage.

NAME must be specified if VET is to interact with an independent VOST; a VOST created outside of the procedure family.

NEXT
Specifies that the next update to the virtual screen is to be returned to the stage. This update can either be currently pending or can be received at a future point in time when the application next updates the virtual screen. CORRWAIT should be the next stage after VET NEXT. CORRWAIT automatically ends when:
- The application running on the VOST is ready for input.
- The application terminates.

For additional information on CORRWAIT, see "PIPE CORRWAIT" on page 43.

In general, VET NEXT does not return a complete screen image. Only the parts of the virtual screen sent by the application as the screen updates are returned. Depending on the application, all or part of the virtual screen may be updated.

ROWS
Specifies that the data displayed on the virtual screen is to be returned to the stage as a series of 24 lines of 80 characters each following the BNH150I message header. Positions on the virtual screen occupied by start field orders are blank (X'40'). When using VET NEXT ROWS, those screen positions not updated by the application running on the VOST will contain X'FF' characters.

For additional information about start field orders, refer to the 3270 Information Display System library.

ROW.COL
Specifies the starting row and column on the virtual screen where the data specified by the input stream or /string/ is to be written. If ROW is specified without COL, the default value of 1 is used for COL.

If ROW.COL is not specified, VET writes the data specified in the input stream /string/ beginning at the current cursor position on the VOST virtual screen. If the current cursor position is in a protected field, VET simulates a tab to the next unprotected field and writes the data beginning in that unprotected field.

A null string (/ /) is handled as a tab to the next unprotected field. By using null strings you can tab through the unprotected fields on the virtual screen, filling in data as you proceed. If you specify more tabs than unprotected fields on the virtual screen, you will tab back to the first unprotected field on the screen and continue with your data input.
All pending application I/O requests are applied to the virtual screen before writing the /string/ to the virtual screen.

/string/
Specifies the data to be written to the virtual screen.
/string/ is only valid when VET is used as a command.

Data that is too long for the unprotected field is truncated. When the data is truncated, no error condition or warning is returned to the stage.

Data that is shorter than the unprotected field is padded on the right with blanks.

The first nonblank character encountered after the stage name or row.col is the delimiter, which establishes the boundary of the text string used by the stage. The delimited string ends when the same character is encountered a second time.

Multiple unprotected fields on the virtual screen can be filled by including null values for /string/. A null value is indicated by coding two delimiters consecutively, for example:

//

A null string causes nothing to be written to the unprotected field, but the cursor tabs to the next field on the virtual screen. In this way you can input data to some fields and skip other fields. If you specify more tabs than unprotected fields on the virtual screen, the cursor tabs back to the first unprotected field on the screen and continues with your data input.

Usage Notes

- While the application is still locked from accepting input, VET enables you to queue input for the application. Queued input assumes that the application will accept your input without any intervening errors. You can queue as much input as necessary, but the chance for error increases dramatically with each queued input request. If the application ends before all queued input has been passed to the application, the remaining queued input is discarded without generating an error or warning.

- If PIPE VET NEXT is issued while queued inputs are pending, the results of PIPE VET NEXT are not returned to the stage until all pending queued inputs have been passed to the application and inputs have been processed by the application and the results displayed on the VOST virtual screen.

- A null string (//) need not be specified on VET. If you want to send a PF3 action key to the application running on the VOST without altering any fields on the virtual screen, you can specify the following:

  VET PF3

- If VET has both an input stream and a /string/ specified, the /string/ is written to the panel first, followed by data from the input stream.

- PF and PA keys cannot be specified for action key if the application running on the VOST allows user-defined PF keys. The BNH150I application field contains the application name if the application enables the user to define PF keys. If BNH150I contains a value in the application field, NetView rejects any VET command with a PF or PA action key. Instead of specifying a PF or PA key, place the required command in /string/ and use ENTER for the action key. For example, code NETVIEW VET /RETURN/ ENTER instead of VET PF3.

- After a VOST terminates, NetView retains the last data transmitted by the application that was running on the VOST for up to 5 minutes.
obtains this data as though the application was still active. After the last transmitted data has been obtained by VET, an additional VET NEXT obtains no data and a VET CURRENT returns a blank screen.

- VTAM commands are not supported on a VOST. The MVS command must be used to issue VTAM commands on a VOST.
- The line count attribute is set in BNH150I for VET ROWS output. This line count value can only be used by the EDIT pipe stage.

The BNH150I label line has LINECOUNT=0 and all other lines are numbered 1 through 24 corresponding to their line number on the virtual screen. An example use for the line count data is in determining the line where certain data was found. This can be done in pipeline processing using LOCATE and the line count.

Examples

**Example: Request Current Screen**
The following VET stage specification requests the current view of the application panel running on the VOST named MYVOST. The screen data is to be returned in ROWS format.

```
PIPE VET CURRENT ROWS NAME MYVOST | ...
```

**Example: Writing Data to a VOST**
This example writes FORWARD to the command line, which in the application running on VOST MYVOST is found at row 24 column 8. The default action key ENTER is assumed.

```
VET 24.8 /FORWARD/ NAME MYVOST
```

**Example: Writing Multiple Lines to a VOST**
Null values for `/string/` can be used to tab through fields. The following example shows a stem variable containing five values. The first four are null and the fifth contains an X. The following REXX fragment places nothing in the first four unprotected fields, an X in the fifth, and uses Enter for the action key.

```rexx
input. = '
input.5 = 'X'
input.0 = 5
```

```
PIPE (NAME REQNCP)
   STEM input.
   VET 1.1
```
PIPE VTAM

Syntax

VTAM:

\[
\text{VTAM} \left( \text{ECHO NOPANEL MOE} \right) \text{ cmdtext}
\]

Command Description

The VTAM stage command runs the VTAM commands DISPLAY, VARY, and MODIFY in a local or remote domain. The VTAM stage command supports implicit and explicit routing for resources in remote domains, in addition to resources in your local domain. *Explicit routing* means that the command’s operand or resource name contains a NetView domain name delimited by a period. For example, DISPLAY NET,CNM02.APPLS is taken to request a display of applications at domain CNM02. *Implicit routing* refers to those resource names defined on an RRD statement in CNMSTYLE as belonging to a specific domain.

The VTAM stage can be used anywhere in the pipeline. If the VTAM stage has an argument, it must be the first stage. If the VTAM stage has no argument, it cannot be the first stage and any input stream other than DISPLAY, VARY, or MODIFY commands causes the following message to be inserted into the output stream:

DSI0711 INVALID VTAM COMMAND

The VTAM stage arranges that the same timeout and termination conditions that are specified for a pipeline apply when running the VTAM commands at the remote domain. The following conditions are considered transferable:

- TOSTRING
- TAKE (first)
- NLOCATE
- LOCATE
- CORRWAIT

Multiple conditions can be transferred, if no other stage intervenes.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

When specified as a first stage, the VTAM stage terminates when it finishes processing its output. As a subsequent stage, the VTAM stage terminates when the input stream disconnects.
Operand Descriptions

```
cmdtext
This is the command for VTAM. Only DISPLAY, MODIFY, and VARY are supported.

If VTAM is the first stage, `cmdtext` is required.

If VTAM is not the first stage, `cmdtext` must not be used. The VTAM stage extracts the first line of a message in the input stream as the command and additional lines, if any, as the data to be processed by that command. The VTAM stage is run once for each message delivered by the previous stage. Every time the command runs, the input message becomes the current message during the processing, and is then discarded.

Additional messages in the input stream are treated in the same way. If the command is not a supported VTAM command (DISPLAY, VARY, or MODIFY), the error message DSI071I INVALID VTAM COMMAND is inserted into the output stream.
```

ECHO
When ECHO is specified, the text of the command itself is written to the pipeline before the command is executed.

MOE
Message on error (MOE) specifies to examine the return code from the command and, if the return code is not zero, insert message DWO369I containing the return code into the stream, after any messages the command could have returned.

For local resources the return codes are those documented for the commands: DISPLAY, VARY and MODIFY. If the RMTCMD command on the VTAM stage is used to access remote resources, the return codes are those documented for RMTCMD.

NOPANEL
When NOPANEL is specified, the command will not display a full-screen panel. If it attempts to do so, message BNH113W will be inserted into the pipeline and the command will receive an I/O error code from NetView presentation services.

Usage Notes

- Command authorization checking applies to all commands invoked using the VTAM stage.

Return Codes

The following return codes are valid only when the MOE operand is used:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>Installation exit 03 generated USERDROP.</td>
</tr>
<tr>
<td>-500 to -599</td>
<td>Failure attempting to call installation exit 03. Please report specific return code to the Tivoli Customer Support.</td>
</tr>
<tr>
<td>-108</td>
<td>Command is type=I or type=P.</td>
</tr>
<tr>
<td>-112</td>
<td>Command search failure, usually because the command is too long.</td>
</tr>
<tr>
<td>-116</td>
<td>ACCESS not authorized. Command authorization restrictions prevent processing.</td>
</tr>
</tbody>
</table>
There are other possible return codes indicating storage failure. The code you get depends upon the processing phase when storage failure was detected. Look for DSI124I at the system console for this condition.

Examples

Example: Issuing a VTAM Command at a Remote Domain
Suppose you want to analyze the status of applications at (remote) domain CNM02. If you want to allow up to 10 seconds between the messages that constitute the response but you do not want to see the IST097I DISPLAY ACCEPTED message, your command procedure should issue:

```
PIPE VTAM D NET,CNM02.APPLS
  CORRWAIT 60
  NLOCATE 1.7 /IST097I/
  TAKE 1
  SEPARATE
  LOCATE /CONCT/
  STEM appldata
```

Note: In a REXX command procedure, the last stage could be STEM appldata. with a period (.)

If a RMTCMD session between your station and CNM02 does not already exist, then one is started by this command. The VTAM stage sends your D NET,APPLS to the remote domain. Moreover, the VTAM stage arranges that the same timeout (60 seconds) and selection conditions (NLOCATE and TAKE) apply at the remote domain. The transfer of stages for application at the remote domain ended when the SEPARATE stage, a nontransferable stage, was encountered.

The 60-second timeout is not excessive, because the termination condition (TAKE 1) ends the wait (on both domains) when a message (any message except IST097I) is received.
PIPE XLATE

Syntax

XLATE:

The XLATE stage accepts a message from its input stream, translates specified characters to other characters, and writes the message to its output stream.

Use XLATE to translate:
• Uppercase letters to lowercase letters
• Lowercase letters to uppercase letters
• ASCII characters to EBCDIC characters
• EBCDIC characters to ASCII characters

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>1</td>
</tr>
<tr>
<td>Output</td>
<td>1</td>
</tr>
</tbody>
</table>

Termination Conditions

The XLATE stage terminates when either its input stream or output stream disconnects.

Operand Descriptions

position.length
Specifies the character position where translation begins and the length from that point that translation occurs. The default is 1.*, which means start at the first position and continue to the end.

UPPER
Specifies that the standard 26 Latin letters are translated to uppercase.

A2E
Specifies that ASCII characters are translated to EBCDIC. ASCII code set ISO 8859-1 and EBCDIC code set IBM-1047 are used.

COMMON
Specifies that EBCDIC character codes that are not common to all code sets are translated to X'5C' (asterisk), except X'00' (null) and X'FF' (EO), which are translated to X'40' (blank).
E2A
Specifies that EBCDIC characters are translated to ASCII. EBCDIC code set IBM-1047 and ASCII code set ISO 8859-1 are used. It is recommended that you use XLATE COMMON to remove format control characters (for example, line feed, new line, and end of file) before you translate host messages to ASCII.

LOWER
Specifies that the standard 26 Latin letters are translated to lowercase.

Examples

Example: Removing Characters Using PIPE XLATE COMMON
The following is an example of using PIPE XLATE COMMON to remove characters that cannot be translated before converting EBCDIC text to ASCII text:

```plaintext
'PIPE NETV LIST STATUS=TASKS',
' | XLATE COMMON',
' | XLATE E2A',
' | NETV SOCKET TYPE=SEND SOCKID=0',
' | WAIT 5',
' | CONS'
```

Example: Translating Text to ASCII Prior to Using the SOCKET Command
The following is an example of translating text to ASCII prior to using the SOCKET command:

```plaintext
'PIPE (NAME SENDto0)', /* send on socket ID 0 */
' | VAR data_string',   /* EBCDIC data to send */
' | XLATE E2A',         /* convert to ASCII */
' | NETV SOCKET TYPE=SEND SOCKID=0 ', /* send converted data */
' | WAIT MOE 5',        /* wait for result */
' | STEM msgsock.'      /* msgs about send */
```

Example: Translating an ASCII Value to EBCDIC
The following sample PIPE command translates the data taken from the `linedata` variable from ASCII to EBCDIC and stores the translated data back into the `linedata` variable:

```plaintext
'PIPE | VAR linedata | XLATE A2E |VAR linedata'
```
PIPE < (From Disk)

Syntax

(From Disk):

```
< DSIPARM. ddname.* member INCL DISKONLY
```

Command Description

The < (From Disk) stage reads data from DASD into the pipeline. The records read from DASD are single-line messages in the pipeline.

A return code indicating the success or failure of the stage is passed to the secondary output stream if one is connected. These return codes are described below. If a secondary output stream is connected, failure (such as no such member) does not terminate the pipeline or cause error messages.

Note: The < (From Disk) stage sets the buffer’s origin field (HDRDOMID) to be the member name. For example, if reading a panel member and then displaying data from the panel, the HDRDOMID will then contain the name of the panel.

Streams

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Number Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>0</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
</tr>
</tbody>
</table>

Termination Conditions

The < (From Disk) stage terminates when end-of-file is reached or when the primary output stream is disconnected.

Operand Descriptions

* Specifies that NetView is to search all standard DDNAMEs for the specified member name. The following libraries are searched, if allocated, in the following order:

1. DSICLD
2. DSIPARM
3. DSIPRF
4. DSIVTAM
5. DSIMSG
6. CNMPNL1
7. BNPJNL1
8. BNPJNL2
9. DSILIST
10. DSIOOPEN
11. DSIASRC
PIPE < (From Disk)

12. DSIARPT

ddbname
Specifies the name of a standard NetView DDNAME, such as DSIPARM or DSICLD, from where to read the member. Refer to the BROWSE command help for a list of valid DDNAMEs. When ddbname is not specified, the default is DSIPARM. When specifying ddbname, a period (.) is used to separate it from the member name. Do not use spaces before or after the period.

member
Specifies the 1–8 character name of the member or file to be read (parameter synonyms are not supported). This name is a member of the data set concatenation associated with the ddbname being used.

INCL
Specifies that %INCLUDE are expanded when the member or file is read.

DISKONLY
Indicates that any member loaded by the INSTORE stage should be ignored.

Usage Notes

- The < (From Disk) stage must be the first stage command.
- Access security for the < (From Disk) stage is provided through the READSEC command. Refer to the Tivoli NetView for z/OS Administration Reference for information about the READSEC command.

Return Codes

The following return codes are returned to the secondary output stream if one is connected, as signed 10-digit decimal numbers:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful completion.</td>
</tr>
<tr>
<td>12</td>
<td>Authorization problem.</td>
</tr>
<tr>
<td>32</td>
<td>DSIDKS macro failed trying to CONNECT or FIND.</td>
</tr>
<tr>
<td>40</td>
<td>Invalid INCLUDE operation (%INCLUDE record normally returned unchanged to primary output stream).</td>
</tr>
<tr>
<td>Other</td>
<td>Refer to return codes documented for the DSIDKS assembler macro. Refer to Tivoli NetView for z/OS Customization: Using Assembled for more information about DSIDKS return codes.</td>
</tr>
</tbody>
</table>

Examples

**Example: Reading the Contents of a File into a Pipeline**

To display the contents of the DSICMD file as a multiline message:

```
PIPE < DSICMD
    | COLLECT
    | CONSOLE
```

This example reads lines of the DSICMD member into the pipeline. The COLLECT stage builds these lines into a multiline message. When all lines are read from DSICMD and collected, the CONSOLE stage displays the multiline message to the console.

**Example: Counting Comment Lines in a File**

Analyze the lines of the CNMSTYLE member.
/* REXX sample command list */
'PIPE < DSIPARM.CNMSTYLE INCL',
  '  STEM A.',
  '  LOCATE 1.1 /*/',
  '  STEM B.'
Say 'There are 'A.0' lines in CNMSTYLE'
Say 'of which 'B.0' are comment lines.'

This example reads the lines of the CNMSTYLE member under the DSIPARM ddname. The contents of the expanded member are saved in a stem variable named A, and the comment lines are located and saved to a stem variable named B. The messages indicate how many lines were read from the CNMSTYLE expanded member and how many of those lines are comment lines.
Chapter 3. NetView Pipelines Device Drivers

This chapter documents general-use programming interface and associated guidance information. For information about using pipelines in high-level languages, refer to the Tivoli NetView for z/OS Customization: Using PL/I and C.

Device drivers are stage commands that move data between your pipeline and other system resources (such as command procedure variables, DASD, keyboards, displays, and so on).

When using drivers that can be placed anywhere in a pipeline (such as, STEM, VAR, and SAFE) be aware that they work differently depending on where they are placed. 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.

Attention: You can overwrite or destroy data when you misplace these device drivers.

Interfacing with the Task: CONSOLE, HELDMSG, LITERAL, LOGTO

This section describes several device drivers that interface with the task. You can:
- Display pipeline contents on the screen (CONSOLE)
- Route pipeline contents to another pipeline (CONSOLE)
- Copy held messages from your operator console (HELDMSG)
- Insert text into the pipeline (LITERAL)
- Copy pipeline contents to a specified log (LOGTO)

Displaying Messages: CONSOLE

The CONSOLE stage enables the user to:
- Display messages on the screen while these messages remain in the pipeline for use by the next stage.
- Remove the status of held messages that are in the pipeline before rewriting them on the screen (using the DELETE option).
- Return messages to its caller (without displaying the messages), when it is a stage of the inner pipeline as part of a PIPE-within-a-PIPE structure.

Example 1: Displaying Results While Avoiding Logging

This example shows how to use the CONSOLE stage with the ONLY option to display messages in the pipeline without logging or exposing the messages:

```plaintext
PIPE NETVIEW LIST '' | CONSOLE ONLY
```

Output from the pipeline follows:
Processing steps:

1. The NETVIEW stage invokes the LIST command and places the corresponding response messages in the pipeline.

2. The CONSOLE ONLY stage reads the messages and displays them on the operator console, but does not expose the messages for automation or logging.

Note: The PIPE command is echoed in the log although the results of the PIPE command are not logged.

Example 2: Deleting Held Messages

The next example shows how to use the CONSOLE stage with the DELETE option to release the held status of a message on the operator’s screen:

```
PIPE HELDMSG | CONSOLE DELETE
```

Output showing existing held messages looks like this:

```
NCCF NETVIEW CNM01 OPER6 04/14/99 15:06:33
* CNM01 IEI04I 15.05.33 99.104 ACTIVITY 973
  JOBS M/S TS USERS SYSSAS INITS ACTIVE/MAX VTAM
  00000 00007 00001 00014 00002 00001/00300
  LLA LLA LLA NSW S VLF VLF VLF NSW S
  JES2 JES2 IEPROC NSW S MYVTAM MYVTAM VTAM NSW S
  TSO TSO TCAS OWT S MYESSI MYESSI NETVIEW NSW S
  MYENV MYENV NETVIEW NSW S
  USER2 OWT
* CNM01 MVS D A,L
```

Output from the pipeline looks like this:

```
NCCF NETVIEW CNM01 OPER6 04/14/99 15:07:00
* CNM01 IEI04I 15.05.33 99.104 ACTIVITY 973
  JOBS M/S TS USERS SYSSAS INITS ACTIVE/MAX VTAM
  00000 00007 00001 00014 00002 00001/00300
  A LLA LLA NSW S VLF VLF VLF NSW S
  S2 JES2 IEPROC NSW S MYVTAM MYVTAM VTAM NSW S
  0 TSO TCAS OWT S MYESSI MYESSI NETVIEW NSW S
  ENV MYENV NETVIEW NSW S
  R2 OWT
* CNM01 PIPE HELDMSG | CONSOLE DELETE
```

Processing steps:

1. The HELDMMSG stage reads the held message queue and writes a copy of it to the output stream.

2. The CONSOLE DELETE stage resets the hold status of the messages and writes them back on the screen.
Notice that the vertical bar (|) has replaced the first characters of each line, indicating that the message is no longer held. When the user presses ENTER the message will disappear.

Example 3: Multiple CONSOLE Stages

This example shows how the insertion of multiple CONSOLE stages into the pipeline affects output. It illustrates how the CONSOLE stages handle their input streams, processing messages as they receive them. If single-line messages are processed by multiple CONSOLE stages there is no way to predict in what order the messages by one CONSOLE stage interfaces with messages written by other CONSOLE stages. This example should be studied in conjunction with Example 4 which demonstrates how adding the COLLECT stage to gather the pipeline messages into a multiline message prior to a CONSOLE stage modifies the screen output.

This example shows how the insertion of the CONSOLE stage between other stages affects output from the pipeline.

PIPE LITERAL ? This is the CCC message ?
| CONSOLE
| LITERAL / This is the BBB message /
| CONSOLE
| LITERAL ! This is the AAA message !

Output from the pipeline looks like this:

```
NCCF NETVIEW CNM01 OPER5 02/01/99 10:40:10
* CNM01 PIPE LITERAL ? THIS IS THE CCC MESSAGE ? | CONSOLE | LITERAL / THIS IS THE BBB MESSAGE / | CONSOLE | LITERAL ! THIS IS THE AAA MESSAGE ! | CONSOLE
| CNM01 THIS IS THE CCC MESSAGE (written by console stage # 1)
| CNM01 THIS IS THE BBB MESSAGE (written by console stage # 2)
| CNM01 THIS IS THE AAA MESSAGE (written by console stage # 3)
| CNM01 THIS IS THE CCC MESSAGE (written by console stage # 2)
| CNM01 THIS IS THE BBB MESSAGE (written by console stage # 3)
| CNM01 THIS IS THE CCC MESSAGE (written by console stage # 3)
```

Processing steps:
1. The first LITERAL stage writes the CCC message to the pipeline.
2. The first CONSOLE stage reads the CCC message and displays it on the screen. The message remains in the pipeline.
3. The second LITERAL stage writes the BBB message to the output stream in front of the CCC message in the stream.
4. The second CONSOLE stage reads the BBB message in its input stream and writes it on the screen. The message also remains in the pipeline.
5. The third LITERAL stage writes the AAA message to the output stream in front of the BBB and CCC messages in the stream.
6. The third CONSOLE stage reads the AAA message and writes it on the screen. It also remains in the pipeline, although there are no additional stages to process it.
7. The second CONSOLE stage reads the CCC message in its input stream and writes it on the screen. The message also remains in the pipeline.
8. The third CONSOLE stage reads the BBB message in its input stream and writes it on the screen. The message also remains in the pipeline, although there are no additional stages to process it.
9. The third CONSOLE stage reads the CCC message in its input stream and writes it on the screen. It also remains in the pipeline, although there are no additional stages to process it.

Example 4: Using Multiple CONSOLE Stages with COLLECT

This example shows how to modify the previous example by using the COLLECT stage preceding each CONSOLE stage. COLLECT gathers single-line messages into an MLWTO, which affects the structure of the output:

```
PIPE LITERAL ? This is the CCC message ?
  | COLLECT
  | CONSOLE
  LITERAL / This is the BBB message /
  | COLLECT
  | CONSOLE
  LITERAL ! This is the AAA message !
  | COLLECT
  | CONSOLE
```

Output from the pipeline looks like this:

```
+ CNM01 * CNM01
  | CNM01 | CNM01
  | CNM01 | CNM01
  | CNM01
  THIS IS THE CCC MESSAGE
  THIS IS THE CCC MESSAGE
  (written by console stage # 1)
  (written by console stage # 2)
  THIS IS THE BBB MESSAGE
  THIS IS THE CCC MESSAGE
  (written by console stage # 3)
```

Copying Held Messages into the Pipeline: HELDMSG

The HELDMSG stage enables the user to copy messages from the operator’s held message queue into the pipeline.

Example 1: Routing Held Messages

This example shows how to use the HELDMSG stage to copy held messages into the pipeline and route them to another operator, OPER3:

```
PIPE HELDMSG | NETVIEW MSGROUTE OPER3
```

The existing held message at ORIGOPER’s screen follows:

```
NCCF NETVIEW CNM19 ORIGOPER 05/17/99 10:18:43
```

Output from the pipeline on OPER3’s screen looks like this:

```
NCCF NETVIEW CNM19 OPER3 05/17/99 10:26:46
```

Processing steps:

1. The HELDMSG stage writes a copy of the held message queue to the pipeline.
2. The NETVIEW stage reads the pipeline messages and uses them as input to the MSGROUTE command which sends a copy of the held messages to OPER3’s screen, where they are held also. The number of messages in the pipeline affects how many times the NETVIEW stage runs. In this case, one message is in the pipeline.

**Note:** The message that is displayed on OPER3’s screen is only a copy and the original held message is still displayed on ORIGOPER’s console.

### Example 2: Deleting Held Messages

This example shows how to use the HELDMSG stage with the CONSOLE stage to delete held messages from an operator’s screen:

```plaintext
PIPE HELDMSG | CONSOLE DELETE
```

Output showing existing held messages follows:

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW</th>
<th>CNM01 OPER6</th>
<th>03/20/99 15:06:33</th>
</tr>
</thead>
<tbody>
<tr>
<td>* CNM01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEE1041</td>
<td>15.05.33</td>
<td>99.079</td>
<td>ACTIVITY 973</td>
</tr>
<tr>
<td>JOBS</td>
<td>M/S</td>
<td>TS USERS</td>
<td>SYSAS</td>
</tr>
<tr>
<td>00000</td>
<td>00007</td>
<td>00001</td>
<td>00014</td>
</tr>
<tr>
<td>LLA</td>
<td>LLA</td>
<td>NSW</td>
<td>VLF</td>
</tr>
<tr>
<td>JES2</td>
<td>JES2</td>
<td>IEFPROC</td>
<td>NSW</td>
</tr>
<tr>
<td>TSO</td>
<td>TSO</td>
<td>TCAS</td>
<td>OWT</td>
</tr>
<tr>
<td>ENV</td>
<td>ENV</td>
<td>NETVIEW</td>
<td>NSW</td>
</tr>
<tr>
<td>USER2</td>
<td>OWT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* CNM01</td>
<td>MVS</td>
<td>D</td>
<td>A,L</td>
</tr>
</tbody>
</table>

Output of the PIPE command follows:

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW</th>
<th>CNM01 OPER6</th>
<th>02/01/99 15:07:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>* CNM01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1041</td>
<td>15.05.33</td>
<td>99.032</td>
<td>ACTIVITY 973</td>
</tr>
<tr>
<td>BS</td>
<td>M/S</td>
<td>TS USERS</td>
<td>SYSAS</td>
</tr>
<tr>
<td>00</td>
<td>00007</td>
<td>00001</td>
<td>00014</td>
</tr>
<tr>
<td>* CNM01</td>
<td>PIPE HELDMSG</td>
<td>CONSOLE DELETE</td>
<td></td>
</tr>
</tbody>
</table>

**Processing steps:**

1. The HELDMSG stage writes a copy of the held message queue to the output stream.
2. The CONSOLE DELETE stage resets the hold status of messages in the pipeline.

Notice that the vertical bar (|) has replaced the first characters of each line, indicating that the message is no longer held.

### Inserting Text into the Pipeline: LITERAL

The LITERAL stage enables the user to insert text into the pipeline.

### Example 1: Inserting Text into the Pipeline
This example shows how to use the LITERAL stage to add a message to the pipeline and then display it.

```
PIPE LITERAL % JACK BE NIMBLE % | CONSOLE
```

Output from the pipeline using the LITERAL stage follows:

```
NCCF NETVIEW CNM01 OPER5 03/01/99 09:13:24
* CNM01 PIPE LITERAL % JACK BE NIMBLE % | CONSOLE
| CNM01 JACK BE NIMBLE
```

Processing steps:
1. The LITERAL stage writes the JACK BE NIMBLE text string to the output stream.
2. The CONSOLE stage reads its input and displays the message.

**Example 2: Inserting Text Containing Command List Functions**

This example, in a REXX command list named DISPOPID, shows how to use the LITERAL stage to add a message containing a REXX function to the pipeline:

```
/* REXX COMMAND LIST - DISPLAY OPERATOR ID */
'PIPE LITERAL !My OPID is!', /* Add text to pipe */
   OPID('!'), /* Get my operator ID */
   ' | CONSOLE' /* Display to terminal */
EXIT
```

Output from DISPOPID follows:

```
NCCF NETVIEW CNM01 OPER5 03/26/99 09:15:10
* CNM01 DISPOPID
| CNM01 MY OPID IS OPER5
```

Processing steps:
1. The LITERAL stage writes the MY OPID IS text string to the output stream along with the REXX OPID function results.
2. The CONSOLE stage reads its input and displays the messages on the screen.

**Example 3: Inserting Multiple Text Strings**

This example shows how to use the LITERAL stage to add multiple text strings to a pipeline:

```
PIPE LITERAL ? This is the CCC message ?
   LITERAL / This is the BBB message /
   LITERAL ! This is the AAA message !
   CONSOLE
```

Output from the pipeline follows:

```
NCCF NETVIEW CNM01 OPER5 05/02/99 10:40:10
* CNM01 PIPE LITERAL ? THIS IS THE CCC MESSAGE ? | LITERAL / THIS IS THE BBB MESSAGE / | LITERAL ! THIS IS THE AAA MESSAGE ! | CONSOLE
| CNM01 THIS IS THE AAA MESSAGE
| CNM01 THIS IS THE BBB MESSAGE
| CNM01 THIS IS THE CCC MESSAGE
```

Processing steps:
1. The first LITERAL stage writes the CCC message text string to the output stream.
2. The second LITERAL stage writes the BBB message text string to the output stream in front of the CCC text already in the stream.

3. The third LITERAL stage writes the AAA message text string to the output stream in front of the BBB which is in front of the CCC message.

4. The CONSOLE stage reads its input and displays the messages on the screen.

Copying Pipeline Contents to a Log: LOGTO

The LOGTO stage enables the user to send a copy of the pipeline contents to a specified log. The contents also remain in the pipeline for processing by the next stage.

You can use any of several options to control the logging destination. The NETLOG, SYSLOG and HCYLOG options identify the NetView, system and hardcopy logs respectively. Messages in the pipeline are sent to the specified log regardless of how the system defaults and overrides options are set. The ALL option indicates that messages in the pipeline should be sent to all the logs.

The asterisk (*) option indicates that messages are logged consistently with how the NetView system defaults and overrides are set.

Example: Logging Output from an MVS Command

This example shows how to use the LOGTO stage to log information to both the NetView log and the system log:

```
PIPE NETVIEW MVS D A,L | CORRWAIT 5 | LOGTO NETLOG SYSLOG
```

Output in the NetView log follows:

```
STATMON.BROWSE ACTS NETWORK LOG FOR 04/14/99 (92300) COLS 037 114
HOST:HOST1 *1* *2* *3* *4* S:CSR
---4---------5---------6---------7---------8---------9---------10---------11----

PIE NETVIEW MVS D A,L | CORRWAIT 5 | LOGTO NETLOG SYSLOG
IEE104I 15.17.55 99.104 ACTIVITY 513
JOBS M/S TS USERS SYSAS INIT S:CSR
00000 00007 00001 00014 00002 00001/00300
VLF VLF VLF NSW S LLA LLA NSW S
JES2 JES2 IEFPROC NSW S MYVTAM MYVTAM VTAM NSW S
TSO TSO TCAS OWT S MYESSI MYESSI VTAM NSW S
MYENV MYENV MYENV MYENV MYENV MYENV MYENV MYENV MYENV MYENV
USER2 OWT
```

Output in the system log follows:
NetView Pipelines Device Drivers

Processing steps:
1. The NETVIEW stage invokes the MVS command and places the corresponding response message, in this case an MLWTO, in the pipeline.
2. The CORRWAIT stage allows 5 seconds for each message to return from MVS. CORRWAIT must be used when sending commands to other applications, such as VTAM or MVS, to allow enough time for a response to return.
3. The LOGTO stage reads the MLWTO and writes it to the NetView log and the system log.

Note: There is no output to the screen from this pipeline other than the echo of the PIPE command. The results of the command appear in the system log twice because both MVS and the LOGTO pipe stage have logged the message.

Interfacing with Other Applications: NETVIEW, VTAM

Two useful device drivers, VTAM and NETVIEW, can be used to invoke VTAM and NetView commands respectively. Responses from these commands are inserted into the pipeline for manipulation by subsequent steps. From there, you can use another device driver to put the data wherever you need it.

Running a NetView Command: NETVIEW

Use the NETVIEW stage to run NetView commands (local or cross-domain) and MVS or local VTAM commands with the RMTCMD command. The resulting messages are placed in the pipe.

Note: To issue MVS commands successfully in a pipeline, use extended multiple console support (EMCS) consoles.

Example 1: Issuing an MVS Command

This example shows how to use the NETVIEW stage to run an MVS command:
NetView Pipelines Device Drivers

Output from the pipeline follows:

```
<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCCF NETVIEW CNM01 OPER6 05/17/99 17:21:19</td>
<td></td>
</tr>
<tr>
<td>CNM01 PIPE NETVIEW MVS D A,L</td>
<td>CORRWAIT 5</td>
</tr>
<tr>
<td>IEE1041 17.21.12 99.137 ACTIVITY 620</td>
<td></td>
</tr>
<tr>
<td>JOBS</td>
<td>M/S</td>
</tr>
<tr>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>VLF</td>
<td>VLF</td>
</tr>
<tr>
<td>JES2</td>
<td>JES2</td>
</tr>
<tr>
<td>TSO</td>
<td>TSO</td>
</tr>
<tr>
<td>MYENV</td>
<td>MYENV</td>
</tr>
<tr>
<td>USER2</td>
<td>OWT</td>
</tr>
</tbody>
</table>
```

Processing steps:
1. The NETVIEW stage invokes the MVS command and places the corresponding response messages in the pipe.
2. The CORRWAIT stage allows 5 seconds for each message to return from MVS. Note that MVS is actually a NetView command which sends command text to the MVS system. Use CORRWAIT when sending commands to other applications, such as VTAM or MVS, to allow enough time for a response to return.
3. By selecting the first message, the TAKE stage imposes early termination of the CORRWAIT stage.
4. The CONSOLE stage reads the messages and displays them.

Example 2: Generating a Return Code from the NETVIEW Stage

This example shows how the MOE option with the NETVIEW stage generates a message containing a return code. The command, NONESUCH, is purposely incorrect.

```
PIPE NETVIEW MOE NONESUCH | CONSOLE
```

Output from the pipeline follows:

```
<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCCF NETVIEW CNM01 OPER6 05/17/99 17:05:01</td>
<td></td>
</tr>
<tr>
<td>CNM01 PIPE NETVIEW MOE NONESUCH</td>
<td>CONSOLE</td>
</tr>
<tr>
<td>CNM01 DSI002I INVALID COMMAND: 'NONESUCH'</td>
<td></td>
</tr>
<tr>
<td>CNM01 DWO369I NETVIEW STAGE (1) HAD RETURN CODE 4</td>
<td></td>
</tr>
</tbody>
</table>
```

Processing steps:
1. The NETVIEW stage runs the NONESUCH command and places the messages in the pipe.
2. The CONSOLE stage reads the messages and displays them.

Example 3: Acting on Pipeline Data

This example shows how to use the NETVIEW stage to process data incoming to a command through the pipeline. In the following example, an operator named ORIGOPER sends a message to another operator named OPER6.

```
PIPE LITERAL % READY FOR LUNCH ? %
    | NETVIEW MOE MSGROUTE OPER6 HOLD(Y)
```

Output (on OPER6’s screen) from the pipeline follows:
Processing steps:
1. The LITERAL stage writes the message READY FOR LUNCH? to the pipeline.
2. The NETVIEW stage has an operand, the MSGROUTE command. The pipe contains one message, the READY FOR LUNCH message. The NETVIEW stage invokes the MSGROUTE command one time to act on the message in the pipeline, routing it to OPER6.

If there were multiple messages in the pipeline, the NETVIEW stage would have invoked MSGROUTE once for each message. No copy of the routed message is written to the pipeline output stream. If the stage had an error, the MOE option would have caused a DWO369I message, with nonzero return code, to be placed in the pipe.

Example 4: Acting on a Pipeline Message

This example shows how to use the NETVIEW stage, without a command parameter, as a non-first stage to process messages in the pipeline.

PIPE LITERAL ? LIST STATUS=TASKS ? | NETVIEW MOE | LOCATE /NOT ACTIVE/ | CONSOLE

Output from the pipeline follows:

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW CNM29 OPER3 02/01/99 09:07:40</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN29</td>
<td>PIPE LITERAL ? LIST STATUS=TASKS ?</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OPT TASKID: TASKNAME: SQLOGTSK STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OPT TASKID: TASKNAME: ALIASAPL STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OPT TASKID: TASKNAME: DSISVRT STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OPT TASKID: TASKNAME: DSIOVS STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OPT TASKID: TASKNAME: CMN29VMT STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OPT TASKID: TASKNAME: CMN29VMT STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OPT TASKID: TASKNAME: DSIKREM STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OPT TASKID: TASKNAME: VPDTASK STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OPT TASKID: TASKNAME: DSIITASK STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OPT TASKID: TASKNAME: DSIRQJOB STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OST TASKID: RESOURCE: A01A441 STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OST TASKID: RESOURCE: A01A442 STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OST TASKID: RESOURCE: A01A443 STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OST TASKID: RESOURCE: A01A444 STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OST TASKID: RESOURCE: A01A445 STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: OST TASKID: RESOURCE: A01A446 STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: NNT TASKID: RESOURCE: STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: NNT TASKID: RESOURCE: STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: NNT TASKID: RESOURCE: STATUS: NOT ACTIVE</td>
</tr>
<tr>
<td></td>
<td>- CMN29 TYPE: NNT TASKID: RESOURCE: STATUS: NOT ACTIVE</td>
</tr>
</tbody>
</table>

Processing steps:
1. The LITERAL stage writes the LIST STATUS=TASKS command to the pipe.
2. The NETVIEW stage reads the message in the pipe (the LIST command) and separates the words into the command and its arguments. The command is then invoked. MOE writes the DWO369I message to the pipeline if the stage generates a nonzero return code. In this example, DWO369I was not generated.
If there had been multiple messages in the pipeline, the NETVIEW stage would have run one time for each message.

3. The LOCATE stage selects inactive tasks and writes that information to the pipeline.

4. The CONSOLE stage reads the messages and displays them.

Running a VTAM Command: VTAM

The VTAM stage enables the user to run a DISPLAY, VARY, or MODIFY VTAM command in a local or remote domain.

Example 1: Issuing a VTAM Command

This example shows how to use the VTAM stage to issue a command:

```
PIPE VTAM D NET,STATIONS,SCOPE=ALL
  CORRWAIT 5
  NLOCATE /IST097I/
  TAKE 1
  CONSOLE
```

Output from the pipeline follows:

```
NCCF NETVIEW CNM01 OPER6 05/02/99 11:09:23
* CNM01 PIPE VTAM D NET,STATIONS,SCOPE=ALL | CORRWAIT 5 | NLOCATE /IST097I/
  | TAKE 1 | CONSOLE
' CNM01
IST350I DISPLAY TYPE = STATIONS
IST393I PUT4/5 MAJOR NODE A01MPU , SUBAREA = 1
IST172I NO LINK STATIONS EXIST
IST314I END
```

Processing steps:
1. The VTAM stage invokes the DISPLAY command and places the corresponding response messages in the pipeline.
2. The CORRWAIT stage allows 5 seconds for each message to return from VTAM. CORRWAIT must be used when sending commands to other applications, such as VTAM or MVS, or to another NetView to allow enough time for a response to return.
3. The NLOCATE stage discards the IST097I message and passes the next message, an MLWTO, to its output stream.
4. The TAKE 1 stage selects the first message in its input stream and also imposes an early termination to the timer on the CORRWAIT stage.
5. The CONSOLE stage reads the message and displays it.

Example 2: Generating a Return Code from the VTAM Stage

This example shows how to use the VTAM stage with the MOE option to generate a message containing a return code to the pipeline. The D NETT, TERMS command is purposely incorrect.

```
PIPE VTAM MOE D NETT,TERMS | CORRWAIT 5 | CONSOLE
```

Output from the pipeline follows:
NetView Pipelines Device Drivers

Processing steps:
1. The VTAM stage invokes the incorrect DISPLAY command and returns error messages DSI071I and IST191I, to the pipe. The MOE option generates the DW0369I message.
2. The CORRWAIT stage allows 5 seconds for each message to return from VTAM. CORRWAIT must be used when sending commands to other applications, such as VTAM or MVS, or to another NetView to allow enough time for a response to return.
3. The CONSOLE stage reads the messages and displays them.

Example 3: Running a VTAM Command in a Remote Domain

This example shows how to use the VTAM stage to issue the D NET, CDRMS command in a remote domain and return the results to the screen at the local domain:

```
PIPE NETVIEW RMTCMD LU=CNM01, PIPE (STAGESEP %) VTAM D NET,CDRMS
% CORRWAIT 10
% CONSOLE
```

Output from the pipeline follows:

```
IST350I DISPLAY TYPE = CDRMS
IST099I C00CDRMS TYPE = CDRM SEGMENT , ACTIV
IST482I C01M ACTIV, SA 1, EL 1, NETID = NETC
IST482I C02M NEVAC, SA 2, EL 1, NETID = NETC
IST482I C13M NEVAC, SA 13, EL 1, NETID = NETC
IST482I A09M NEVAC, SA N/A, EL N/A, NETID = NETA
IST482I A19M ACTIV, SA 10, EL 3, NETID = NETA
IST482I A29M NEVAC, SA N/A, EL N/A, NETID = NETA
IST482I A69M NEVAC, SA N/A, EL N/A, NETID = NETA
IST482I A99M NEVAC, SA N/A, EL N/A, NETID = NETA
IST482I B10M NEVAC, SA N/A, EL N/A, NETID = NETB
IST482I B20M NEVAC, SA N/A, EL N/A, NETID = NETB
IST482I B24M NEVAC, SA N/A, EL N/A, NETID = NETB
IST482I B52M NEVAC, SA N/A, EL N/A, NETID = NETB
IST482I D09M NEVAC, SA N/A, EL N/A, NETID = NETD
IST314I END
```

Processing steps:
1. The outer pipe starts running.
2. The NETVIEW stage issues the RMTCMD to send the inner PIPE command to the remote domain to run.
3. The inner pipe starts running.
4. The inner pipe runs at the remote domain. Within it, the VTAM stage issues the DISPLAY command and resulting messages are returned to the inner pipe. (The messages are not displayed at the remote domain.)

5. The CORRWAIT stage waits 10 seconds for each message to return to the inner pipeline from the VTAM stage.

6. The CONSOLE stage reads the messages and returns them to the outer pipe. They are NOT displayed on the screen at the remote location.

7. The inner pipe terminates.

8. The outer pipe resumes.

9. The CORRWAIT stage waits for the messages to return from the inner pipe.

10. The CONSOLE stage at the local domain displays the messages on the screen.

11. The outer pipe terminates.

---

### Working with DASD Data: < (From Disk)

NetView pipelines provides a stage command for reading DASD data into the pipeline, whereupon the records are treated as single-line messages.

#### Reading from DASD: (<)

The < (From Disk) stage enables the user to read data from a member of a partitioned data set. The default PDS ddname is DSIPARM; however, a different ddname may be specified with the member name, as long as the ddname is supported by DSIDKS.

#### Example 1: Reading Data from DASD

This example shows how to use the < stage to read data into the pipeline from a member of a partitioned data set:

```
PIPE < DSIPARM.CNMSTYLE | NLOCATE 1.1 /*/ | CONSOLE
```

Output (first page only) from the pipeline follows:

```
NCCF Tivoli NetView NTVD OPER4 07/24/01 21:07:39
- NTVD DWOB8II PIPE is PIPE implemented in DSIPipe. Type = R
  C NTVD PIPE < DSIPARM.CNMSTYLE | NLOCATE 1.1 /*/ | CONSOLE
| NTVD styleMsg = NetView initialization style sheet processing has begun.
| NTVD There
| NTVD View is one word. The value of &DOMAIN. is "&DOMAIN." and &NV2I is
| NTVD "&NV2I."
| NTVD
| NTVD DOMAIN = CANV2I.01
| NTVD %INCLUDE CNMPWD
| NTVD
| NTVD SuppChar = ?
| NTVD
| NTVD %INCLUDE CNMSTNXT
| NTVD
| NTVD TRACE.OPTION=DISP,PSS,QUE,STOR,UEXIT // list options
| NTVD TRACE.MODE = INT // INT, EXT, GTF, or
| NTVD "NONE"
```

Processing steps:
NetView Pipelines Device Drivers

1. The < stage reads the records from the member named CNMSTYLE of a data set associated with ddname DSIPARM and places them in the pipe, converting each record into a single-line message.

2. The NLOCATE stage reads its input stream and discards all comment lines (those with an asterisk in column 1). The uncommented lines remain in the pipeline.

3. The CONSOLE stage reads the messages and displays them.

Example 2: Reading from a DSICLD ddname Data Set

This example shows how to use the < stage to read records from the CNMSTYLE member associated with ddname DSICLD.

```
PIPE < DSICLD.CNME1035 | LOCATE /STARTCNM/ | CONSOLE
```

Output from the pipeline looks like this:

```
NCCF NETVIEW CNM01 OPER6 04/14/99 15:26:40
* CNM01 PIPE < DSICLD.CNME1035 | LOCATE /STARTCNM/ | CONSOLE
| CNM01 STARTCNM ALL
```

Processing steps:

1. The < stage reads the records from the member named CNMSTYLE associated with the DSICLD ddname. The records are converted to single-line messages and placed in the pipe.

2. The LOCATE stage selects all messages that have the text string STARTCNM and places them in the pipe.

3. The CONSOLE stage reads the messages and displays them on the screen.

Accessing Variables within Command Procedures: VAR, STEM, SAFE

Several device drivers read and write variables in a command procedure environment.

You can:
- Read from or write uniquely named variables (VAR)
- Read from or write to stem variables (STEM)
- Read from or write to the command procedure message queue (SAFE).

Reading from or Writing to Named Variables: VAR

The VAR stage enables the user to read records from, or write records to, variables in a command procedure variable pool.

When used as a first stage, VAR reads variables from a command procedure variable pool into the pipeline. As anything other than the first stage, VAR writes single-line messages or lines of a MLWTO to uniquely named variables.

Example: Working with a Variable

In this example, command list PIPEVAR illustrates how to use the VAR stage in a pipeline to write a message to a variable. Another pipeline is then used to read the variable and display it.

```
********************************************************************
PIPEVAR CLIST
&CONTROL ERR
```
Output from PIPEVAR command list follows:

```
+------------------------------------------------------------------+
| NCCF NETVIEW OP2 02/01/99 10:01                                  |
| CNM01 PIPEVAR                                                  |
|                CNM01 PA3 IS:                                  |
| CNM01 DSI608I PA3 IMMED,IGNORE RETRIEVE AND EXECUTE          |
| CNM01 RETURN CODE IS 0                                         |
+------------------------------------------------------------------+
```

Processing steps for first pipeline:
1. The NETVIEW stage issues the DISPFK command to display function key information. The messages are not displayed, but are be placed in the pipeline.
2. The SEPARATE stage reads the messages, which are in the form of an MLWTO, and it splits the MLWTO into single-line messages.
3. The LOCATE stage selects any message (in this case, one) that contains the string PA3.
4. The VAR stage writes the message to the variable named CLVAR1.

Processing steps for second pipeline:
1. The VAR stage reads the variable, CLVAR1, into the pipeline whereupon it becomes a message.
2. The LITERAL stage inserts the PA3 IS: message in front of the message existing in the pipeline.
3. The CONSOLE stage reads the messages from the pipeline and displays them.

**Reading from or Writing to Variables in a Stemmed Array:**

**STEM**

When used as a first stage, STEM reads stem variables from the command procedure variable pool. As anything other than the first stage, STEM writes single-line messages or lines of a MLWTO to stem variables. Stem variables are available to REXX, HLL, and NetView command list languages.

**Example 1: Writing to Variables Using STEM**

In this example, a REXX command list named PIPSTEMA shows how to use the STEM stage to write the output from a command to stem variables:

```rexx
/* REXX COMMAND LIST PIPSTEMA */
'PIPE VTAM D NET,BFRUSE', /* Run the D NET,BFRUSE cmd */
' | CORRWAIT 25', /* Allow msgs time to return*/
```

```
Partial (page 1) output from PIPSTEMA looks like this. Output continues until the contents of BFRUSE.54 are displayed.

```
NCCF NETVIEW CNM01 OPER6 09/20/99 16:49:11
  * CNM01 16:49:11 PIPSTEMA
  C CNM01 16:49:11 BFRUSE.0 IS 54
  C CNM01 16:49:11 BFRUSE.1 IS IST097I DISPLAY ACCEPTED
  C CNM01 16:49:11 BFRUSE.2 IS IST350I DISPLAY TYPE = BUFFER POOL DATA
  C CNM01 16:49:11 BFRUSE.3 IS IST920I 1000 BUFF SIZE 00567 EXP INCREMENT 00007
```

Processing steps:
1. The NETVIEW stage processes the DISPLAY command and, instead of displaying messages on the screen, writes them to the pipeline.
2. The CORRWAIT stage waits 25 seconds for each related message.
3. The TOSTRING stage selects all messages up to and including the last line of an MLWTO in which the text string END is found.
4. The STEM stage writes the messages to variables named BFRUSE1, BFRUSE2, BFRUSE3, and so forth. BFRUSE0 contains an integer count of the total number of stem variables.

Example 2: Reading from Variables Using STEM

This example shows how to use the STEM stage, as a first stage, to read variables into the pipeline. It is shown in a REXX command list named PIPSTEMB.

```
/* REXX COMMAND LIST PIPSTEMB */
ADDRESS NETVASIS
A.1= 'APPLES'
A.2= 'PEARS'
A.3= 'ORANGES'
A.0=3
'PIPE STEM A. | CONSOLE'
   IF RC = 0 THEN
      SAY 'RC='RC' FROM PIPE '
   EXIT
```

Output from PIPSTEMB follows:

```
NCCF NETVIEW CNM29 OPER6 03/01/99 19:16:00
  * CNM29 PIPSTEMB
  CNM29 APPLES
  CNM29 PEARS
  CNM29 ORANGES
```

Processing steps:
1. The STEM stage reads the variables A.1, A.2, and A.3 into the pipeline. Each variable read is changed into a message. The variable A0 indicates that there are 3 variables to be read.

2. The CONSOLE stage displays the messages.

Example 3: Writing a Null Message to a Variable Using STEM

This is an example of using the STEM stage to write a null message in the pipeline to a variable. It is shown in a REXX command list called PIPSTEMC.

/* REXX COMMAND LIST PIPSTEMC */
'PIPE (STAGESEP !) LITERAL %THIS IS MY MESSAGE% ',
'! DROP 1 ',
'! STEM C.'
/*
SAY 'VALUE OF C.0 IS:' C.0
EXIT

Output from PIPSTEMC follows:

```
NCCF NETVIEW CNM01 OPER2 12/10/99 12:39:00
* CNM01 PIPSTEMC
C CNM01 VALUE OF C.0 IS: 0
```

Processing steps:
1. The LITERAL stage inserts the THIS IS MY MESSAGE message to the pipeline, passing it to the next stage.
2. The DROP stage reads the message and discards it, leaving no messages in the pipeline.
3. The STEM stage reads its input stream and since it contains no messages cannot create variables C.1 ... C.n. It sets the variable C.0 to zero to indicate that no messages were written.

Reading from or Writing to a Command Procedure Message: SAFE

The SAFE stage writes to or reads from a queue of pipeline messages which have complete message attributes and structure. Messages in a named SAFE may be processed by a different pipeline as long as the pipeline resides within the same NetView command list or the same family of nested NetView command lists. Once the NetView command list family is exited, the storage occupied by the named SAFE is freed.

Messages in an unnamed SAFE may not be processed by a different NetView command list. Once the current NetView command list is exited, the storage occupied by the unnamed SAFE is freed.

Example: Reading and Writing a Message Using SAFE

This example shows how to use a REXX command list to process two PIPE commands using the SAFE stage. The first PIPE command writes messages to a named SAFE, while the second reads the messages from the same named SAFE, manipulates and displays them.

/* PIPSAFEA REXX COMMAND LIST */
/* THIS COMMAND LIST USES THE PIPE COMMAND AND THE SAFE STAGE */
/*****************************/
/* */
NetView Pipelines Device Drivers

/* PIPE # 1 */
/* DISPLAY REMOTE DOMAIN INFORMATION AND STORE IN SAFE NAMED */
/* HOLDMSG */
/* ******************************************************************/
'PIPE VTAM D NET,CDRMS', /* issue command */
| 'CORRWAIT 10', /* wait for messages */
| 'CONSOLE', /* display to terminal */
| 'SAFE HOLDMSG' /* save in SAFE msg queue */
/* ******************************************************************/
/* PIPE # 2 */
/* READ MESSAGES FROM THE SAFE, MANIPULATE THEM AND DISPLAY */
/* ******************************************************************/
'PIPE SAFE HOLDMSG ', /* read msgs from safe */
| 'SEPARATE ', /* separate to single lines */
| 'LOCATE \ACTIV\ ', /* select lines with 'activ'*/
| 'CONSOLE' /* display to terminal */
/* ******************************************************************/
/* WRITE RETURN CODE INFORMATION TO TERMINAL AND EXIT CLIST */
'/******************************************************************/
SAY 'RETURN CODE='R C
EXIT
/* ******************************************************************/

Output from PIPISAFEA follows:

| NCCF N E T V I E W | CNM19 OPER6 02/01/99 18:00:00 |
| + CMN19 PIPISAFEA CMN19 |
| IST097I DISPLAY ACCEPTED |
| IST3591 DISPLAY TYPE = CDRMS |
| IST089I A19CDRMS TYPE = CDRM SEGMENT , ACTIV |
| IST482I A19M ACTIV, SA 19, EL 1, NETID = NETA |
| IST482I A09M NEVAC, SA 9, EL 1, NETID = NETA |
| IST482I A29M NEVAC, SA 29, EL 1, NETID = NETA |
| IST482I A69M NEVAC, SA 69, EL 1, NETID = NETA |
| IST482I A99M NEVAC, SA 99, EL 1, NETID = NETA |
| IST482I B18M NEVAC, SA N/A, EL N/A, NETID = NETB |
| IST482I B20M NEVAC, SA N/A, EL N/A, NETID = NETB |
| IST482I B24M NEVAC, SA N/A, EL N/A, NETID = NETB |
| IST482I B52M NEVAC, SA N/A, EL N/A, NETID = NETB |
| IST482I C01M NEVAC, SA N/A, EL N/A, NETID = NETC |
| IST482I C02M NEVAC, SA N/A, EL N/A, NETID = NETC |
| IST482I C11M NEVAC, SA N/A, EL N/A, NETID = NETC |
| IST482I D09M NEVAC, SA N/A, EL N/A, NETID = NETD |
| IST314I END |
| ' CMN19 IST089I A19CDRMS TYPE = CDRM SEGMENT , ACTIV |
| ' CMN19 IST482I A19M ACTIV, SA 19, EL 1, NETID = NETA |
| C CMN19 RETURN CODE = 0 |

Processing steps for first pipeline:
1. The NETVIEW stage processes the display of the remote domain information, however instead of displaying the messages, they are placed in the pipeline.
2. The CORRWAIT stage allows 10 second intervals between each message returning to the pipeline. Should the 10 second duration be exceeded, no more messages would be placed in the pipeline.
3. The CONSOLE stage reads the messages from the pipeline and displays them.
4. The SAFE stage reads the messages from the pipeline, and writes them to a message queue named HOLDMSG.

Processing steps for second pipeline:
Building Large PIPE Commands: INTERPRT

The INTERPRT stage enables the user to build and run pipeline stage commands from input command data. The commands are processed by the INTERPRT stage as an inner pipeline in a PIPE-within-a-PIPE structure. In some command procedure environments, commands are limited to 240 characters. The INTERPRT stage allows pipeline specifications to exceed these limitations.

Using the INTERPRT Stage

Example: Building a Pipeline with INTERPRT

This example shows how to use the INTERPRT stage to build a pipeline specification from stage specifications read into the pipeline through the STEM stage.

The example is shown within a NetView command list, named PIPEEX10:

```clist
PIPEEX10 CLIST
&CONTROL ERR
*
*******************************************************************
** THIS CLIST USES THE PIPE COMMAND AND INTERPRT STAGE TO PROCESS**
** A PIPELINE. **
*******************************************************************
*
*******************************************************************
** CREATE VARIABLES FOR EACH STAGE TO BE INTERPRETED **
*******************************************************************
&STAGE1 = 'NETVIEW LIST KEY=ALL'
&STAGE2 = 'SEPARATE'
&STAGE3 = 'TAKE FIRST 10'
&STAGE4 = 'COLLECT'
&STAGE5 = 'CONSOLE'
&STAGE0 = 5
*******************************************************************
**
** EXECUTE PIPE COMMAND, PLACE RESULTS IN A SAFE NAMED MYKEYS **
*******************************************************************
PIPE STEM STAGE +
| COLLECT +
| NETVIEW PIPE INTERPRT * +
| SAFE MYKEYS +
| CONSOLE
*******************************************************************
** WRITE RETURN CODE INFORMATION TO TERMINAL AND EXIT CLIST **
*******************************************************************
&WRITE RETURN CODE = &RETCODE
&EXIT
```

Output from PIPEEX10 follows:
### NetView Pipelines Device Drivers

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW</th>
<th>NTVAA OPERI 04/29/99 15:00:01</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ NTVAA MYC ' NTVAA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS1606I DISPLAY OF PF/PA KEY SETTINGS FOR NCCF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS1607I KEY ----TYPE---- -----------COMMAND------------ SET-APPL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS1608I PA1 IMMED,IGNORE RESET NETVIEW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS1608I PA2 IMMED,IGNORE AUTOWRAP TOGGLE NETVIEW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS1608I PA3 IMMED,IGNORE RETRIEVE AND EXECUTE NETVIEW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS1608I PF1 IMMED,APPEND HELP NETVIEW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS1608I PF2 IMMED,APPEND GO NCCF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS1608I PF3 IMMED,IGNORE RETURN NETVIEW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS1608I PF4 IMMED,APPEND DISP FK NETVIEW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS1608I PF5 IMMED,IGNORE BROWSE NETLOGA NETVIEW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C NTVAA RETURN CODE = 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Processing steps:

1. The STEM stage reads the variables named STAGE1 through STAGE5 into the pipeline, where each one becomes a message. By analyzing variable STAGE0, the STEM stage is aware that there are 5 stage variables to be read.
2. The COLLECT stage gathers the messages into an MLWTO.
3. The INTERPRT stage reads the MLWTO, creates and invokes a pipeline specifications, which is the inner pipeline in the PIPE-within-a-PIPE structure. The last stage (CONSOLE) of this pipeline returns output to the outer pipe.
4. The SAFE stage writes the messages output from the invoked pipeline into a safe named MYKEYS.
5. The CONSOLE stage displays the pipeline messages.
Chapter 4. NetView Pipeline Filters

This chapter describes general-use programming interface and associated guidance information. For information about using pipelines in high-level languages, refer to the Tivoli NetView for z/OS Customization: Using PL/I and C.

A filter is a stage command that reads messages from its input stream, manipulates them, and writes the results to its output stream. One common use of a filter is to select or discard messages based on some search or positional criteria. The difference between a filter and a device driver is that a filter does not interact with devices or other system resources, as device drivers do.

The output stream from a filter stage can be far different from the data read from the input stream. By stringing filters together, you can transform raw data into useful results.

This chapter describes filters that:
- Manipulate messages
- Select messages by content
- Select messages by position
- Discard messages from the pipeline

Manipulating Messages: SEPARATE, COLLECT

NetView Pipelines includes stage commands that can modify messages in the pipeline.

You can:
- Separate multiline messages into multiple single-line messages (SEPARATE)
- Collect multiple single-line messages into a multiline message (COLLECT).

Breaking Up an MLWTO: SEPARATE

The SEPARATE stage allows the user to break a multiline write-to-operator (MLWTO) message into single-line messages, each of which inherits the attributes of the MLWTO.

Example: Separating an MLWTO into Single-line Messages

This example shows how to use the SEPARATE stage to break the MLWTO created by the NETVIEW stage into single-line messages:

PIPE NETVIEW MVS D A,L | CORRWAIT 20 | SEPARATE | CONSOLE

Output from the pipeline looks similar to:
NetView Pipeline Filters

Processing steps:
1. The NETVIEW stage executes an MVS command and writes the messages as an MLWTO to the output stream.
2. The CORRWAIT stage allows a 20 second wait for messages to be returned from MVS, resetting the timer as each message is received.
3. The SEPARATE stage reads the input stream and splits the MLWTO into single-line messages.
4. The CONSOLE stage reads the input stream and displays the messages on the screen.

Note: If the SEPARATE stage had not been used, the pipeline output would have looked similar to:

Building an MLWTO: COLLECT
The COLLECT stage enables the user to create a single MLWTO from the messages in the pipeline.

Example: Building an MLWTO from Single-Line Messages
This example shows how to use the COLLECT stage within the PIPE command to gather the messages output from the LIST "" command, into an MLWTO:
PIPE NETVIEW LIST "" | COLLECT | CONSOLE

Output from the pipeline looks similar to:
Processing steps:
1. The NETVIEW stage executes a LIST command and writes the results to the output stream as single-line messages.
2. The COLLECT stage reads its input stream and collects the single-line messages into one MLWTO, before writing the MLWTO to the output stream. Notice that there is only one message prefix for an MLWTO.
3. The CONSOLE stage reads its input and displays the MLWTO.

Note: If the COLLECT stage had not been used, the pipeline output would have looked like this:

```
NCCF NETVIEW CNM01 OPER6 04/14/95 13:45:00
* CNM01 PIPE NETVIEW LIST '' | CONSOLE
  - CNM01 STATION: OPER6 TERM: A01A701
    HCOPY: NOT ACTIVE PROFILE: DSIPROFA
    STATUS: ACTIVE
    DOMAIN LIST: CNM01 (I) CNM02 (I) CNM20 (I) CNM99 (I)
    ACTIVE SPAN LIST: NONE
    END OF STATUS DISPLAY
```

Selecting Messages by Content: LOCATE, NLOCATE, TOSTRING

Several stage commands select pipeline messages. That is, they read all messages in the pipeline, but write only those that meet some selection criteria to the stage’s output stream. This section describes filters that select messages based on the content of the message itself. You can:

- Keep all messages that contain a match for a specified text string (LOCATE)
- Discard all messages that contain a match for a specified text string (NLOCATE)
- Keep messages up to and including the message that contains a match for a specified text string (TOSTRING)

All of these filters are case sensitive: they pay attention to uppercase and lowercase. To a filter, the words Apple and apple are not equal.

Be aware that sometimes the NetView program automatically translates input to uppercase. An easy way to determine whether this is happening is to insert a CONSOLE stage and notice whether the output to the screen is in uppercase. If case sensitivity is required, code the PIPE command in a REXX command list using the ADDRESS NETVASIS instruction to establish an environment sensitive to case. For additional information about REXX command lists, refer to Tivoli NetView for z/OS Customization: Using REXX and the NetView Command List Language.
Keeping or Discarding Matching Messages: LOCATE, NLOCATE

The LOCATE stage keeps messages that contain specified text strings and discards messages that do not. Conversely, the NLOCATE stage discards messages that contain specified text strings and keeps messages that do not. You can specify up to 40 delimited strings, each with optional position and length to limit the column range of the search.

Example 1: Keeping Messages that Contain a Specified Text String

This example shows how to use the LOCATE stage to select single-line messages that contain a specified text string:

```
PIPE NETVIEW LIST STATUS=TASKS | LOCATE /OST/ | CONSOLE
```

Output from the pipeline looks like this:

```
NCCF NETVIEW CNM01 OPER6 05/17/95 13:38:00
  * CNM01 PIPE NETVIEW LIST STATUS=TASKS | LOCATE /OST/ | CONSOLE
  - CNM01 TYPE: OST TASKID: OPER6 RESOURCE: A01A701 STATUS: ACTIVE
  - CNM01 TYPE: OST TASKID: OPER6 RESOURCE: A01A702 STATUS: NOT ACTIVE
  - CNM01 TYPE: OST TASKID: OPER6 RESOURCE: A01A703 STATUS: NOT ACTIVE
  - CNM01 TYPE: OST TASKID: OPER6 RESOURCE: A01A704 STATUS: NOT ACTIVE
  - CNM01 TYPE: OST TASKID: OPER6 RESOURCE: A01A705 STATUS: NOT ACTIVE
  - CNM01 TYPE: OST TASKID: OPER6 RESOURCE: A01A706 STATUS: NOT ACTIVE
  - CNM01 TYPE: OST TASKID: AUTO1 RESOURCE: AUTO1 STATUS: ACTIVE
  - CNM01 TYPE: OST TASKID: AUTO2 RESOURCE: AUTO2 STATUS: ACTIVE
  - CNM01 TYPE: OST TASKID: DSILCOPR RESOURCE: DSILCOPR STATUS: ACTIVE
```

Processing steps:
1. The NETVIEW stage executes the LIST command and writes the results to the output stream as single-line messages.
2. The LOCATE stage reads its input stream (output stream of the NETVIEW stage), examines the messages for occurrences of the OST string, and writes all matching messages to the output stream discarding unmatched messages.
3. The CONSOLE stage reads the messages and displays them.

Example 2: Keeping Messages that Contain Multiple Text Strings

This example shows how to use the LOCATE stage to select messages containing any one of the multiple text strings specified:

```
PIPE NETVIEW LIST DEFAULTS | SEPARATE | LOCATE /LOG/ /DISP/ | CONSOLE
```

Output from the pipeline looks similar to:

```
NCCF NETVIEW CNM01 OPER6 05/17/95 13:40:30
  * CNM01 PIPE NETVIEW LIST DEFAULTS | SEPARATE | LOCATE /LOG/ /DISP/ | CONSOLE
  ' CNM01 DWO654I DISPLAY DEFAULTS
  ' CNM01 SYSLOG: NO
  ' CNM01 NETLOG: YES
  ' CNM01 HCYLOG: YES
  ' CNM01 DISPLAY: YES
```
Processing steps:

1. The NETVIEW stage executes the LIST command and writes the results to the output stream. The result of the LIST DEFAULTS command is an MLWTO.

2. The SEPARATE stage reads its input stream, which contains the MLWTO, and breaks it into single-line messages, each message preserving the characteristics of the MLWTO. These messages are written to the output stream.

3. The LOCATE stage reads its input stream, examines the messages for an occurrence of either the LOG or DISP text strings, and writes any messages that match to the output stream.

4. The CONSOLE stage reads its input and displays the messages.

**Example 3: Discarding Messages that Contain a Specified Text String**

This example shows how to use the NLOCATE stage to discard messages that contain a text string which resides in a specified column range:

```
PIPE NETVIEW LIST STATUS=TASKS | NLOCATE 55.10 /NOT ACTIVE/ | CONSOLE
```

Output (first page) from the pipeline looks similar to:

```
NCCF NETVIEW CNM01 OPER6 04/14/95 13:38:00
* CNM01 PIPE NETVIEW LIST STATUS=TASKS | NLOCATE 55.10 /NOT ACTIVE/ | CONSOLE
- CNM01 TYPE: MNT TASKID: MNT RESOURCE: CNM01 STATUS: ACTIVE
- CNM01 TYPE: PPT TASKID: CNM01PPT RESOURCE: CNM01PPT STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSILOG TASKNAME: DSILOG STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSICRTR TASKNAME: DSICRTR STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSITRACE TASKNAME: DSITRACE STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: CMNCSSIR TASKNAME: CMNCSSIR STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: CMNCALRT TASKNAME: CMNCALRT STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DISVRRT TASKNAME: DISVRRT STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIROVS TASKNAME: DSIROVS STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIGDS TASKNAME: DSIGDS STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSELTSK TASKNAME: DSELTSK STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: AAUTSKLP TASKNAME: AAUTSKLP STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: AAUTCNMI TASKNAME: AAUTCNMI TASKNAME: AAUTCNMI STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIAMLUT TASKNAME: DSIAMLUT STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: BNJDSERV TASKNAME: BNJDSERV STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: BNMNPDA TASKNAME: BNMNPDA STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: BNDSE36 TASKNAME: BNDSE36 STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: CNMTAMEL TASKNAME: CNMTAMEL STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: OPER6 RESOURCE: AUTO1 STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: AUTO1 RESOURCE: AUTO1 STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: AUTO2 RESOURCE: AUTO2 STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: DSILOGPR RESOURCE: DSILOGPR STATUS: ACTIVE
- CNM01 END OF STATUS DISPLAY
```

Output (second page) from the pipeline looks similar to:

```
NCCF NETVIEW CNM01 OPER6 04/14/95 13:38:00
- CNM01 TYPE: OPT TASKID: DSIKREM TASKNAME: DSIKREM STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIUDST TASKNAME: DSIUDST STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: CNMTAMEL TASKNAME: CNMTAMEL STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIGDST TASKNAME: DSIGDST STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIHDPST TASKNAME: DSIHDPST STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSITSK TASKNAME: DSITSK STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: OPER6 RESOURCE: AUTO1 STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: AUTO1 RESOURCE: AUTO1 STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: AUTO2 RESOURCE: AUTO2 STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: DSILOGPR RESOURCE: DSILOGPR STATUS: ACTIVE
- CNM01 END OF STATUS DISPLAY
```
1. The NETVIEW stage is used to execute the LIST command and write the results as single-line messages to the output stream.

2. The NLOCATE stage reads the messages and examines them for the NOT ACTIVE string in columns 55–64. If that string is found, the message is discarded. All other messages are written to the output stream.

3. The CONSOLE stage reads the messages and displays them.

**Note:** If the example were changed to add an additional selection criteria to the NLOCATE stage, searching for OPT in columns 7–9, the results from the pipeline are similar to:

```
NCCF NETVIEW CNM01 OPER6 05/17/95 13:38:43
+ CNM01 PIPE NETVIEW LIST STATUS=TASKS | NLOCATE 55.10 /NOT ACTIVE/ 7.3 /OPT/ | CONSOLE
  - CNM01 TYPE: MNT TASKID: MNT RESOURCE: CNM01 STATUS: ACTIVE
  - CNM01 TYPE: PPT TASKID: CNM01PPT RESOURCE: CNM01PPT STATUS: ACTIVE
  - CNM01 TYPE: OST TASKID: OPER6 RESOURCE: A01A702 STATUS: ACTIVE
  - CNM01 TYPE: OST TASKID: AUTO1 RESOURCE: AUTO1 STATUS: ACTIVE
  - CNM01 TYPE: OST TASKID: AUTO2 RESOURCE: AUTO2 STATUS: ACTIVE
  - CNM01 TYPE: OST TASKID: DSILCOPR RESOURCE: DSILCOPR STATUS: ACTIVE
  - CNM01 END OF STATUS DISPLAY
```

**Example 4: Using NLOCATE to Process an MLWTO**

This example shows how to use the NLOCATE stage to discard an MLWTO that contains a specified text string:

```
PIPE NETVIEW LIST DEFAULTS | NLOCATE /LOG/ | CONSOLE
```

Output from the pipeline is similar to:

```
NCCF NETVIEW CNM01 OPER6 02/01/95 10:11:15
+ CNM01 PIPE NETVIEW LIST STATUS=DEFAULTS | NLOCATE /LOG/ | CONSOLE
```

**Processing Steps:**

1. The NETVIEW stage executes the LIST command and writes the results to the output stream as an MLWTO.
2. The NLOCATE stage reads its input stream, examines the messages for the LOG text string and because LOG occurs at least once in the MLWTO, the entire MLWTO is discarded.
3. Because there are no messages in its input stream, the CONSOLE stage ends without displaying anything. Only the command echo is seen.

**Note:** If the example were changed to insert the SEPARATE stage prior to the NLOCATE stage, the pipeline results would change. The SEPARATE stage breaks the output of the LIST command, an MLWTO, into single-line messages. NLOCATE would read its input stream, and examine each single-line message (instead of the entire MLWTO) for the occurrence of LOG, discarding matches as appropriate.

```
PIPE NETVIEW LIST DEFAULTS | SEPARATE | NLOCATE /LOG/ | CONSOLE
```

Output (first page) from the pipeline looks similar to:
Selecting Messages Up to and Including a Message That Matches a Specified Text String: TOSTRING

The TOSTRING stage enables the user to select messages in the input stream up to and including the message containing the text that matches the specified string. Selected messages continue in the pipeline and others are discarded. You can specify up to 40 delimited strings, each with optional position and length to limit the column range of the search.

TOSTRING is a terminating stage, meaning that once a match is found, processing for this stage terminates along with any outstanding CORRWAIT. Any unprocessed records in the input stream is discarded.

Example 1: Processing an MLWTO Using TOSTRING

This example shows how to use TOSTRING with the LAST option to search for a text string in either a single-line message or the last line of an MLWTO.

```
PIPE NETVIEW D NET,CDRMS | CORRWAIT 60 | TOSTRING LAST 1.7 /IST314I/ | CONSOLE
```

Output from the pipeline looks similar to:

```
NetView Pipeline Filters
Chapter 4. NetView Pipeline Filters
NetView Pipeline Filters

Processing steps:
1. The NETVIEW stage executes the command to DISPLAY CDRMS and writes the messages to the output stream.
2. The CORRWAIT stage allows a 60 second wait for messages to be returned from VTAM, resetting the timer as each message is received. When the CORRWAIT timer expires, further messages generated by the NETVIEW stage are not accepted into the pipeline.
3. Before 60 seconds expire, the message DISPLAY ACCEPTED is passed to TOSTRING and the last (only) line is examined for the matching string. There is no match, so the message is passed to the output stream.
4. Again, before 60 seconds expire, the MLWTO is passed to TOSTRING and because the last line of the MLWTO matches the delimited text string, the entire MLWTO is passed to the output stream. Because TOSTRING is satisfied, the TOSTRING stage terminates, making CORRWAIT also terminate, and ending the receipt of messages from the D NET,CDRMS command.
5. The CONSOLE stage reads the messages and displays them.

Example 2: Processing Single-Line Messages Using TOSTRING

This example shows how to use TOSTRING to search for a delimited string in a particular column range, within single-line messages. This example is shown as a REXX command list named DISP482I:

```
/* DISP482I REXX command list */
/* process PIPE command with TOSTRING stage */
'PIPE NETVIEW D NET,CDRMS',
  'CORRWAIT 60', /* wait 60 seconds */
  'SEPARATE', /* split up MLWTO */
  'TOSTRING 1.7 /IST482I/', /* search for match */
  'CONSOLE' /* display on screen */
SAY 'RC IS ' RC
EXIT
```

Output from DISP482I looks similar to:

```
NCCF NETVIEW CNM01 OPER6 03/26/95 16:50:00
+ CNM01 DISP482I
  CNM01 IST097I DISPLAY ACCEPTED
  ' CNM01 IST350I DISPLAY TYPE = CDRMS
    CNM01 IST098I AGICDRM TYPE = CDRM SEGMENT , ACTIV
    ' CNM01 IST482I ADIM ACTIV, SA 1, EL 1, NETID = NETA
  ' CNM01 IST482I A01M ACTIV, SA 2, EL 1, NETID = NETB
C CNM01 RC IS 0
```

Processing steps:
1. The NETVIEW stage executes the DISPLAY command and writes messages, which include an MLWTO to the output stream.
2. The CORRWAIT stage allows a 60 second wait for messages to be returned from VTAM, resetting the timer as each message is received. When the CORRWAIT timer expires, further messages generated by the NETVIEW stage is not accepted into the pipeline.

3. The SEPARATE stage reads its input stream and splits the MLWTO into single-line messages, each message preserving the characteristics of the MLWTO.

4. The TOSTRING stage examines each message for an occurrence of IST482I, which begins in column 1. Each message is read and passed to the output stream until a match is found.

5. The CONSOLE stage reads its input and displays the messages.

Selecting Messages by Position: TAKE, DROP

Pipeline messages are processed by a stage respective to the order of their position in the pipeline. Several stages enable the user to either keep or discard records based on their position.

You can specify the first or last n messages to be:

- Kept in the pipeline
- Discarded from the pipeline

Keeping the First or Last n Messages: TAKE

Example 1: Keeping the First two Messages in the Pipeline

This example shows how to use the TAKE stage to select the first two messages to remain in the pipeline:

```
PIPE NETVIEW D NET,CDRMS | CORRWAIT 60 | TAKE 2 | CONSOLE
```

Output from the pipeline looks similar to:

```
NCCF NETVIEW CNM01 OPER5 03/01/95 11:00:00
+ CNM01 PIPE NETVIEW D NET,CDRMS | CORRWAIT 60 | TAKE 2 | CONSOLE
' CNM01 IST097I DISPLAY ACCEPTED
' CNM01 IST350I DISPLAY TYPE = CDRMS
IST089I A01CDRM TYPE = CDRM SEGMENT , ACTIV
IST482I A01M ACTIV, SA 1, EL 1, NETID = NETA
IST482I A02M NEVAC, SA 2, EL 1, NETID = NETA
IST482I A99M NEVAC, SA 99, EL 1, NETID = NETA
IST482I B20M NEVAC, SA N/A, EL N/A, NETID = NETB
IST314I END
```

Processing steps:

1. The NETVIEW stage executes the DISPLAY command and writes messages to the output stream.

2. The CORRWAIT stage allows a 60 second wait for a message to be returned from VTAM, then resets the timer to wait for the next message. When the CORRWAIT timer expires, messages generated by the NETVIEW stage are not accepted into the pipeline.

3. Before the 60 seconds expire, the first message DISPLAY ACCEPTED is passed to TAKE and is passed on to the output stream. The CORRWAIT timer is reset.

4. Again, before the 60 seconds expire, the second message, an MLWTO, is passed to TAKE and is passed to the output stream. TAKE has now selected 2 messages; therefore, it terminates.
5. CORRWAIT is also terminated, due to the termination of TAKE. Therefore, messages from the display command are no longer read into the pipeline.

6. The CONSOLE stage reads the messages and displays them.

Example 2: Keeping the Last two Messages in the Pipeline

This example shows how to use the TAKE stage to select the last two messages resulting from a command. This is similar to the previous example; however, the MLWTO has been split into single-line messages using the SEPARATE command and the TAKE command is selecting the last messages rather than the first:

```
PIPE NETVIEW D NET,CDRMS | CORRWAIT 60 | SEPARATE | TAKE LAST 2 | CONSOLE
```

Output from the pipeline looks similar to:

```
NCCF NETVIEW CNM01 OPER5 03/20/95 12:00:10
* CNM01 PIPE NETVIEW D NET,CDRMS | CORRWAIT 60 | SEPARATE | TAKE LAST 2 | CONSOLE
  " CNM01 IST482I B20M NEVAC, SA N/A, EL N/A, NETID = NETD
  " CNM01 IST314I END
```

Processing steps:
1. The NETVIEW stage executes the DISPLAY command and writes messages to the output stream.
2. The CORRWAIT stage allows a 60 second wait for each message to be returned from VTAM. When the CORRWAIT timer expires, further messages generated by the NETVIEW stage are not accepted into the pipeline.
3. Before the 60 seconds expire, the message DISPLAY ACCEPTED arrives in the input stream, is read by SEPARATE, and passed as is to the output stream. The timer is reset.
4. Again, before the 60 seconds expire, the MLWTO is passed to SEPARATE and because it is an MLWTO, it is split into single-lines, then passed to the output stream.
5. The stream is passed to TAKE, which selects the last 2 messages and discards the others.
6. CORRWAIT waits the full 60 seconds, because TAKE LAST is not considered a terminating stage condition.
7. The CONSOLE stage reads the messages and displays them.

Discarding the First or Last n Messages: DROP

Example: Discarding the First Two Messages

This example shows how to use the DROP stage to discard the first two messages output from a command. The example shown contains a REXX command list named DROP2:

```
/* REXX COMMAND LIST - DROP2 */
'PIPE NETVIEW MAJNODES', /* ISSUE COMMAND */
  ' CORRWAIT 10', /* WAIT 10 SECONDS */
  ' DROP FIRST 2', /* DROP FIRST 2 MSGS. */
  ' CONSOLE' /* DISPLAY PIPE CONTENTS */
SAY 'RETURN CODE IS' RC /* DISPLAY RETURN CODE */
EXIT
```

Output from DROP2 looks similar to:
Emptying the Pipeline: HOLE

This section describes how to discard all messages from the pipeline, thus emptying it of all contents.

Determining Correlation: HOLE

Discarded messages from the pipeline Using the HOLE stage, you can, for instance, empty the pipeline of messages after writing them to a variable or to test a command for correlated output.

Example: Determining Command Response Correlation to a Command

This example shows how to use the HOLE stage to determine whether a command and its messages are correlated. If the messages are correlated, they enter the pipeline and are discarded; otherwise, they are displayed.

PIPE NETVIEW MVS $D I | CORRWAIT 25 | HOLE

Output (first page only) from the command looks similar to:

Note: These messages represent uncorrelated messages, indicating that the command is NOT supported for pipeline processing. They displayed immediately after the PIPE command was entered.
Processing steps:
1. The NETVIEW stage executes the MVS command and writes messages to the output stream. Only correlated messages are processed in the pipe. Uncorrelated messages return to the panel immediately without waiting for the CORRWAIT timer to expire.
2. The CORRWAIT stage allows a 25 second wait between each message received. If more than 25 seconds elapses, CORRWAIT disconnects and does not receive further messages from the pipeline.
3. The HOLE stage discards the messages in the pipeline.
Chapter 5. Full-Screen Automation

This chapter describes general-use programming interface and associated guidance information. The information describes how to interact with NetView full-screen panels from customer-written applications and the commands provided by NetView to create, manage, and terminate full-screen applications for the purpose of automation. The NetView commands are:

- ATTACH
- DETACH
- VET

What Is Full-Screen Automation

The full-screen automation function enables a program to interact with NetView full-screen applications in the same way an operator interacts with NetView. From a REXX, PL/I, or C program, you can:

- Read data from a NetView application panel.
- Write data to a NetView application panel.
- Press PF, PA, Enter, or clear keys on a NetView application panel.

Full-screen automation can access other full-screen applications using the Terminal Access Facility (TAF). For considerations when using TAF, see "Partial Screens" on page 281.

Full-screen automation is intended as an automation tool and not a function to assist NetView operators by providing fast-path or other simplified commands.

A Simple Example

Full-screen automation has three main steps:

1. Starting a NetView application. A full-screen automation program interacts with the NetView application. The NetView application is started with the ATTACH command.
2. Interacting with the NetView application. This is done with the VET command and pipe stage.
3. Terminating the NetView application. This can be done explicitly with the DETACH command or allowed to occur implicitly.

Consider a simple procedure to return alarm threshold information for a modem and write the information to the console. The example in Figure 21 on page 268 does not show all the options available in full-screen automation, nor does it contain error handling. Instead, the example illustrates the sequence of starting, interacting with it, and terminating the application.
Starting a NetView Application

The first step in full-screen automation is starting the application. In Figure 21 on page 268 the ATTACH command starts the command MDMCNFG. All parameters that are valid on MDMCNFG can be included on the ATTACH command. In this case, MDMCNFG requests changeable configuration information for the remote modem on the link connecting NCP1 and TERM4.

The ATTACH command in this example starts MDMCNFG and makes the panel shown in Figure 22 available as a virtual screen.

Note: A virtual screen is not physically displayed on a hardware screen. Instead, it is a way to make the data, which would be displayed on a hardware screen, available to a full-screen automation program. For more information on virtual screens, see “Virtual OSTs” on page 270.

Starting a NetView Application

The first step in full-screen automation is starting the application. In Figure 21 on page 268 the ATTACH command starts the command MDMCNFG. All parameters that are valid on MDMCNFG can be included on the ATTACH command. In this case, MDMCNFG requests changeable configuration information for the remote modem on the link connecting NCP1 and TERM4.

The ATTACH command in this example starts MDMCNFG and makes the panel shown in Figure 22 available as a virtual screen.

Note: A virtual screen is not physically displayed on a hardware screen. Instead, it is a way to make the data, which would be displayed on a hardware screen, available to a full-screen automation program. For more information on virtual screens, see “Virtual OSTs” on page 270.

```
/* REPTALRM: REXX procedure to report the alarm thresholds of a modem. */

'ATTACH MDMCNFG ID=NCP1,STATION=TERM4,MODEM=REMOTE,BROWSE=CONFIG'
'PIPE (NAME GETCONFG)',
'  VET NEXT ROWS',
'  CORRWAIT 60',
'  SEPARATE',
'  NOT TOSTRING /ALARM THRESHOLDS/,'
'  TOSTRING NOINCL /HIT ENTER/,'
'  CONSOLE'
```

Figure 21. Simple Full-Screen Automation Example: REPTALRM

Interacting with and Terminating a NetView Application

The following example shows how to return to the console the following four lines on the MDMCNFG panel followed by a blank line:

```
/* REPTALRM: REXX procedure to report the alarm thresholds of a modem. */

'ATTACH MDMCNFG ID=NCP1,STATION=TERM4,MODEM=REMOTE,BROWSE=CONFIG'
'PIPE (NAME GETCONFG)',
'  VET NEXT ROWS',
'  CORRWAIT 60',
'  SEPARATE',
'  NOT TOSTRING /ALARM THRESHOLDS/,'
'  TOSTRING NOINCL /HIT ENTER/,'
'  CONSOLE'
```

Figure 21. Simple Full-Screen Automation Example: REPTALRM

```
NETVIEW NCF01 OPER1 01/16/99 11:55:13

* BROWSE CHANGEABLE 7861 CONFIGURATION PARAMETERS *
ID = NCP1  STATION = TERM4  MODEM = REMOTE  LEVEL = 1

BASIC MODEM CONFIGURATION
SPEED CONTROL MODE M (M=MODEM, D=DTE)
TRAINING SEQUENCE L (L=LONG, S=SHORT)
CONFIGURATION P (M=MULTI-POINT, P=POINT TO POINT)
NETWORK FUNCTION C (C=CONTROL/PRIMARY, S=SECONDARY)
ANTISTREAMING N (Y=YES, N=NO)
TRANSMIT CLOCK OPTION E (I=INTERNAL, E=EXTERNAL, R=RECEIVE)
COMPLEMENTARY RFS DELAY 0 MS (0 TO 250 IN 10MS INCREMENTS)
DEFAULT SPEED F (F=FULL, B=BACKUP)
LOCAL LOOP BACK WRAP N (Y=YES, N=NO)

CUSTOMER INFORMATION (10 CHARACTER LIMIT)

ALARM THRESHOLDS
RECEIVE LEVEL THRESHOLD -43 DBM (-43 to 0)
IMPULSE HITS THRESHOLD 21 (0 TO 63)
LINE QUALITY THRESHOLD 10 (0 TO 14)

HIT ENTER TO END COMMAND
```

Figure 22. Browse Changeable Configuration Panel
Consider each stage in the PIPE to determine how the required four lines are selected and returned.

Processing steps:
1. PIPE (NAME GETCONFG) starts the pipeline and gives the pipeline the name GETCONFG.
2. VET NEXT ROWS returns the screen image shown in Figure 22 on page 268 to the pipeline in the following multiline message:

```
BNH150I ROWS/NEXT OUTPUT FOR ' ' RECEIVED FROM MDMCNFG
NETVIEW NCF01 OPER1 01/16/99 11:55:13
* BROWSE CHANGEABLE 7861 CONFIGURATION PARAMETERS *
ID = NCP1 STATION = TERM4 MODEM = REMOTE LEVEL = 1
BASIC MODEM CONFIGURATION
SPEED CONTROL MODE M (M=MODEM, D=DTE)
TRAINING SEQUENCE L (L=LONG, S=SHORT)
CONFIGURATION P (P=MULTI-POINT, P=POINT TO POINT)
NETWORK FUNCTION C (C=CONTROL/PRIMARY, S=SECONDARY)
ANTISTREAMING N (Y=YES, N=NO)
TRANSMIT CLOCK OPTION E (I=INTERNAL, E=EXTERNAL, R=RECEIVE)
COMPLEMENTARY RFS DELAY 0MS (0 TO 250 IN 10MS INCREMENTS)
DEFAULT SPEED F (F=FULL, B=BACKUP)
LOCAL LOOP BACK WRAP N (Y=YES, N=NO)
CUSTOMER INFORMATION (10 CHARACTER LIMIT)
ALARM THRESHOLDS
RECEIVE LEVEL THRESHOLD -43 DBM (-43 to 0)
IMPULSE HITS THRESHOLD 21 (0 TO 63)
LINE QUALITY THRESHOLD 10 (0 TO 14)
HIT ENTER TO END COMMAND
```

Figure 23. Example of Screen Returned for VET NEXT ROWS

The BNH150I message indicates the type of VET command used in the PIPE and provides information about the attached command. The ROWS/NEXT format returns the panel in the format as it is displayed on an operator’s screen. More information on BNH150I can be found in “Handling Returned Messages” on page 274. Also, see the VET command information in “Interacting with Virtual OSTs” on page 272.

3. CORRWAIT 60 causes the pipeline to wait up to 60 seconds for MDMCNFG to build its panel on the virtual screen. The wait ends when MDMCNFG is ready to receive input.
4. SEPARATE changes the multiline message returned by VET into multiple single-line messages.
5. NOT TOSTRING /ALARM THRESHOLDS/ discards all lines up to the line containing the string /ALARM THRESHOLDS/.
6. TOSTRING NOINCL /HIT ENTER/ selects all remaining lines of the panel up to, but not including, the line containing the string /HIT ENTER/.
7. In Figure 24 on page 272 the selected lines are written to the CONSOLE.
Full-Screen Automation

8. The REXX EXEC terminates and the virtual screen running MDMCNFG is automatically detached.

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW</th>
<th>CNM19 PETE</th>
<th>03/26/99 13:10:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ CNM19</td>
<td>REPTALRM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>' CNM19</td>
<td>ALARM THRESHOLDS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>' CNM19</td>
<td>RECEIVE LEVEL THRESHOLD</td>
<td>-43 DBM (-43 to 0)</td>
<td></td>
</tr>
<tr>
<td>' CNM19</td>
<td>IMPULSE HITS THRESHOLD</td>
<td>21 (0 TO 63)</td>
<td></td>
</tr>
<tr>
<td>' CNM19</td>
<td>LINE QUALITY THRESHOLD</td>
<td>10 (0 TO 14)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 24. REPTALRM Results

See “Chapter 2. Pipeline Stages and Syntax” on page 19 for information on pipe stages.

Because MDMCNFG was attached by this procedure, without the ATTACH NAME parameter, MDMCNFG automatically terminates when the procedure ends. You can also terminate MDMCNFG explicitly using the DETACH command. For more information about the ATTACH and DETACH commands, see “Attaching and Detaching Virtual OSTs” on page 271.

Note: Additional examples can be found in CNME2011 and CNMS1101. CNME2011 contains a SESSMGET example and CNMS1101 contains an NPDA example.

Virtual OSTs

NetView uses virtual operator station tasks (virtual OSTs or VOSTs) instead of operator station tasks (OSTs) to present full-screen data to a program. The VOST has a virtual screen where full-screen operations and subcommands of the application running on the VOST can be run.

For information about valid subcommands for each NetView application, refer to the NetView online help. The application name is included in parenthesis in the heading of each subcommand description.

VOST virtual screens are 24 lines by 80 characters. All NetView operations and subcommands that can be entered on a real 24-line by 80-character 3270 full-screen terminal can be entered on a VOST virtual screen.

VOSTs are created using the ATTACH command. See “Attaching VOSTs” on page 271.

Dependent and Independent VOSTs

A VOST can be dependent or independent of its creating procedure group.

VOSTs, which are named when they are created, are independent of the procedure group in which they are run. Independent VOSTs persist until specifically ended using the DETACH command or until the command running on the VOST ends.
Unnamed VOSTs are local to the procedure group. When all components of the procedure group terminate, the VOST terminates. Unnamed VOSTs can be explicitly ended with the DETACH command.

## Attaching and Detaching Virtual OSTs

NetView provides an ATTACH command to create VOSTs and a DETACH command to explicitly end VOSTs. An ATTACH command must be executed in the code before the VET stage or command.

### Attaching VOSTs

VOSTs, and the applications running on them, are started using the ATTACH command. The simplest form of command that starts a VOST to run NPDA code is:

```
ATTACH NPDA
```

An ATTACH command, without a name, creates a VOST that is dependent on the procedure group. A dependent VOST terminates after all procedures in the procedure group terminate except under the following conditions:

- A DETACH command for the VOST is included within the procedure group.
- The command running on the VOST terminates.

In both conditions, the VOST detaches before all the procedures in the procedure group terminate.

A name must be specified on the ATTACH command to create an independent VOST. For example, to start an independent VOST named RUN$NPDA running NPDA, code:

```
ATTACH NAME RUN$NPDA NPDA
```

The RUN$NPDA VOST continues to run until NPDA terminates or the VOST is specifically terminated using the DETACH command.

VOST names can be 1–8 characters in length and must only contain uppercase and lowercase alphabetic characters, numbers, or the characters @, #, and $.

### Note:

- Some commands are not valid for ATTACH. Refer to the NetView online help for the ATTACH command for a list of commands that are not valid.
- VOSTs are accessible only from the owning task. In addition, dependent VOSTs are accessible only from the procedure family in which they were attached.
- The attached command is checked against the authority of the owner of the VOST.
- If you want to use NAME or DUMP as the NetView command running on the VOST, consider:
  - If the NetView command you want to run on the VOST is NAME, you cannot code:
    ```
    ATTACH NAME
    ```
    However, you can use the CMD command to start the NetView command NAME. For example:
    ```
    ATTACH CMD NAME
    ```
To attach a VOST called NAME that will run the NetView NAME command, code:

```
ATTACH NAME NAME NAME
```

- If the NetView command you want to run on the VOST is DUMP, and you do not want to use the ATTACH DUMP keyword, you can use the CMD command to start NetView DUMP:

```
ATTACH CMD DUMP
```

For the NetView commands NAME and DUMP, you can define a CMDSYN or embed the commands in a REXX procedure.

**Detaching VOSTs**

VOSTs can be detached explicitly or implicitly. In most cases, it is not necessary to explicitly detach a VOST. As described in "Attaching VOSTs" on page 271, if a VOST was created without a NAME on the ATTACH command, the VOST is detached when the procedure group terminates.

However, both named and unnamed VOSTs can be explicitly detached using the DETACH command. DETACH is the same as a LOGOFF on an end-user terminal.

The simplest form of the DETACH command is:

```
DETACH
```

**Note:** A DETACH with no keywords detaches one VOST. Do not use DETACH without a name if you have more than one VOST created within the procedure group. An unnamed DETACH, in the case of multiple VOSTs in a procedure group, can cause unpredictable results. The VOST intended might not be detached.

If a name was not specified on an ATTACH, you can still detach using a DETACH NAME. Default attach names are the same as the command name. So, you can code: DETACH NPDA if you previously coded: ATTACH NPDA

You can also use the STOP FORCE command to terminate a VOST. However, STOP FORCE may terminate the VOST before all VOST processing completes. You may get unpredictable results.

**Interacting with Virtual OSTs**

After starting the application using the ATTACH command, you are ready to interact with the application running on the VOST. The VET command and PIPE stage are used to communicate with a VOST.

**Hint:** VOSTIO is synonymous with VET.

VET has two forms: a first stage form and a form that can be used as a subsequent stage or a command.

In both forms, the NAME keyword on the VET command permits interaction with a specific VOST if you have more than one VOST attached. The parameter specified on VET NAME must match the default name or the name specified on the NAME keyword of the ATTACH command. For example, if you attached two VOSTs using the following commands:

```
ATTACH NAME VOST1 NPDA
ATTACH NLDM
```
The first ATTACH would create an independent VOST named VOST1 running NPDA. The second would create an unnamed, dependent VOST running NLDM. If you want to interact with the NPDA application, include NAME VOST1 in your VET stage specification.

**VET First Stage**

The VET command used as a first stage VET command creates the input stream for subsequent pipe stages by reading data from the virtual screen and passing the data in the form of messages to the output stream.

If the application running on the VOST presents a full-screen, VET produces a special message, BNH150I, in the pipe containing information about the VOST virtual screen. If the application does not present a full-screen, messages generated by the application are returned.

See "Handling Returned Messages" on page 276 for information about BNH150I and other messages that can be returned by PIPE VET.

VET, as a first stage, returns the data on the virtual screen in one of two ways:

- **ROWS**
  Requests the screen data be presented to the output stream as 24 lines of 80 characters each following the BNH150I message header. ROWS returns the data exactly as it is displayed on the virtual screen.

- **FIELDS**
  Requests that each data field on the virtual screen be presented as one line following the BNH150I message header.

See "Handling Returned Messages" on page 276 for information about the format of the data returned by VET ROWS and VET FIELDS.

The VET first stage also enables you to specify when you want to read the virtual screen. The following options are available:

- **CURRENT**
  Specifies that the content of the virtual screen at the time VET executes is to be presented to the output stream. A VET CURRENT after an ATTACH and before a VET NEXT returns a blank screen of data.

- **NEXT**
  Specifies that the next change made to the virtual screen is to be presented to the output stream. In general, the data passed to the output stream is not a complete screen image. Instead, only the portions of the screen that have been changed since the last time a first stage VET was executed are passed to the output stream. If used with ROWS, all unchanged fields are filled with X'FF'.

  VET NEXT must be used to retrieve the first screen of data after an ATTACH.

  VET NEXT should normally be followed by CORRWAIT. For more information about CORRWAIT, see "Chapter 2. Pipeline Stages and Syntax" on page 19.

**A VET Command or Subsequent Stage**

After reading data from the virtual screen using the VET first stage, the subsequent stage form of VET is used to interact with the application running on the VOST by writing data to the virtual screen and simulating the pressing of the PF, PA, or...
Enter keys. To the application running on the VOST, it appears as though a human operator is entering data on the application full-screen.

When using VET on subsequent stages, you can specify the data to be written to the virtual screen, the cursor position where the data should be written, and the action key to be pressed after the data is written to the virtual screen.

For example, ATTACH NPDA creates a VOST running NPDA. The first panel presented on the virtual screen is NPDA-01A. To request the total events panel (NPDA-40A):

```
NETVIEW VET /2/
```

In this example, a row.col was not specified, so the string `/2/` is written to the VOST in the first unprotected field on the panel at, or after, the current cursor position. Because NPDA-01A only has one unprotected field, which is the command line, 2 is written to the unprotected field. The VET action key defaults to ENTER. After 2 is written to the command line, information is passed to NPDA indicating that the Enter key was executed on the VOST. NPDA responds to the VOST as though a human operator entered 2, which is a NPDA subcommand or selection choice, and pressed the Enter key.

Notes:

1. You can use ENTER, PF, PA, and CLEAR keys as action keys. ENTER is the default.
2. Using NOKEY as an action key enables you to enter data on the virtual screen, but indicates that no action is to be taken. This is the same as when a human operator types data on a panel and does not press an action key, such as Enter, to indicate to the application that the input is complete.
3. If the application running on the VOST enables dynamic remapping of PF and PA keys using the NCCF SET command, the PF and PA key action keys cannot be used on the VET command. Instead, use the command to be executed in the `/string/` with an Enter action key. For example, if the application running on the VOST can have PF keys remapped and as a default has PF3 set to END, code VET /END/ ENTER.

**Using Delimiters**

Delimited strings can be used when VET is a command.

Strings to be written to the virtual screen must be enclosed in delimiters. The first nonblank character encountered after the stage name or row.col is the delimiter, which establishes the boundary of the text string used by VET. The delimited string ends when the same character is encountered a second time.

If you want to execute an action key on the VOST without entering data on the virtual screen, specify either a null string with two delimiters with no space between them or omit the `/string/` altogether. For example, if you want to press PF8 without entering data on a panel, code either of the following:

```
NETVIEW VET //PF8
VET PF8
```

**Writing in Protected Fields**

After a string is written to a panel on the virtual screen, the cursor is moved to the next field on the panel. If you attempt to write a string to a protected field, the cursor moves to the next unprotected field before writing the data. If there are no more unprotected fields, the cursor wraps back to the top of the panel and finds...
the first unprotected field and writes the data. Strings are truncated if they are longer than the unprotected field in which they are written.

**Note:** If you attempt to write a /string/ to a panel that does not have unprotected fields, the /string/ is ignored, but the action key is processed. DSRBS is an application that presents panels that do not have unprotected fields.

For example, STATMON running on a VOST displays:

<table>
<thead>
<tr>
<th>STATMON.DSS</th>
<th>DOMAIN</th>
<th>STATUS</th>
<th>SUMMARY</th>
<th>(REFRESH=ON)</th>
<th>10:50 A</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST: HOST126</td>
<td>+1*</td>
<td>+2*</td>
<td>+3*</td>
<td>+4*</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>....</td>
</tr>
<tr>
<td>...1 NCP/CA/LAN/PK...</td>
<td>...1</td>
<td>...1</td>
<td>...1</td>
<td>...1</td>
<td>NCP/CA/LAN/PK</td>
</tr>
<tr>
<td>...1</td>
<td>...1</td>
<td>...1</td>
<td>...1</td>
<td>...1</td>
<td>...1</td>
</tr>
<tr>
<td>...1</td>
<td>...1</td>
<td>...1</td>
<td>...1</td>
<td>...1</td>
<td>...1</td>
</tr>
<tr>
<td>...5</td>
<td>...5</td>
<td>...5</td>
<td>...5</td>
<td>...5</td>
<td>...5</td>
</tr>
<tr>
<td>...49</td>
<td>...49</td>
<td>...49</td>
<td>...49</td>
<td>...49</td>
<td>...49</td>
</tr>
<tr>
<td>...151</td>
<td>...151</td>
<td>...151</td>
<td>...151</td>
<td>...151</td>
<td>...151</td>
</tr>
<tr>
<td>...603</td>
<td>...603</td>
<td>...603</td>
<td>...603</td>
<td>...603</td>
<td>...603</td>
</tr>
<tr>
<td>...815</td>
<td>...815</td>
<td>...815</td>
<td>...815</td>
<td>...815</td>
<td>...815</td>
</tr>
</tbody>
</table>

CMD==>

TO SEE YOUR KEY SETTINGS, ENTER ‘DISPFK’

On this panel, there are 89 unprotected fields. The fields include the command line and each of the following fields, delimited with vertical bars (|).

| .... | NCP/CA/LAN/PK | | LINES | | PUS/CLUSTERS | | LOCAL MAJ NDS | | LUS/TERMS | | APPL MAJ NDS | | APPLICATIONS | | CD RMS MAJ NDS | | CD RMS | | CD RSC MAJ NDS | | CD RSCS | | TOTAL NODES |
| ---- | ------------- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- | ---- |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |
| .... | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 | ...1 |

The following REXX program written to interact with this panel requests a STATMON detail panel on NCP/CA/LAN/PK. An end-user would place a nonblank character in the first field of the NCP/CA/LAN/PK line on summary panel. To do so in a full-screen automation REXX program, code:

```
NETVIEW VET 4.2 /X/
```

This command positions the cursor at the 4th row and 2nd column, places an X in that column, and presses the Enter key. You can also code the following in your REXX program:

```
input. = ""
input.5 = 'X'
input.0 = 5
```
In this case, a stem variable is created with five values. The first four are null. A null value passed to VET places nothing in an unprotected field, and the cursor skips to the next field. In the previous example, VET begins in column 1 row 1, skips the first four unprotected fields (*1*, *2*, *3*, and *4*), and places an X in the fifth unprotected field (.....1 NCP/CA/LAN/PL). When the entire stem is written to the panel, the Enter key is pressed.

**Hint:** The following items should be considered when coding your full-screen automation program:

- The /string/ coded on a VET command can be a pipeline specification.
- Before detaching a VOST, you may want to code NETVIEW VET /LOGOFF/, or the appropriate logoff, or termination command for the application running on the VOST.
- A VET first stage returns a blank screen if a logoff or termination command was executed for the application on the VOST.

### Handling Returned Messages

The processing of message BNH150I is key to full-screen automation. All full-screen data returned by a VET first stage is formatted in this multiline message.

The first line of the multiline message is:

```
BNH150I format/action OUTPUT FOR 'application' RECEIVED FROM attachname
```

The following message variables, which are contained in this BNH150I message header line, indicate the options specified on the VET stage, the VOST name, and the application name:

- **format** Specifies whether the VET first stage requested the full-screen data to be returned in ROW or FIELDS format.
- **action** Specifies whether the VET first stage requested a CURRENT or NEXT view of the virtual screen.
- **application** Indicates whether the application running on the VOST is enabled for dynamic remapping of PF and PA keys using the NCCF SET command. If it does, an application name will be included in this message variable. If the **application** is null (""), the application running on the VOST does not allow remapping of PF and PA keys, so PF and PA keys can be used as action keys on VET subsequent stages.

In most cases, **application** is null or the same as the command name. However, **application** can be different from the ATTACH command in the following situations:

- ATTACH commands that are CLISTS
- ATTACH commands that are synonyms for other commands
- ATTACH commands that attach non-IBM applications

- **attachname** Specifies the name of the VOST that has its data returned. This is either the
NAME specified on the ATTACH command when the VOST was created or the default name when no NAME is specified on the ATTACH.

The lines following the BNH150I message header line depend on whether the message results from VET ROWS or VET FIELDS.

**ROWS Format**

If BNH150I results from VET ROWS, the BNH150I header line is followed by 24 lines of 80 characters, returning the data exactly as it is displayed on the virtual screen. Each field is shown in its position on the screen. If BNH150I results from VET NEXT ROWS, all fields that have not been updated since the last VET NEXT will contain X’FF’.

VET ROWS presents the rows following the BNH150I message, in order, from row 1 through row 24. For example, VET CURRENT ROWS to a VOST with the STATMON panel shown on page "STATMON Summary Panel" on page 273 returns:

![Example Screen for VET CURRENT ROWS](image)

**Figure 25. Example Screen for VET CURRENT ROWS**

In this example, BNH150I indicates that a VET CURRENT ROWS was issued, the name specified on the VOST ATTACH was either specified or defaulted to STATMON, and STATMON is given as the application name running on the VOST. Because an application name is given in the message, STATMON allows dynamic remapping of PF and PA keys using NCCF SET, so subsequent VET stages cannot pass PF or PA keys to the VOST.

**Note:** The example panel contains the following line:

TO SEE YOUR KEY SETTINGS, ENTER ‘DISPFK’

This line is included in the panel for a NetView operator. DISPFK is an independent command and cannot be used with full-screen automation.
**FIELDS Format**

For the FIELDS format, each field, protected or unprotected, is returned individually. There may be one field or hundreds of fields. These fields are presented in the order they are returned to VET by the application running on the VOST. VET FIELDS also provides the 3270 data stream attributes for each field.

Each line following the BNH150I message header line, which describes a field, is formatted as follows:

```
row.col Fx Iy RESERVED %data
```

If COLOR was specified on the ATTACH command, color and highlighting information are also available:

```
row.col Fx Iy Cz Ha RESERVED %data
```

The individual parts of each line are:

- **row.col** The starting row and column of the field. For example, if you have a field that is from row 3 column 10 through row 3 column 15, row.col will be 3.10 and 6 characters will be in data. Fields can be contained in one row or may be multiple rows deep.

- **Fx** Indicates whether the field is protected. FA indicates that the field is protected. FI indicates that it is not protected.

- **Iy** Indicates whether the field is intensified; Iy can have one of these three values:
  - **IN** The field is not intensified.
  - **IH** The field is intensified.
  - **ID** The field is not visible on the panel.

- **Cz** The color of the field; Cz indicates that the field is one of the following colors:
  - **CB** Blue
  - **CR** Red
  - **CP** Pink
  - **CY** Yellow
  - **CG** Green
  - **CW** White
  - **CT** Turquoise
  - **CD** Default

- **Ha** The highlighting of the field; Ha indicates that the field has one of the following highlights:
  - **HR** Reverse Video
  - **HU** Underlined
  - **HB** Blinking
  - **HD** Default

**RESERVED**

Additional information may be included in the reserved area. Information contained in the reserved area is subject to change and should not be used for automation purposes.

- **%** Is a delimiter separating the field attributes from the actual data contained within the field. The % delimiter immediately precedes the field data and should be used when parsing the message line.
data

Contains the data within the field. The first character is the data position of the field order, if any, from the 3270 data stream. This character is presented as a blank.

The first line following the BNH150I header line has the following format:

```
write-type row.col option
```

The individual parts of this line are:

```
write-type
```

The `write-type` is either WRITE or ERASE-WRITE. Write-type indicates the action taken by the application running on the VOST when updating the panel. WRITE indicates that changes were made to the panel, but the panel was not totally rewritten. ERASE-WRITE indicates that the panel was erased and rewritten.

```
row.col
```

The starting row and column of the options. This is also the current cursor position on the virtual screen.

```
options
```

Can be BEEP, LOCK, or both. Options indicates whether the terminal beep is sounded or the panel is locked from input.

Reference: For more information about 3270 data stream attributes, refer to the 3270 Information Display System library.

For example, if you use the following code in your full-screen automation application:

```
ATTACH NPDA
PIPE (NAME BADCOM)
  |VET NEXT FIELDS
  ...
Stages to process returned panel
  ...
  NETVIEW VET /BAD COMMAND/
PIPE (NAME GETPANEL)
  |VET NEXT FIELDS
  |CONSOLE
```

A VOST starts running NPDA. BAD COMMAND is entered on the command line. Because BAD COMMAND is not a valid command, NPDA returns an error. VET NEXT FIELDS returns only those fields that have changed. In this example, the following is displayed on the console:

```
BNH150I FIELDS/NEXT OUTPUT FOR 'NPDA' RECEIVED FROM NPDA
WRITE 24.8 BEEP
22.1 FA IN % BN905I INVALID COMMAND ENTERED OR INCORRECT OPERANDS
23.1 FA IN %
23.80 FA IH % CMD==>
24.8 FI IH % BAD COMMAND
```

```
Figure 26. Example Screen for VET NEXT FIELDS
```

The BNH150I message header line indicates that the message results from VET NEXT FIELDS from a VOST running NPDA. WRITE 24.8 BEEP shows that the panel was updated without first being erased and that the 3270 data stream BEEP
command was issued. The next four messages show the fields that were changed by NPDA. The field beginning at row 22 column 1 shows the message that was displayed on the virtual screen as a result of the command. The other fields show the changed fields resulting from VET /BAD COMMAND/.

The following example shows a BNH150I message after an ATTACH command with the COLOR option. Note that the Cz and Ha information is included indicating color and highlighting for each field:

* NTV7E TOM ATTACH (ACTION='PIPE VET CURRENT FIELDS|CONS',COLOR) HELP
* NTV7E TOM PIPE VET CURRENT FIELDS|CONS
' NTV7E TOM
BNH150I FIELDS/CURRENT OUTPUT FOR 'HELP' RECEIVED FROM HELP.
BEEP 24.13
1.1 FA IN CB HD % CNMKNCFF
1.2B FA IH CW HD % COMMAND FACILITY HELP MENU
2.1 FA IN CT HD %
3.1 FA IN CT HD %
4.1 FA IH CT HU % Select
4.8 FA IN CB HD % To get information about
5.1 FA IN CT HD %
6.1 FA IN CT HD %
6.4 FA IH CW HD % 1
6.9 FA IN CT HD % Operator's overview of the command facility
7.1 FA IN CT HD %
7.4 FA IH CW HD % 2
7.9 FA IN CT HD % Using the terminal access facility (TAF)
8.1 FA IN CT HD %
9.1 FA IN CT HD %
9.4 FA IH CW HD % 3
9.9 FA IN CT HD % The command facility screen
10.1 FA IN CT HD %
10.4 FA IH CW HD % 4
10.9 FA IN CT HD % Command facility commands and command lists

In this case, line 4 shows that the word Select is in turquoise and underscored.

**Other Messages**

A full-screen automation program must be able to handle BNH150I and other returned messages from VET NEXT and VET CURRENT.

For example, you might try to ATTACH a VOST and run a command that does not display a full-screen. Messages that would result from the command being entered on an end-user screen are returned to the full-screen application.

If both a full-screen and messages are presented by the application running on the VOST, a VET first stage presents both BNH150I and other messages to the output stream. The order in which these messages are presented depends on how the application generates them. For example, if the application first issues two messages and then presents a full screen, the messages are returned to VET first, then BNH150I with the full-screen information. However, if the application presents a full screen and then messages, the messages are held until the full-screen portion of the application terminates. Only after the full screen terminates are the messages returned to VET.

In the example shown in [Figure 27 on page 281](#), message BNJ924I is returned from a VET NEXT ROWS in addition to BNH150I containing the NPDA-01A panel. NPDA-01A and BNJ924I resulted from ATTACH NPDA SD NODOM.
Partial Screens

Full-screen automation programs can require considerable knowledge of the screen-handling techniques used by the applications running on the VOST. This is especially true if the applications write partial screens of data to the virtual screen.

To a human operator these partial screens may not be detectable, but an application handling data at computing speeds must be tolerant of partial screen updates.

There are also special considerations for applications running on a VOST running the Terminal Access Facility (TAF). TAF sessions are started using BGNSESS FLSCN.

Sample CNMS1098 demonstrates techniques you can use to avoid problems with applications running in the TAF environment and those providing partial screen updates. This sample uses TAF to log on to a TSO ID, collect data from the Spool Display and Search Facility (SDSF) active display, and log off.

Debugging Full-Screen Automation Programs

You can use the DUMP keyword to create a display of all data sent to and from the VOST. This data might be useful in problem determination.

Figure 27 on page 283 shows a partial dump from ATTACH DUMP NPDA:
Dump data will be returned each time data is received or sent from the application running on the VOST.

NPDA Automation Example

The full-screen automation example HALERT shown in Figure 30 on page 283 returns alert history information. Although HALERT can be entered as a command, the example output shown in Figure 29 shows HALERT executed as part of the following PIPE:

```
PIPE NETVIEW HALERT | COLLECT | CONSOLE ONLY
```

In this PIPE, output generated to the CONSOLE by HALERT is further processed by the PIPE in which it is executed. The same is true if you include REXX SAY commands in routines that are to be executed as part of a PIPE.

Note: The source code for this example is included in CNMS1101.
Within this example there are a number of important things to note:

1. The ATTACH command creates a VOST and starts NPDA for automation. Full-screen applications can be automated only if started with ATTACH.

2. The PIPE AHIST1 waits for a response from NPDA using VET NEXT and CORRWAIT. This pipeline ends normally when NPDA is ready to receive input.

3. If BNH150I is returned, the NPDA main menu was successfully displayed,
and processing continues. If BNH150I was not returned, an error was detected. No further processing is done on the main menu panel.

4. If the timeout message DWO369I is generated, an error has occurred during CORRWAIT. In this simple example, the error is captured.

5. Messages not selected by LOCATE 1.7 /DWO369I/ are passed as an input stream to the connector A:. The messages are then written to the variable *npdamsg*. HALERT terminates if messages are contained in the *npdamsg* variable.

6. The VET command is used to type /ALH/ on the NPDA command line. The default action key, ENTER, is passed to NPDA. NPDA responds as though the command ALH was entered on a terminal by a NetView operator.

7. There may be many pages of alert history. The last page of alert history is determined by finding PAGE x OF x on the NPDA panel, where x is the total number of alert history pages.

8. VET NEXT ROWS collected a screen image from NPDA. This time we want to examine the screen image. Messages, other than BNH150I, will be handled by the simple pipe connected with the $C: connector.

9. After the data from this panel is reported to the caller of the example procedure, one line is reserved to trigger a command.

10. A VET command ignores the line used to trigger it. /FORWARD/ is passed to NPDA with an Enter action key. If VET is coded as a stage, instead of a command, it is a subsequent stage and attempts to write the data line passed from TAKE 1 to the NPDA screen.
Chapter 6. Using NetView SQL Stages for Access to DB2

This chapter describes how to use the NetView System Query Language (SQL) stages to access the relational database DB2. Access to SQL is through pipe stages, enabling you to quickly write powerful systems and network management applications using REXX, C, PL/I, or other NetView supported languages. Also, the SQL and SQLCODE provide compatible language to similar functions available on VM and MVS/TSO. This simplifies work on multiple platforms. You can access multiple DB2 databases on the local host.

Accessing and Maintaining Relational Databases (SQL Tables)

NetView SQL stages can use DB2 if the NetView task DSIDB2MT is started and is connected to a DB2 subsystem. Records are passed between DB2 and the NetView SQL stage without changing the format of fields, for example, integers are not printable in a query. This gives you control over the format of data moving to and from the database. However, the following sample program SQSELECT, which is supplied with NetView SQL stages, formats a query and converts data from internal representation to character strings:

```
SQSELECT * from inventory where substr(description,1,4)='S390'
```

```
SUBAREA_NO DESCRIPTION------------- JOBS_ACTIVE
+221 S390 Raleigh NC +2000
+222 S390 Austin TX +1250
```

The SQL Stage

The SQL stage connects NetView to the relational database product IBM DATABASE 2™ (DB2) for MVS. Potential users of SQL include the following:

- Novice users of DB2 and NetView pipelines:
  - SQSELECT is a NetView REXX procedure to format a query for display.
  - Input lines read by a NetView SQL stage are issued as DB2 statements such as INSERT or SELECT statements. The format of data is the internal DB2 format; use the EDIT stage to convert between external and internal formats.

- NetView SQL stage users who are new to DB2. Advanced DB2 education is outside the scope of this book. Refer to the examples provided and DB2 publications to learn about DB2.

The REXX samples provide program templates, illustrating the major steps to be performed. Writing SQL applications using NetView pipelines eliminates many of the programming details, such as DB2 precompilation, reentrant considerations, and the details of memory management.

Defining DB2 to NetView

- The DSIDB2DF member of DSIPARM is read when task DSIDB2MT starts, and defines the DB2 subsystem (SUBSYSTEM=DB2) to which NetView will connect. It uses the following definition statement:
  SUBSYSTEM=DB2

The SUBSYSTEM definition identifies which DB2 is the NetView default. The DB2ACCESS definition sets which DB2 interface is used for NetView unless the
Using NetView SQL Stages for Access to DB2

DB2SEC keyword in CNMSTYLE is set to RRS or CAF. In either case, the setting is applied the first time either definition is processed, and the interface is not changed while the NetView program is running.

- The NetView SQL stage must be defined to DB2. The NetView SQL stage is defined once in a process called binding the plan. Sample CNMSJSQSQL contains a sample installation job to perform this process. In the following sample job, the name of the library on the SYSUT2 JCL statement must match the name specified in the BIND statement in the second step of the job. The sample uses the name USER2.DBRMLIB, which you can modify to suit your system:

```sql
//SYSUT2 DD DSN=USER2.DBRMLIB(DSISQLDO),DISP=SHR
```

The following sample shows the BIND statement:

```sql
BIND PACKAGE(DSISQL04) MEM(DSISQLDO) ACT(REF) -
  ISOLATION(CS) LIB('USER2.DBRMLIB') OWNER(USER2)
BIND PACKAGE(DSISQL14) MEM(DSISQLDP) ACT(REF) -
  ISOLATION(CS) LIB('USER2.DBRMLIB') OWNER(USER2)
BIND PLAN(DSISQL04) ACT(REF) -
  PKLIST(DB2L01.DSISQL04.DSISQLDO,DSISQL14.DSISQLDP) ISOLATION(CS) OWNER(USER2)
```

DSISQL04, DSISQLDO and DSISQLDP are NetView names that do not change. Change IBMUSER to a value suitable to your installation to identify the NetView that is using SQL. In general, the NetView plan name is DSISQLnn where nn is changed due to service or future releases. The CNMSJSQSQL sample is reshipped when a change to the DSISQLDO and DSISQLDP programs causes the plan to be incompatible.

To run CNMSJSQSQL, the job user ID must have BINDADD privilege.

- The NetView job must access DB2 load libraries. Sample CNMSJ009 contains comments, in the STEPLIB DD statements, to help you modify your production procedure.

- If you use Distributed Relational Database Access (DRDA), ensure that all other systems know about NetView SQL stages. Unload the plan from the system where you have installed NetView SQL stages and bind it to the other systems.

- You must be registered as an DB2 user to query a table. Contact your database administrator to determine whether you are registered. When you have connect authority, you can query tables.

- To create tables, you must have a TABLESPACE or write privileges to a space owned by another user. Your database administrator can allocate space.

- You can define the level of DB2 access with the DB2SEC keyword in CNMSTYLE.

If DB2SEC is set to RRS, the NetView program loads the RRS interfaces and you have operator level security checking. You can then access multiple DB2 subsystems on your system.

If DB2SEC is set to CAF, the NetView program loads the CAF interfaces. You do not have operator level security, but you can still access multiple DB2 subsystems on your system.

When the RRS or CAF interfaces are loaded, tasks can access DB2 directly without needing the DSIDB2MT task. DSIDB2MT must still be active for tasks that access the DB2 subsystem that DSIDB2MT accesses. When starting the DSIDB2MT task, ensure that SQL requests that do not specify which DB2 to access always runs to the same DB2. A new parameter (SSIDssidname |*) on the SQL stage defines whether a specific or the default DB2 is accessed. When an
SQL stage defines a subsystem to be accessed, that subsystem remains in effect for that task until you reset it by using another SQL stage.

- The DSIDB2MT task must be started if you want to define a default DB2 for the NetView address space. This task connects NetView to a specific DB2 subsystem so that any task in the NetView address space has access to DB2 unless the SQL stage specifies otherwise.
  - Running the RRS level of interfaces requires MVS system definitions.
  - RRS requires a sysplex; this can be a single system monoplex.
  - It does not require a coupling facility if RRS is installed with the DASD only logging option. This requires a COUPLExx sysparm member which identifies the COUPLE and LOGR data sets.
  - Define CFRM data sets if you need them. RRS requires 4 logs and has an additional optional one. They are an ARCHIVE, MAIN.UR, DELAYED.UR, RESTART, and RM.DATA. The *z/OS MVS Programming Resource Recovery* describes these logs.
  - To enable the DASD only logging option, you may need to upgrade to OS/390 Release 4 and apply the following APAR: OW30206, PTF # UW43930. The upgrade to OS/390 requires an upgrade to VTAM Version 4 Release 4.
  - To start RRS, you can add it to the COMMNDxx system parameter member or add it to your automation.

**SQSELECT - Format a Query**

SQSELECT formats a query to be displayed on a terminal. The filter takes a query as the argument, describes the query, and formats the result. In the following example, the first line of the response contains the names of the columns padded with hyphens to their maximum length. The remaining lines represent the result of the query.

The SSID DB2 option is supported for SQSELECT, which passes the value to the underlying SQL stages for example:

```
SQSELECT (SSID DB2) project_name from projects

PROJECT_NAME---
BLUE MACHINE
GREEN MACHINE
ORANGE MACHINE
RED MACHINE
WHITE MACHINE

pipe SQL describe select salary, name from staff | console

484   DECIMAL   8,2   5 SALARY
449   VARCHAR   9    11 NAME

SQSELECT salary, name from staff where years is null

SALARY--- NAME------
96808.30 GRUMPY
93504.60 BASHFUL
92954.75 SLEEPY
91508.60 DOPEY
```

In this example, SSID DB2 defines the DB2 subsystem named DB2 as the subsystem this task will reference.
**Creating, Loading, and Querying a Table**

Use SQL to query and maintain DB2 tables. The following examples show two ways to create a table. In the first example, a single DB2 statement is issued; in the second example, SQL EXECUTE reads statements from its primary input stream. The following example shows that you can supply multiple DB2 statements to a single invocation of the SQL stage:

```sql
pipe SQL execute create table jtest (kwd char(8), text VARCHAR(80))
in netview.xmydb | cons

'pipe 'lit /create table ktest (kwd char(8), '
' text VARCHAR(80)) in netview.xmydb/','
' | SQL execute ',
' | cons '

/* Insert lines in a table */
signal on novalue
"PIPE",
"lit /AAA Automated Banking System/",
" lit /BBB Network Program/",
" lit 'CCC TIVOLI NetView for OS/390 Automation'",
" edit /insert into jtest (kwd, text) values('/ 1 ",
"1.8 next /', '/ next 9.* next /')/ next",
" SQL execute",
"CONS"
exit RC
```

**Note:** You will receive error messages from SQL if you do not enter the names of the columns, even when you are setting all of them.

The following example shows how to use SQL DESCRIBE SELECT to see the format of the input record or the result of a query:

```sql
pipe SQL SSID DB2 describe select * from jtest | console
453 CHAR 8 8 KWD
449 VARCHAR 80 82 TEXT
```

In the previous example, each line describes a column in the table. The first column of the record is the numeric DB2 field code. The field code is decoded in the next column. A third column is the length (or precision) of the field as perceived by DB2. The fourth column is the number of characters required to represent the field when loading with SQL INSERT and when queried with SQL SELECT. Note that the varying character field has two bytes reserved for the length prefix. Finally, the name of the column is shown. SSID DB2 defines that DB2 is the name of the DB2 subsystem to access. Subsequent SQL requests would access that DB2 unless the keyword SSID was used.

The following is an example of an SQL SELECT query:

```sql
pipe SQL select * from jtest | console
""CCCA = ""TIVOLI NETVIEW FOR OS/390 AUTOMATION
""BBB ""NETWORK PROGRAM
""AAA ""AUTOMATED BANKING SYSTEM
```
The double quotes represent unprintable binary data. The first two positions of each column contain the DB2 indicator word that specifies whether the column is null or contains data. This information may be required to process the result of a table query containing columns that can have the null value (no data).

The following example shows how indicator words are suppressed in the output record; the query seen by DB2 is the same in both the preceding and the following examples. The remaining two unprintable bytes contain the length, in binary, of the varying field. Use EDIT to discard these columns.

```
pipe SQL noindicators select * from jtest | console
CCC ""TIVOLI NETVIEW FOR OS/390 AUTOMATION
BBB ""NETWORK PROGRAM
AAA ""AUTOMATED BANKING SYSTEM
```

As an alternative to the previous method, the following example shows how to use EDIT to format binary data:

```
/* Query the test table without formatting */
Signal on novalue
'PIPE',
'SQL noindicators select * from jtest ',
'SEPARATE',
'EDIT 1.3 1 9.2 c2d substr 2.* right 2 5 11.* 8 ',
'console'
Exit RC
CCC 36 TIVOLI NETVIEW FOR OS/390 AUTOMATION
BBB 15 NETWORK PROGRAM
AAA 24 AUTOMATED BANKING SYSTEM
```

In this example, EDIT supports conversion between character and binary or floating point and constructs varying length character fields.

**Note:** The SEPARATE stage creates individual data records for the EDIT stage. If you omit the SEPARATE stage, only the first data record found appears in the output.

**SQSELECT** formats a query against a sample table in the following example:

```
'pipe NETV SQSELECT * from jtest | COLLECT | CONSOLE'
KWD----- TEXT----------------------------------------------------
CCC TIVOLI NETVIEW FOR OS/390 AUTOMATION
BBB NETWORK PROGRAM
AAA AUTOMATED BANKING SYSTEM
```

**Querying a Database and Formatting the Results**

Use SQSELECT to issue a DB2 query and converts the result to a easy to read, printable format with column headings.

**SQSELECT:**

```
| SQSELECT | { SQL options } | DB2 SELECT operands |
```

Note the following when using SQSELECT:
Using NetView SQL Stages for Access to DB2

- If the first nonblank character is a left parenthesis, the string up to the first right parenthesis is processed as options. The remainder of the argument is processed as a DB2 SELECT statement. The default SQL option is NOCOMMIT and implies the plan is not closed.
- Code SQSELECT as a first stage in a pipe:
  PIPE NETV SQSELECT...
  The SSID\textit{ssidname} option is supported.

Working with Graphic Character Strings (DBCS)

- The SQL stage and the SQSELECT procedure can process graphic (DBCS) fields.
- When issuing an SQL INSERT, the shift-in and shift-out characters must be included in the data in the VALUES clause.
- When doing an SQL SELECT, the shift-in and shift-out characters are not returned by SQL.
- The SQSELECT procedure inserts the shift-in and shift-out characters as part of the output fields so that values are properly displayed.
- The field sizes for graphic characters are counted by DB2 as the number of DBCS characters, and the amount of space the field uses is 2 * (number of DBCS characters) bytes.
- A SQL DESCRIBE SELECT request will display the number of DBCS characters in the Field Length column. The number in the Maximum Field Length column is the number of bytes in the record for the field. This is twice the number of DBCS characters plus two bytes if the field has varying length. The shift-in and shift-out characters are counted in the maximum field length (but they take up space in the external data format.) SQSELECT uses a column width equal to the maximum number of bytes the data required (excluding the 2-byte varying length) plus 2 bytes for the shift-in and shift-out characters.

Defining the Unit of Work

DB2 commits changes to the database at the end of the unit of work. The unit of work ends with an explicit COMMIT or by reaching the end of the pipe. Unless instructed by an option, SQL performs an explicit commit and closes the plan when processing is complete. Use the option COMMIT when you want the unit of work to be committed without closing the plan. Use NOCOMMIT in concurrent SQL stages, and to treat a subsequent SQL stage as the same unit of work.

The unit of work can also be rolled back. That is, the database is restored to the state before the unit of work began. SQL automatically rolls the unit of work back when it receives an error code from DB2; use SQL ROLLBACK WORK to perform an explicit rollback, possibly, in response to a pipeline error condition.

Use NOCLOSE to leave the plan open from one pipe to another on the same NetView task. Because the close of the plan takes place when all concurrent SQL stages have terminated, a NOCLOSE option used in any concurrent stage of a pipe keeps the plan open when the pipe ends. If you use NOCLOSE, a subsequent SQL stage may fail if it tries to change the subsystem name, which is the SSID\textit{ssidname} parameter on the SQL stage. DB2 enables only one subsystem at a time per task.

The plan closes when none of the SQL stages specify NOCLOSE. If even one SQL stage specifies NOCLOSE, the plan remains open.
The previous example is an example of committing the unit of work and closing the active plan.

**Note:** If a plan is left open and the REXX procedure ends, the plan remains open until a subsequent pipe closes it or the task ends. A REXX procedure might use `PIPE SQL COMMIT WORK | CONS` at the start of SQL processing to ensure that any previous plan is closed. Alternatively, a `PIPE SQL NOCLOSE CONNECT RESET | CONS` can be used to ensure the local database is being used and the plan is open.

The following are reasons to use NOCLOSE:

- When using SQL CONNECT to connect a remote database, a NOCLOSE enables you to keep the remote connection between two pipes. You may find it convenient to open a remote connection in one pipe, do some processing in REXX, and finish working with the remote connection in a second pipe. Specify NOCLOSE in the first pipe and omit the operand everywhere in the second pipe.
- When using database locks in SQL, use NOCLOSE to keep the locks from one pipe to the next.
- Other applications requiring two pipes to implement one function, typically, with other (REXX) processing between the two pipes.

### Using Secondary Output Streams with SQL

SQL EXECUTE processes multiple I/O requests from the primary input stream that has multiple insert, or query statements, or a mixture of these.

The results of the queries are written to the primary output stream, and have the color attribute green (CG). Output data is written using NetView message type `double quote` (HDRTYPEK).

If the secondary output stream is defined and connected, any error messages are written to it. If a secondary stream is not defined, error output is not sent to the primary stream, but escapes to the parent pipe or is displayed. Error messages also have the color attribute red (CR). Error messages are written with the NetView message type `apostrophe` (HDRTYPEJ).

Diagnostic messages (from the SQL TEST or DIAGNOSE options) are color-coded yellow (CY) and are written to the secondary stream. Diagnostic messages are written with the NetView message type `apostrophe` (HDRTYPEJ).

### Using Concurrent SQL Stages

You can process the results of a query to construct DB2 statements and queries processed in a subsequent SQL stage. As seen from DB2, all concurrent SQL stages are considered to be the same program using a single cursor.

The option NOCOMMIT must be specified when multiple SQL stages are running concurrently. Each stage uses its own work areas. Up to 10 SQL stages can be coded in one PIPE.
Using NetView SQL Stages for Access to DB2

If one stage fails with a DB2 error, the unit of work is rolled back. All subsequent SQL stages in the pipeline will ignore all input. Messages indicating the failure will be issued. NOCLOSE will be respected if coded within the pipeline.

You can process a query and store the result in a REXX stemmed array, test the return code, and issue the second SQL pipeline only when the first one has completed satisfactorily.

When accessing DB2 subsystems from NetView, you cannot directly access multiple DB2 subsystems on a single task without having the SQL close and reopen the plan. Consider using multiple autotasks that interface with a different DB2 subsystem as servers for other tasks. You can use labeled commands and pipes to correlate the SQL requests running on the separate tasks.

Using CONSOLE DUMP to Diagnose SQL Output

The CONSOLE DUMP stage is useful for displaying nonprintable data while writing SQL applications. The output provides the hexadecimal and character representations of the data so that you can determine indicators, lengths, or other numeric fields.
Chapter 7. REXX Access to VSAM Files

The NetView program provides two command processors that access VSAM files:

- DSIVSMX defines, reads, and writes VSAM keyed files without using data services tasks. See "DSIVSMX: Generic Access to Keyed VSAM Files from REXX or Command Lists" for information about this command processor.
- DSIVSAM reads VSAM keyed files that are defined by NetView data services tasks. See "DSIVSAM: Access to Keyed VSAM Files Defined by NetView DSTs" on page 294 for information about this command processor.

The standard way to use both of these command processors is on a NETVIEW pipe stage. For more information about the NETVIEW stage, see "PIPE NETVIEW" on page 137.

DSIVSMX: Generic Access to Keyed VSAM Files from REXX or Command Lists

The DSIVSMX command processor can define, read, and write VSAM keyed files directly from REXX without using data services tasks. This enables the implementation of VSAM applications, including end-use application development in REXX (in conjunction with the pipeline facility), and intensive VSAM diagnostics.

The DSIVSMX command provides REXX access to keyed VSAM files and to IDCAMS, the VSAM Access Method Services utility.

Samples CNMS8013 through CNMS8015 and CNMS8017 through CNMS8020 illustrate the use of the DSIVSMX command.

Refer to Volume 2 of the Tivoli NetView for z/OS Command Reference or the NetView online help for more information about the DSIVSMX command.

Using DSIVSMX with Alternate Index VSAM Files

DSIVSMX can be used with alternate index VSAM files. Alternate index VSAM files can be defined, accessed, and deleted.

Defining a VSAM File with Alternate Index

When setting up an alternate index, follow these steps to build the base and index:

1. Define the BASE cluster, using DSIVSMX IDCAMS with the VSAM DEFINE CLUSTER statements.
2. Load the BASE cluster with the necessary information (prime it), using DSIVSMX PUT requests.
3. Define the AIX® cluster, using DSIVSMX IDCAMS with the VSAM DEFINE CLUSTER statements.
4. Define the PATH for the base and AIX clusters, using DSIVSMX IDCAMS with the VSAM DEFINE PATH statements.
5. Build the Alternate Index cluster, using DSIVSMX IDCAMS with the VSAM BINDEX statements.
Accessing VSAM Files Using Alternate Keys

The following example code fragments are written in REXX:

- To access the VSAM information using alternate keys, perform the VSAM I/O requests using the PATH cluster. For example:
  - VSAMC320.BASE is the base cluster.
  - VSAMC320.AIX is the Alternate Index cluster.
  - VSAMC320.PATH is the PATH.

- To access the VSAM information using the primary keys, ALLOC the BASE cluster, then attempt VSAM I/O requests using DSIVSMX.

  `ALLOC FILE(BASE) DSN(VSAMC320.BASE)`
  `DSIVSMX vsmx_func BASE <count> <key> <key>`

- To access the VSAM information using the alternate keys, ALLOC the PATH, then attempt VSAM I/O requests using DSIVSMX.

  `ALLOC FILE(PATH) DSN(VSAMC320.PATH)`
  `DSIVSMX vsmx_func PATH <count> <key> <key>`

Where vsmx_func is one of the following DSIVSMX functions:
- OPEN or CLOSE
- GET or GETREV
- PUT
- DEL
- INQUIRE

Deleting an Alternate Index File

To delete the base and Alternate Index clusters, delete the base using DSIVSMX IDCAMS with the DELETE CLUSTER parameters. If you delete the base, the alternate index and the path will also be deleted.

Using the AUTOTOKE Value Provided by DSIVSMX

An AUTOTOKE value is set by DSIVSMX whenever the VSAM file is opened or closed. This value is provided in all messages issued by DSIVSMX, and can be retrieved using the AUTOTOKE() REXX function, or in the output of the DSIVSMX INQUIRE command. To determine that the VSAM file has not been opened or closed since the last I/O, do the following:

1. Save the AUTOTOKE value when the data set is opened. The DSI633I message has the AUTOTOKE() value. Alternately, use DSIVSMX INQUIRE after the DSIVSMX OPEN to retrieve and save the value from the INQUIRE message text.
2. Check the AUTOTOKE value after you issue the DSIVSMX commands. If the value is the same as the saved value, the data set was neither opened nor closed since the last access.

DSIVSAM: Access to Keyed VSAM Files Defined by NetView DSTs

The DSIVSAM command processor can access VSAM keyed files that are defined by NetView data services tasks such as DSILOG. This allows for implementation of all kinds of VSAM applications, including end-use application development in REXX (in conjunction with the pipeline facility) and intensive VSAM diagnostics. For more information about DSIVSAM, refer to the NetView online help.

The DSIVSAM command provides REXX access to any keyed VSAM file on any data services task.
Samples CNMS8016 and CNMS8021 illustrate the use of the DSIVSAM command.

Refer to Volume 2 of the *Tivoli NetView for z/OS Command Reference* or the NetView online help for more information about the DSIVSAM command.

**Using the AUTOTOKE Value Provided by DSIVSAM**

An AUTOTOKE value is set by the data services task (DST) whenever the VSAM file is opened or closed. This value is provided in all messages issued by DSIVSAM, and can be retrieved using the AUTOTOKE() REXX function, or in the output of the DSIVSAM INQUIRE command. If you need to determine that the VSAM file has not been closed or opened since you last did I/O to it, do the following:

1. Use DSIVSAM INQUIRE to retrieve and save the value from the INQUIRE message text.
2. Check the AUTOTOKE value after issuing DSIVSAM commands. If the value is the same as the saved value, the data set was neither closed nor opened since the last access.
Chapter 8. Debugging NetView Pipelines

If your pipelines are not functioning, there are several things that you can do to identify the source of the problem. The pipeline processor examines your pipeline specification for syntax errors, before attempting to run it. When the processor finds syntax errors, it displays error messages.

Understanding Error Messages

Pipeline error messages do not display the name of a failed stage; instead, error messages provide the position of the failed stage in the pipeline specification. If there are multiple syntax errors, the processor displays an error message for the first syntax error found. When that error is corrected, the processor examines the specification for any additional syntax errors.

For example, an operator enters:

```
PIPE NETVIEW CLOSE IMMED | CONSOOOLE
```

The command contains the two stages, NETVIEW and CONSOLE (which is misspelled). The pipeline processor finds the syntax error and displays messages to the console as shown below. Notice that although the DWO362E message does not name the CONSOLE stage, it does indicate that an error was found in the second stage specification in the PIPE command, thus pointing to the CONSOLE stage.

```
NCCF NETVIEW CNM01 OPER6 03/20/95 12:15
* CNM01 PIPE NETVIEW CLOSE IMMED | CONSOOOLE
- CNM01 DWO364E PIPELINE TERMINATED. NO STAGE CONSOOOLE EXISTS.
- CNM01 DWO362E PIPELINE TERMINATED. ERROR IN STAGE 2 IN PIPELINE 'PIPE'.
```

Note: Stage commands inserted with the INTERPRT stage command always have a stage number greater than 10 000. The CLOSE IMMED command did not run because the pipeline did not start. See "PIPE INTERPRT" on page 115 for more information.

Online Help

To display online help for the pipe command, enter:

```
HELP PIPE
```

To display online help for a specific stage command, enter:

```
HELP PIPE stage_command_name
```

Where:

`stage_command_name`

is any NetView PIPE stage command.

See "Chapter 2. Pipeline Stages and Syntax" on page 19 for a list of stages supported by the NETVIEW PIPE command.
Clogged Pipelines

Complex pipelines with multiple data streams can become deadlocked or clogged. Clogs are caused by errors in the structure of the data stream connections.

In a clogged pipeline one stage requires data from another and this data cannot be provided. All pipeline processing stops.

A clog occurs when either:
- A stage needs to give data to another stage that cannot receive the data.
- A stage needs to get data from another stage that cannot provide the data.

The following example creates a clogged pipeline, because it attempts to read a file, convert the last two lines into label lines, and collect all the lines into a multiline message.

```rexx
/* REXX Fragment */
...
'PIPE (NAME CLOGGER END '),
'  | < somemember /* generate data */
'  | A: DROP LAST 2', /* last two lines goto "A" */
'  | LABS: COLLECT', /* data from DROP, labels from "LABS"*/
'  | CONSOLE', /* output, we hope */
'  | \ A:', /* obtain lines from DROP */
'  | LABS:' /* give lines to secondary of COLLECT*/
```

Because this pipeline clogs if the file contains more than three lines, the pipeline comes to a deadlock, or clogs, during the pipeline process. The following steps describe the actions that occur:

1. The first line is read and passed to the DROP stage. DROP holds this line. Because we want to DROP the last two lines, DROP needs to hold the last two lines received.
2. The second line is read and passed to the DROP stage. Again, DROP holds this line.
3. The third line is read and passed to the DROP stage. Here too DROP holds the line. Because the first line received is no longer in consideration of being dropped, DROP passes it to the primary input stream of the COLLECT stage.
4. Because COLLECT must read all lines from its secondary input stream before processing primary input stream data, the message passed by DROP is not yet processed by COLLECT.
5. The fourth line is read from the file and given to DROP. DROP attempts to pass the second line received to COLLECT because it is no longer one of the lines to be dropped.

Here is where the pipeline clogs. COLLECT has a message waiting that it cannot process. DROP has a message that must pass on the same data stream. Most stages do not support data streams containing more than one message at a single time. Enabling only one message at a time preserves the order of messages in complex pipelines.

Only FANIN and FANINANY allow more than one message on a data stream at a given time.

To solve the example problem, you can add FANIN or FANINANY:
/* REXX Fragment */

...

'PIPE (NAME CLOGGER END \)', /* generate data */
' | <' somemember /* last two lines goto "A" */
' | A: DROP LAST 2', /* could also be FANINANY */
' | FANIN', /* data from DROP, labels from "LABS"*/
' | LABS: COLLECT', /* output, we hope */
' | CONSOLE', /* give lines to secondary of COLLECT*/
' \ A:', /* obtain lines from DROP */
' | LABS:' /* do nothing in this example because it has only one input
stream. However, FANIN and FANINANY accumulates, or buffers, data passed to
them if they cannot pass data to their output data stream. By adding FANIN, the
data sent from DROP can be accumulated for future processing by COLLECT.

The DEBUG 2 option helps you to diagnose a clogged pipeline. For more
information about DEBUG 2, see "Clogged Pipeline Details" on page 302.

FANIN appears to do nothing in this example because it has only one input
stream. However, FANIN and FANINANY accumulates, or buffers, data passed to
them if they cannot pass data to their output data stream. By adding FANIN, the
data sent from DROP can be accumulated for future processing by COLLECT.

Tracing Pipelines

Once the pipeline syntax is correct and the pipeline runs, you may face other
problems:
• The pipeline does not produce output
• The pipeline produces incorrect output
• You receive an unexpected message on the terminal.

Most problems can be solved by inspection or by tracing.

Displaying Stage Results

To solve a problem with a pipeline, it is very helpful to see the output from each
stage. Snapshots of the messages flowing through the pipeline can be captured by
copying messages to:
• The terminal with the CONSOLE stage
• The log using the LOGTO stage
• A special message save area using the SAFE stage. Use a named SAFE if you
want to save more than one message.
• Copying messages to command procedure variables using the VAR or STEM
stage.

The contents of the pipeline are not affected by copying messages from the
pipeline using the CONSOLE, LOGTO, SAFE, or STEM stages.

Note: If the CONSOLE stage occurs in the pipeline multiple times, be aware that
the output shown on the screen may be misleading. Stages work
independently and simultaneously, each taking action on the messages as
they are received in their input streams. Where multiple CONSOLE stages
are used, the stages may display the first messages at the same time, then
process its second message, and so forth. Sometimes, this creates confusion
when the user looks at the screen and sees messages from several stages
interleaved. To eliminate this, use the COLLECT stage before a stage that
writes to the log or terminal. COLLECT waits, collecting all messages in the
input stream into one multiline message before passing it on to the next
stage.

Displaying Data Stream Information (DEBUG)
The DEBUG option generates information about the input and output data streams
in a pipeline. DEBUG can be used either as a pipeline option, affecting the entire
pipeline, or on individual stages.

Attention:
- DEBUG output is provided for pipeline problem determination only and is not
  intended as a programming interface. The format, type, and amount of
  information provided by DEBUG is subject to future change.
- DEBUG information is available in English only.

DEBUG Stage Option
DEBUG on a stage generates information about input and output streams
connected to the stage and the messages flowing over those streams.

Consider the following pipeline.
/* REXX Example - Formats output from LIST STATUS=TASKS */

address NETVASIS,
'PIPE (NAME TASKLIST END -)',
  '  | NETV LIST STATUS=TASKS', /* generate the data */
  '  | DROP LAST 1', /* no need of "END OF DATA" */
  '  | COLOR GREEN', /* standardize buffers */
  '  | EDIT WORD 2 1', /* reformat data from the lines */
    ' 19.8 8',
    ' 38.8 19',
    ' 55.* 35',
  '  | LABS: COLLECT', /* data and labels, labels read first */
  '  | CONSOLE ONLY', /* display results */
  '  | FAN: (DEBUG) FANIN', /* feed data to "LABS", in order */
  '  | LABS:', /* --- END of simple pipeline, begin new pipeline... */
  '  | LIT !-------- Status of NetView Tasks ---------!',
  '  | COLOR YEL', /* Control line becomes yellow */
  '  | FAN:', /* give line to "FAN" (primary input) */
  '  " LIT !Task Task's Taskname or Current!", 
  '  | COLOR PINK', /* First label line becomes pink */
  '  | FAN:', /* give line to "FAN" (2nd input) */
  '  " LIT !type ID Resource Status!",
  '  | COLOR PINK', /* Second label line becomes pink. */
  '  | FAN:', /* give line to "FAN" (3rd input) */

Note: For more information, see "Example: Formatting LIST STATUS=TASKS
Output" on page 40.

DEBUG on the stage labeled FAN: generates information about the data streams
connected to FANIN and the messages flowing on those data streams. The
following is an example of the type of problem determination information
produced.

Connecting output number 1 of stage 7, "FANIN "
to input number 2 of stage 5, "COLLECT ".
Connecting output number 1 of stage 9, "COLOR YEL "
to input number 1 of stage 7, "FANIN ".

300  Tivoli NetView for z/OS Customization: Using Pipes
Connecting output number 1 of stage 11, "COLOR PINK "
to input number 2 of stage 7, "FANIN ".
Connecting output number 1 of stage 13, "COLOR PINK "
to input number 3 of stage 7, "FANIN ".
Stage 7 reads input from stream 1, "Status of NetVi".
Stage 7 writes output to stream 1, "Status of NetVi".
Stage 7 reads input from stream 2, "Task Task's ".
Stage 7 writes output to stream 1, "Task Task's ".
Stage 7 reads input from stream 3, "type ID ".
Stage 7 writes output to stream 1, "type ID ".
Terminating stage number 7, "FANIN ".

The first lines show the FANIN input and output streams and connected stages. Each stage is shown along with its stage number within the pipeline.

Next, the first few characters of each message flowing on the data streams being debugged are shown. Each line indicates the stage number and whether the message is flowing on an input or output data stream. The data stream number is also included.

**Note:** In this example with only one DEBUG option in the pipeline, it is obvious that the FANIN stage is being traced without having to explicitly specify "Stage 7". However, the stage number is important when multiple DEBUG options are coded in a single pipeline. The stage number, in that case, indicates the stage for which activity is being traced.

Finally, the last message shows that the stage termination conditions have been met and that FANIN is terminating. For information on stage termination conditions, see the individual stage descriptions in "Chapter 2. Pipeline Stages and Syntax" on page 19.

The following coding rules apply to DEBUG as a stage option.

- DEBUG can be added to any pipe stage after the label and immediately before the stage command.
- The DEBUG option must be enclosed in parenthesis.
- If you need to include DEBUG on the first stage in the pipeline and you have not specified pipeline options, include an empty pipeline option string before DEBUG. For example, the following command cannot be used:
  ```
  PIPE (DEBUG) NETV LIST X|...
  ```
  However, either of the following commands will work:
  ```
  PIPE () (DEBUG) NETV LIST X|...
  PIPE | (DEBUG) NETV LIST X|...
  ```
  - For stage commands modifying the action of other stages, for example NOT and CASEI, include DEBUG before the modifier. For example, use the following command:
    ```
    PIPE ...|(DEBUG) CASEI LOCATE /X/|...
    ```
    But, not this:
    ```
    PIPE ...|CASEI (DEBUG) LOCATE /X/|...
    ```

**DEBUG Pipeline Option**

DEBUG can also be included in the pipeline options. When included as a pipeline option, DEBUG affects all stages within the pipeline.
Data Stream Tracing: Coding DEBUG 1 in the pipeline options is the same as coding DEBUG on each pipeline stage. For example:

```
PIPE (NAME SAMP DEBUG 1)
  < MYFILE
  STRIP TRAILING / /
  JOINCONT TRAILING /$/
  CONSOLE
```

Is the same as:

```
PIPE (NAME SAMP)
  (DEBUG) < MYFILE
  (DEBUG) STRIP TRAILING / /
  (DEBUG) JOINCONT TRAILING /$/
  (DEBUG) CONSOLE
```

For details on the output produced by DEBUG 1, see “DEBUG Stage Option” on page 301.

Clogged Pipeline Details: When you receive message BNH155E indicating that a complex pipeline is clogged, it might be difficult to determine the cause of the clog. The DEBUG 2 pipeline option generates additional debugging information when BNH155E is issued.

For example, the following pipeline will clog:

```
/* REXX Example */

'PIPE (NAME CLOGGER END \ DEBUG 2)',
  '< somemember /* generate data */
  ' A: DROP LAST 2', /* last two lines go to "A" */
  ' LABS: COLLECT', /* data from DROP, labels from "LABS" */
  ' CONSOLE', /* output, we hope */
  '
  ' A:', /* obtain lines from DROP */
  ' LABS:' /* give lines to secondary of COLLECT */
```

The DEBUG 2 option generates the following debugging information following message BNH155E.

```
< (1) waiting to output to DROP (2) on stream 1
DROP (2) waiting to output to COLLECT (3) on stream 1
COLLECT (3) awaits input from DROP (2) on stream 2
CONSOLE (4) awaits input from COLLECT (3) on stream 1
```

Each line shows the stage and, in parenthesis, its stage number. The state of each data stream at the time of the clog is also shown.

The debug information shows where the example pipeline clogs. DROP is trying to pass input to COLLECT on stream 1, but COLLECT is expecting input from DROP on stream 2.

The word awaits in DEBUG 2 information is a good indication of where a clog is occurring.
Displaying Return Codes

PIPE command return codes, as specified in “Chapter 2. Pipeline Stages and Syntax” on page 19, can be retrieved within a command list by accessing the return code through a command list control variable, immediately after issuing the PIPE command.

By adding the MOE (message on error) option to the NETVIEW, VTAM, or CORRWAIT stages, nonzero return codes can be captured. This return codes follow execution of a command or as an indicator of a timeout when waiting for asynchronous messages to return from another application, such as MVS or VTAM, or from another NetView program. The return code is embedded within the DWO369I message.

Additional Trouble Shooting Tips

If you are not getting the output you expect from your pipeline, check for these possible causes:

- The HOLE stage is discarding all pipeline messages.
- The CORRWAIT stage is not used following a command-executing stage, such as NETVIEW or VTAM, causing asynchronous messages to be lost because no time interval is allowed for their return.
- Pipeline messages are not generated by the stage preceding the NETVIEW stage; therefore, the command issued by the NETVIEW stage is not triggered to execute.
- In a PIPE-within-a-PIPE structure, the CORRWAIT stage is not used, or is used incorrectly, following the CONSOLE stage, which is used to return messages from a remote pipeline to a local pipeline. The function of the CORRWAIT in the local pipeline is to wait for messages to travel from the remote to the local NetView. The CORRWAIT stage must be used. If it is already being used increase the interval. In addition, you may want to add the MOE option to check for a timeout. There must be enough time allowed for the messages to travel from the remote to the local NetView.
- You have multiple pipelines in a command list or a PIPE-within-a-PIPE structure, but you have not used the NAME option to distinguish them.
Appendix. Additional NetView Pipeline Examples

This appendix documents general-use programming interface, associated guidance information, and contains examples that show how stage commands can be combined effectively.

Displaying Part of a Lengthy Automated Message

Displaying a small part of a long automated message, while continuing to preserve the DOM criteria, can be done using a PIPE command.

An example may be a tape mount request message, IEF290E, which can be automated to display at the NetView operator’s terminal. Only the first few lines may be of interest.

In this example, a job called BLDTAPE that requested a tape mount was submitted. The panel below shows the message as it is displayed before any manipulation is performed to decrease its length and content. Note the long list of available devices in the message.

```
NCCF  NETVIEW CNM19 OPER6  03/20/99  14:00:00
13:20 17:10 * CNM19 MVS S BLDTAPE * CNM19 IEF290E BLDTAPE
SYST2 NEEDS 1 UNIT(S) FOR VOLUME VSAM01
OFFLINE  NOT ACCESSIBLE
D=   D=  505, 506, 507, 508, 509
D=   D=  50A, 50B, 50C, 50D, 50E
D=   D=  50F, 516, 517, 518, 519
D=   D=  51A, 51B, 51C, 51D, 51E
D=   D=  51F, 520, 521, 522, 523
D=   D=  524, 525, 526, 527, 528
D=   D=  529, 52A, 52B, 52C, 52D
D=   D=  52E, 52F, 530, 531, 532
D=   D=  533, 534, 535, 536, 537
D=   D=  538, 539, 53A, 53B, 53C
D=   D=  53D, 53E, 53F, 540, 541
D=   D=  542, 543, 544, 545, 546
D=   D=  547, 548, 549, 54A, 54B
D=   D=  551, 552, 553, 554, 555
D=   D=  556, 557, 558, 559, 55A
D=   D=  55B, 55C, 55D, 55E, 55F
D=   D=  560, 561, 562, 563, 564
```

Figure 31. Job BLDTAPE Example

The message is automated through the following automation table entry, which routes a command list named PIPEMSG to OPER6.

```
IF MSGID = 'IEF290E' .
   THEN EXEC(ROUTE(ONE OPER6) CMD('PIPEMSG'));
```

The PIPEMSG command list shown here uses the PIPE command to reduce the message to two lines before displaying. The stage named SAFE reads the message buffers which were automated. The message read is an MLWTO. The second stage, SEPARATE, breaks the MLWTO into individual lines. The TAKE stage selects the
Additional NetView Pipeline Examples

first 2 lines, while discarding other messages from the pipeline. The last stage, CONSOLE, displays the pipeline contents to the operator’s console.

PIPEMSG CLIST
&CONTROL ERR
****************************************************************
** CLIST TO READ THE MESSAGE FROM THE SAFE, BREAK THE MLWTO **
** INTO MULTIPLE LINES, SELECT THE FIRST FEW LINES AND **
** DISPLAY RESULTS **
****************************************************************
PIPE SAFE *
| SEPARATE
| TAKE 2
| CONSOLE
&EXIT

Output from the pipeline follows:

Figure 32. Modified Job BLDTAPE Example

Notice the two separate action messages on the screen. When the mount request is satisfied, MVS sends a DOM that matches the original message. This matches both of these messages and both are deemphasized by the DOM. You can add a COLLECT stage to your pipeline to make a 2-line MLWTO, which also matches the DOM.

Transferring a Large Variable to Another Task

An operator can transfer a REXX variable that is longer than 256 bytes to another task by using PIPE commands at both the sender and receiver tasks.

The sender, OPER3, runs the EXECSEND REXX command list that contains a PIPE command. First, the PIPE command loads the content of LONGVAR into the pipeline using the VAR stage. The pipe tells the OPER6 task to invoke an command list named EXECRECV with the EXCMD command. EXCMD is a command that is sensitive to the current pipeline message and sends it to OPER6.

The command list that is submitted by OPER3 is shown in the next example:

/* REXX COMMAND LIST EXECSEND */
ADDRESS NETVASIS
/*******************************************************************/
/* CREATE A VARIABLE NAMED LONGVAR WHICH IS LONGER */
/* THAN 256 CHARACTERS */
/*******************************************************************/
LONGVAR = 'LOOK AT THIS LONG MESSAGE THAT I CREATED. IT IS WELL'
          ' OVER 256 CHARACTERS. AAAAAAAAAAAAA BBBB BBBB BBBB BBBB'
          ' CCCCCCCCCCCCC DDDDDDDDDDDDDDD EEEEEEEEEEEEEE FFFFFF'
          ' FFFFFFFFF GGGGGGGGGGGGGG HHHHHHHHHHHHHH ILLUUUUUUUU'
          ' III JJJJJJJJJJJJJJKKKKKKKKKKK LLLLLLLLLLLLL MM'
          ' NNNNNNNNNNNN 0000000000000000 PRRRRRR'
          ' QQQQQQQQQQQQQQQQ'
/*******************************************************************/
/* READ THE VARIABLE INTO THE PIPELINE. */
*******************************************************************/
The results displayed on the terminal of OPER3, the sender, follows:

```
NCCF NETVIEW CNM19 OPER3 02/01/99 10:43
* CNM19 EXCESEND
C CNM19 TRANSFER SENT SUCCESSFULLY
```

Figure 33. Tranfer Send Results Screen

At OPER6, EXECRECV, which also contains a PIPE command, runs. The PIPE command reads the command procedure message queue to the pipeline using the SAFE stage. Then the VAR stage writes the pipeline contents to the variable named OP6VAR. Lastly, the CONSOLE stage displays the pipeline contents to the terminal.

The command list that is invoked at OPER6’s task is:

```
/* SAMPLE REXX COMMAND LIST NAMED EXECSEND */
ADDRESS NETVASIS

/* USE PIPE TO READ THE CURRENT MESSAGE INTO THE PIPELINE, STORE IT INTO A VARIABLE NAMED OP6VAR, AND DISPLAY TO TERMINAL */
'PIPE SAFE *
  'VAR OP6VAR',
  'CONSOLE'

IF RC ~= 0 THEN
  SAY 'RC="RC" FROM PIPE'
IF RC = 0 THEN
  SAY 'TRANSFER RECEIVED SUCCESSFULLY'
EXIT
```

The results displayed on the terminal of OPER6, the receiver, are:
Additional NetView Pipeline Examples

The PIPE command can be used to search for up to five authorized program analysis reports (APARs) or program temporary fixes (PTFs) on the host NetView system, as shown in the following command procedure:

```rexx
/* Sample REXX command list Named: NVMAINT */
/*******************************************************************/
/**/* Be careful when searching for an APAR/PTF that has been *//* superseded by a different APAR/PTF. *//* *//* Syntax is: NVMAINT apar1 ptf2 apar3 ptf4 apar5 *//* *//* *******************************************************************/
ARG fix.1 fix.2 fix.3 fix.4 fix.5
IF fix.1 = '' THEN
  SAY 'Please supply an APAR or PTF, for example: NVMAINT UY86627'
  EXIT
END
srchtxt = '/* Loop through passed arguments to set up search string. *//* Will search for APAR/PTF in columns 43-49 of DISPMOD output */*******************************************************************/
DO i=1 to 5
  IF fix.i ^= '' then
    srchtxt = srchtxt || ' 43.7 'fix.i''
  END
END
/* SEParate DISPMOD MLWTO output into single lines, *//* LOCate APARs/PTFs supplied as arguments *//* COLlect the matched lines into a single MLWTO *//* Display to the CONsole ONLY *//* (do not put into NetView log or automate) *//* *******************************************************************/
'PIPE NETV DISPMOD ALL ALL',
  'SEP',
  'LOC 'srchtxt,
  'COL',
  'CONS ONLY'
EXIT
```

Figure 34. Transfer Received Results Screen

Note: The data could also be sent to a task in a remote NetView. See "PIPE CORRCMD" on page 43.

Searching for APARs and PTFs

The PIPE command can be used to search for up to five authorized program analysis reports (APARs) or program temporary fixes (PTFs) on the host NetView system, as shown in the following command procedure:

```rexx
/* Sample REXX command list Named: NVMAINT */
/*******************************************************************/
/**/* Be careful when searching for an APAR/PTF that has been *//* superseded by a different APAR/PTF. *//* *//* Syntax is: NVMAINT apar1 ptf2 apar3 ptf4 apar5 *//* *//* *******************************************************************/
ARG fix.1 fix.2 fix.3 fix.4 fix.5
IF fix.1 = '' THEN
  SAY 'Please supply an APAR or PTF, for example: NVMAINT UY86627'
  EXIT
END
srchtxt = '/* Loop through passed arguments to set up search string. *//* Will search for APAR/PTF in columns 43-49 of DISPMOD output */*******************************************************************/
DO i=1 to 5
  IF fix.i ^= '' then
    srchtxt = srchtxt || ' 43.7 'fix.i''
  END
END
/* SEParate DISPMOD MLWTO output into single lines, *//* LOCate APARs/PTFs supplied as arguments *//* COLlect the matched lines into a single MLWTO *//* Display to the CONsole ONLY *//* (do not put into NetView log or automate) *//* *******************************************************************/
'PIPE NETV DISPMOD ALL ALL',
  'SEP',
  'LOC 'srchtxt,
  'COL',
  'CONS ONLY'
EXIT
```

Figure 35. Searching for APARs or PTFs with a PIPE command
Displaying Task Information Summary

An operator can create a summary of information for a task by combining PIPE commands within a command procedure. This example shows how to create a summary for the DSIGDS task:

```rexx
/* Sample REXX command list Named: GDSSUM */
/* Display a summary of information for the DSIGDS task */
/* GDSSUM will display for the task: */
/* 1) DSIGDS status, active or not active (from LIST DSIGDS) */
/* In addition, if the task is active, GDSSUM will display: */
/* 2) which NetView proc has the DSIGDS interface with VTAM */
/* (from DIS DSIGDS) */
/* 3) DSRB usage for DSIGDS (from DSRBS DSIGDS). */
/* 4) storage, CPU usage and buffer queue for DSIGDS (from */
/* TASKUTIL DSIGDS) */

*******************************************************************/

/* Issue a LIST DSIGDS command. Select the first returned message */
/* which contains activity status information. Store the message */
/* in both a safe named GDSSAFE and a variable named STATMSG. */

'PIPE NETV LIST DSIGDS',
    '| TAKE 1',
    '| SAFE GDSSAFE',
    '| VAR STATMSG'

/* Parse the variable STATMSG so that the variable named ACTSTAT */
/* will contain either the word 'ACTIVE' or the word 'NOT' */
/* reflecting the status of the task. */

PARSE VAR STATMSG . 'STATUS:' ACTSTAT .

IF ACTSTAT ¬= 'NOT' THEN
    DO;
        /* Status = ACTIVE */
        /* Issue the DIS DSIGDS command and allow sufficient time for */
        /* asynchronous messages to return from VTAM. Break the */
        /* resulting MLWTO into single-line messages. Locate the */
        /* message containing IST271 which will identify the NetView */
        /* proc that has the use of the DSIGDS VTAM APPL. Append */
        /* the message to the contents of the safe. */

        'PIPE NETV DIS DSIGDS',
            '| CORR 2',
            '| SEP',
            '| LOC 1.7 \IST271\',
            '| SAFE GDSSAFE APPEND'

        /* Issue the DSRBS DSIGDS command and break the resulting */
        /* MLWTO into single-line messages. Discard the first 2 */
        /* messages. Keep other messages up to and including the */
        /* message containing the word 'TOTAL'. Append the messages */
        /* to the contents of the safe. */

        'PIPE NETV DSRBS DSIGDS',
            '| SEP',
            '| DROP 2',
            '| TOS 8.13 \TOTAL\',
            '| SAFE GDSSAFE APPEND'

        /* Issue the TASKUTIL command to show the storage, cpu and */
        /* buffer queue for DSIGDS. Break the resulting MLWTO into */
```
Addtional NetView Pipeline Examples

/* single-line messages. Discard the first message. Keep the */
/* next 3 messages in the pipeline while discarding any that */
/* follow. Add a blank line to the pipeline and finally, */
/* append the messages to the contents of the safe. */

'PIPE NETV TASKUTIL DSIGDS',
  'SEP',
  'DROP 1',
  'TAKE 3',
  'LIT \\',
  'SAFE GDSSAFE APPEND'

END;

/* Read everything stored in GDSSAFE into the pipeline. */
/* Combine all output into a single MLWTO. Clear the screen and */
/* display to the user. Do not log or automate from the displayed */
/* output. */

'PIPE SAFE GDSSAFE',
  'COL',
  'CONS CLEAR ONLY'
EXIT

Output from the pipeline (DSIGDS in active status) follows:

NCCF NETVIEW CNM01 OPER1 05/17/99 12:20:00
- CNM01
  TYPE: OPT TASKID: DSIGDS TASKNAME: DSIGDS STATUS: ACTIVE
  IST277I JOBNAME = MYENV, STEPNAME = MYENV, DSPNAME = 00009IST
  UNSOLICITED DSRBS: 1 USED: 0 FREE: 1
  SOLICITED DSRBS: 5 USED: 0 VSAM REDRIVE: 0 FREE: 5
  TOTAL DSRBS: 6 USED: 0 VSAM REDRIVE: 0 FREE: 6
  TASKNAME TYPE DPR CPU-TIME %CPU %S-CPU% MESSAGEQ STORAGE-K CMDLIST
  ------- ---- ------ --------- -------- --------- --------
  DSIGDS DST 254 0.01 0.00 0.00 0 35 N/A

Figure 36. DSIGDS Task Summary Screen

Displaying or Canceling a JES2 Job

Multiple PIPE commands can be combined in a REXX command list to enables a user to display, or to display and cancel, a job on JES 2. A wildcard character (*) is supported at the end of jobname to display or cancel multiple jobs.

This example shows output from the command being run twice. The first time the command list was submitted using the command 'JES2JOB REOR* D' to display all JES 2 jobnames starting with the characters 'REOR'. The second time the command list was submitted using the command 'JES2JOB REORGA C' to display and cancel the job REORGA.

A customized screen format is used for this example. The JES2 job number displayed. Browse the NetView-supplied sample CNM0SCNFT for more information.
Note: APAR OY52941 must be applied for the JES commands in this command list to have correlated output.

Additional NetView Pipeline Examples

/* Sample REXX command list Named: JES2JOB */
/*
/* Syntax: JES2JOB parm1 parm2
/* where:
/* parm1 is one of the following:
/* jobname (A JOBNAME )
/* jobn* (a partial jobname followed by an asterisk)
/* * (an asterisk, meaning 'all')
/* parm2 is one of the following:
/* C (cancel)
/* any character other than 'C' (indicates display)
/**/

ARG jobname arg2 /* display/cancel parm. */
IF (jobname = '') | (jobname = '?') THEN DO
   SAY 'Enter BR JES2JOB for help and syntax'
   EXIT
END

i = LENGTH(jobname)
IF SUBSTR(jobname,1,1) = '*' THEN /* If using wildcard */ DO
   jobname = STRIP(jobname,T,'*') /* Remove '*' from jobname.*/
   i = i - 1 /* Decrement removed '*' from leng.*/
END ELSE /* If not using wildcard */ DO
   jobname = jobname || ' ' /* Add ' ' to jobname so we do not */ /* match other jobs that start with */ /* the characters of our jobname. */
   i = i + 1 /* Increment added ' ' to length. */
END

/* Issue MVS $DA command on the */ /* NETVIEW stage to get info about */ /* active batch jobs and jobs */ /* waiting for resources. The cmd */ /* results will come back to the */ /* pipeline as 1 MLWTO (L=Z). */ /* */ /* Wait 30 seconds for $DA output. */ /* Terminate wait when response */ /* received. */ /* */ /* Split up lines of MLWTO. */ /* Discard the first line. */ /* Locate msgs containing jobname. */ /* Store matching lines in */ /* stem variable named jobline */ /* */ /* Issue MVS $DN command on the */ /* NETVIEW stage to get info about */ /* jobs waiting for execution. The */ /* results will come back to the */ /* pipeline as 1 MLWTO (L=Z). */ /* Wait 30 seconds for $DN output. */ /* Terminate wait when response */ /* received. */
Additional NetView Pipeline Examples

```
' | SEP', /* Split up lines of MLWTO. */
' | DROP FIRST 1', /* Discard the first line & last */
' | DROP LAST 2', /* 2 lines of $DN,ALL response. */
' | LOC 10.'i+10' \'jobname\', /* Locate msgs containing jobname. */
' | STEM jobline. APPEND' /* Append matching lines to con-
' | tents of stem variable jobline. */
IF jobline.0 = 0 THEN /* if no matches, say so and exit */
DO
  SAY 'No jobs found'
  SIGNAL GETOUT
END

'/**********************************/
/* Info on matching jobs creating */
'/**********************************/

'/**********************************/
/* Display output */
'/**********************************/

'PIPE STEM jobline.', /* output into pipeline. */
  ' | COL', /* Gather all output into 1 MLWTO. */
  ' | CONS ONLY' /* Display. Do not automate orlog.*/
'/*******************************************************************/
/* Uncomment the following section to display more detailed */
/* information on each job. */
'/*******************************************************************/

'/**********************************/
/* Display output (detailed) */
'/**********************************/

/*
DO j = 1 TO jobline.0 /* Go thru matched job info.*/
  jobnum = SUBSTR(jobline.j, 4, 5) /* Get job number. */
  'PIPE NETV MVS $DJ'jobnum',LONG', /* Put JES display in pipe. */
    ' | CORR 30', /* Wait for JES response. */
    ' | TAKE 1', /* Terminate wait when */
      ' | SEP', /* response received. */
    ' | DROP 1', /* Discard first line only */
    ' | TAKE 1', /* Keep next line */
    ' | SAFE cmdsafe APPEND' /* Store for later display. */
END
'PIPE SAFE cmdsafe', /* Display detailed output. */
  ' | CONS ONLY' /* Do not automate or log. */
/*

'/**********************************/
/* 2nd Parm is CANCEL */
'/**********************************/

IF SUBSTR(arg2,1,1) = 'C' then
DO
  DO j = 1 TO jobline.0 /* Go thru all matched jobs.*/
    jobnum = SUBSTR(jobline.j, 4, 5) /* Get job number. */
    'PIPE NETV MVS $CJ'jobnum',P', /* Cancel job, purge output */
      ' | CORR 30', /* Wait for JES response. */
      ' | TAKE 1', /* Terminate wait when */
        ' | SAFE cmdsafe APPEND' /* response received. */
    ' | SAFE cmdsafe', /* Display output from JES */
      ' | CONS ONLY' /* cancel or display cmds. */
/*
```

Output from the JES2JOB display command list follows:

```
NCCF NETVIEW CNM01 OPER1 05/02/99 17:20:30 A
* CNM01 JES2JOB REOR* D
| CNM01
| JOB00049 REORGE EXECUTING A PRIO 10 8609
| JOB00055 REORGAA EXECUTING A PRIO 10 8609
| JOB00056 REORGB EXECUTING A PRIO 10 8609
| JOB00057 REORCG EXECUTING A PRIO 10 8609
| JOB00050 REORGE AWAITING EXECUTION A PRIO 10 DUPLICATE ANY
| JOB00051 REORGE AWAITING EXECUTION A PRIO 10 DUPLICATE ANY
```

* Figure 37. JES2JOB Display Command Output Example *

Output from the JES2JOB cancel command list follows:

```
NCCF NETVIEW CNM01 OPER1 03/01/99 17:20:30 A
* CNM01 JES2JOB REORGA C
| CNM01
| JOB00054 REORG AWA ITING HARDCOPY PRIO 1 CANCEL ANY
| $HASP608 REORG AWA ITING PURGE PRIO 1 PURGE ANY
```

* Figure 38. JES2JOB Cancel Command Output Example *
Additional NetView Pipeline Examples
Index

Special Characters

* (asterisk)  
   LOGTO option 127  
   SAFE option 171  
< (from disk) stage command 232  
   reading file contents into pipelines, example 233  
$STEM stage command  
   description 189  
   saving records processed counts, example 192  
   writing, stemmed variables, example 192  
$VAR stage command 214

A

accessibility information ix  
action key, VET option 223, 274  
ALL, LOGTO option 127  
APAR (authorized program analysis report) 308  
APPEND  
   $STEM option 190  
   SAFE option 171  
   STEM option 190  
APPEND stage command 27  
   asynchronous messages  
      converting, synchronous 139  
      definition 49  
      handling, CORRWAIT 51  
ATTACH command 271  
   authorized program analysis report (APAR) 308

B

BETWEEN stage command 29  
BNH150I  
   FIELDS format 278  
   ROWS format 277  
books  
   feedback vii  
   online vii  
   ordering vii

C

CART (command and response token) 43, 139  
CASEI stage command 31  
CC stage command (CORRCMD synonym) 45  
CCDEF 46, 50  
CHANGE stage command 33  
CHOP stage command 35  
CLEAR, CONSOLE option 42  
   clearing screens, CONSOLE CLEAR option 42  
closed pipelines 298  
CNMS1098 sample 281  
CNMS1101 24  
COLLECT option  
   $STEM command 190  
   alternative, COLLECT command 39  
   STEM command 190  
COLLECT stage command  
   avoiding misleading output from CONSOLE 299  
   collecting single-line messages 255  
   stage description 37  
   command and response token (CART) 43, 139  
   compare and swap 217  
   complex pipelines 7  
   complex pipelines, creating 7  
   CONSOLE stage command  
      avoiding misleading output 299  
      stage description 42  
      taking snapshots, pipeline messages 299  
      task interaction 235  
CORRCMD stage command 45  
   correlated output, HOLE stage command 110  
   correlation method for converting asynchronous messages 139  
CORRWAIT stage command 49  
   COUNT stage command 54  
   CPDOMAIN stage command 58  
   CURRENT, VET option 223, 273  
   Customer Support viii  
device drivers, placement dependency 235  
disability information ix  
DISKONLY, < (from disk) option 233  
   display field attributes 278  
   displaying  
      pipeline messages, CONSOLE stage 235  
DIVERT stage command 63  
   DOM (delete operator message) 42  
   double-byte character set data 42  
DROP stage command  
   example, discarding messages 65  
   selecting messages, position 263  
   stage description 64  
DUMP, CONSOLE option 42  
DUMP keyword, ATTACH 281  
   Duplicat stage command 66

E

e-mail contact viii  
EDIT stage command 68  
end character, pipeline 9  
END parameter, PIPE 20  
ENVDATA stage command 98  
ESC parameter, PIPE 20  
   examining return codes, CORRCMD stage command 46  
   example, NPDA automation 282  
   examples  
      $STEM stage command 192  
      alert to Tivoli Event Console 97  
      APPEND stage command 28  
      automation table sample 208  
      BETWEEN stage command 30  
      breaking multiline messages 175  
      building pipeline specifications 116  
      building pipelines, INTERPR0T 253  
      CASEI stage command 31  
      causing a wait 51  
      CHANGE stage command 34  
      changing separation character 23  
      CHOP stage command 36  
      CNMS1101 24  
      COLLECT stage command 39, 233  
      collecting single-line messages into multiline message 256  
      comparing and setting variables 219  
      comparing values contained in two stems 132  
      compiling and executing a Java sample 212  
      CONSOLE stage command 44  
      converting ASCII text 231  
      converting single-line messages to multiline 39  
      copying task globals 219  
      CORRWAIT stage command 51  
      COUNT stage command 56  
      counting comment lines in files 233
examples (continued)
counting messages 56
CPDOMAIN stage command 59
creating a command 96
creating four consecutive utilization reports 67
creating named SAFEs, NULL messages 172
DELDUPES stage command 61
deleting held messages 108, 236, 239
determining, command response correlated to command 265
determining, named SAFE 171
discarding messages 65
messages by content 142
messages containing specified text strings 259
pipeline contents 111
discarding first messages 264
discover TSO stacks serving a user 205
display last logtime 61
displaying messages in dump format 44
displaying messages on consoles 44
displaying part of automated message 305
displaying results and avoiding logging 235
displaying translated messages 143
dividing text 180
DROP stage command 65
DUPLICAT stage command 67
EDIT stage command 96, 97
determining, ISTORE stage 114
determining a wait, TOSTRING 200
ENVDATA stage command, environment data 99
ENVDATA stage command, genealogy data 99
executing commands, DSIPARM 212
generating return codes, NETVIEW stage 243
generating return codes, VTAM stage 245
handling asynchronous messages 51
HELDMSG stage command 108
hiding members, procedure family 114
HOLE stage command 110, 111
IDLEOFF 132
inserting and logging text strings 128
inserting multiple text strings 240
inserting text, embedded command list function 240
inserting text strings, pipelines 124
INSTORE stage command 114
INTERPRT stage command 116, 117
issuing commands 140
issuing MVS commands 242
issuing VTAM commands 245
issuing VTAM commands at remote domains 229
keeping first messages, pipeline 263
keeping last messages, pipeline 264
examples (continued)
keeping messages, multiple text strings 258
keeping messages, specified text strings 258
listing current UNIX working directory 212
LITERAL stage command 124
loading members, disk into storage 114
LOCATE stage command 126, 140, 233
locating messages, content 126
locating messages by content 150
LOGTO stage command 128
LOOKUP stage command 132
managing load members (INSTORE) 114
MLWTO processing with NLOCATE 260
multiple CONSOLE stages 237
multiple CONSOLE stages, COLLECT stage 238
named SAFE as message queue 172
NLOCATE stage command 142
NLS stage command 143
NOT stage command 146
passing messages, second PIPE command 172
PICK stage command 150
PIPEND stage command 153
processing MLWTOS, TOSTRING 261
processing pipeline data 243
processing pipeline messages 244
processing single-line messages, TOSTRING 262
QSAM stage command 165
reading, DSICLCL ddname data sets 248
reading and writing messages, SAFE 251
reading data from DASD 247
reading file contents into pipelines 233
reading variables, STEM 250
reset line counts 56
REVERSE stage command 167
routing held messages 238
running large pipelines 117
running VTAM commands, remote domains 246
SAFE stage command 171, 172
saving records processed counts 192
searching, APARs and PTFs 308
selecting, words 96
selecting last message 199
selecting single-line messages, multiline messages 175
sending system log messages 118
SEPARATE stage command 175
separating data lines 175
separating multiline messages, single-line messages 255
setting variables 219
SORT stage command 177
sorting 177
examples (continued)
SPLIT stage command 180
splitting text 180
STEM stage command 192, 216, 231, 233
TAKE stage command 199
terminating a CORRWAIT 52
TOSTRING stage command 200
transferring large variables 306
translating an ASCII value to EBCDIC 231
TSO stage command 205
TSROUTE stage command 208
understanding error messages 117
UNIX stage command 212
unloading member, storage 114
update current group members 220
using XLATE COMMON 231
VARLOAD stage command 219, 220
waiting five seconds 110
waiting for messages 52
writing, named variables 216
writing, stemmed variables 192, 249
writing null messages, STEM 251
explicit routing 227
EXPOSE stage command 100
exposed message 42
examples (continued)
FANIN stage command 102
FANINANY stage command 104
FANOUT stage command 106
feedback about publications viii
FIELDS, VET option 224, 273
filters, pipeline 255
first stage, pipelines 6
full-screen automation 267
action key 274
BNH150I 276
FIELDS format 278
ROWS format 277
debugging, DUMP keyword, ATTACH 281
example program 282
interacting, full-screen applications 272
other messages 280
partial screens 281
TAF sample (CNMS1098) 281
VET 222, 272
VOST
attaching a VOST 271
detaching a VOST 272
VOST, definition 270
G
graphic character strings 290
H
HCYLOG, LOGTO option 127
held messages, HELDMSG 108
HELDMSG stage command example, deleting held messages 108
HELDMSG stage command (continued)
  task interaction 235
  hold status, reversing 235
HOLE stage command 110

IBM DATABASE 2, MVS 285
implicit routing 227
INCL, < (from disk) option 233
information, accessibility ix
information, disability ix
input strings, literals 20
INSTORE stage command
description 112
  ending, INSTORE stage, example 114
  hiding members, procedure family, example 114
  loading members, disk into storage, example 114
INTERPRT stage command
building pipeline specifications, example 116
description 115
error messages 116
managing load members, example 114
running large pipelines, example 117
IPLOG stage command
description 118

JOINCONT stage command 119

KEEP stage command 122
keyboard, shortcut keys ix

L
label, pipeline 7, 9
label parameter, PIPE 20
limitations
  controlling messages, multiple
    CONSOLE stages 43
delimiters 23
  held messages, autotask 108
length restrictions, handling 115, 118
messages deleted, displaying 43
  search strings 23
  terminating CORRWAIT 52
literal
  inserting, pipelines 124
  treating command input 20
LITERAL stage command
description 124
  inserting text strings, pipelines, example 124
  task interaction 235
LOCATE stage command
  counting comment lines in files, example 233
description 125
LOCATE stage command (continued)
  issuing commands, example 140
  locating messages, content, example 126, 257
LOCK, CONSOLE option 42
  locking displays 42
logging, pipeline contents 127
LOGTO stage command
description 127
  taking snapshots, pipeline messages 299
  task interaction 235
long running command (LRC) method,
  converting asynchronous messages 139
LOOKUP stage command 129
LRC (long running command) method,
  converting asynchronous messages 139

M
manuals
  feedback vii
  online vii
ordering vii
MEMLIST stage command 133
message BNH150I 276
message on error (MOE)
capturing nonzero return codes 303
CORRWAIT option 51
  option, CORRCMD stage command 46
  option, MVS stage command 135
  option, NETVIEW stage command 138
VTAM option 228
messages
  consumed 5
  exposed 42
  messages, pipeline asynchronous
    CORRWAIT 51
definition 49
  collecting single-line messages 255
  converting multiline, single-line 174
determining correlation (HOLE) 265
discarding, by content 64, 141, 257
displaying 42
  holding, exposing 42
  panels 235
  exposed 42
MLWTOs, NLOCATE 141
primary input to output 63
releasing held status 235
removing held status 42
returning to caller 235
searching 125
MLWTOs, LOCATE 125
MLWTOs, NLOCATE 141
MLWTOs, PICK 149
selecting 125
  by content 125, 149, 257
  by position 257
separating multiline messages 255
  snapshots, held messages 108
  storing 171
MLWTO (multiline write-to-operator) message, converting 39
MOE (message on error)
capturing nonzero return codes 303
CORRWAIT option 51
  option, CORRCMD stage command 46
  option, MVS stage command 135
  option, NETVIEW stage command 138
VTAM option 228
multiline write-to-operator (MLWTO) message 39
multiple stream pipelines 7
MVS stage command 135

N
NAME parameter, PIPE 20
named SAFE 170
naming pipelines 20
NETLOG, LOGTO option 127
NetView pipelines 1
  changing separation characters 20
description 1
device drivers 235
  placement dependency 235
task interaction 235
discarding contents 110
filters 255
general concepts 1
  input strings, literals 20
  logging contents 127
  naming 20
  reading data, DASD 232
  running commands 137
  running NetView, MVS, or local VTAMcommands 135
  running VTAM DISPLAY, VARY, and MODIFY, remote domains 227
trouble shooting 21, 299
NETVIEW stage command 137
NEXT, VET option 224, 273
NLOCATE stage command
description 141
discarding messages, by content 142
discarding messages by content 257
displaying translated messages, example 143
NLS stage command 143
NOT stage command 145
NPDA automation example 282

O
online publications viii
ONLY, CONSOLE option 42
ordering publications viii

P
partial screens, full-screen automation 281
passing messages, REXX command list 172
PERSIST stage command 147
Index 317
PICK stage command 149
   locating messages by content, example 150
PIPE command 20
   pipeline
      end character 9
      label
      connector 9
   pipelines 1
      changing separation characters 20
      complex pipelines 7
      complex pipelines, creating 7
      description 1
      device drivers 235
      placement dependency 235
      task interaction 235
   discarding contents 110
   filters 255
   first stage 6
   general concepts 1
   input 4
   input strings, literals 20
   label 7
   logging contents 127
   naming 20
   output 4
   reading data, DASD 232
   running commands 137
   running NetView, MVS, or local VTAM commands 135
   running VTAM DISPLAY, VARY, and MODIFY, remote domains 227
   subsequent stage 6
   trouble shooting 21, 299
   using, high level language 235
PIPEND stage command
description 152
   example, ending a pipeline with a return code 153
PPI
generating an alert from hex data (PPI), example 158
   receiving a response, example 159
   responding to requests, example 158
PPI stage command 154
   PRESATTR stage command 160
   changing color of selected data, example 162
   program temporary fix (PTF) 308
   PTF (program temporary fix)
   searching, PIPE command 308
   publications
      feedback vii
      online vii
      ordering vii
Q
   QSAM stage command 163
R
   restrictions (continued)
      held messages, autotask 108
      length restrictions, handling 115, 118
      messages deleted, displaying 43
      search strings 23
      terminating CORRWAIT 52
   REVERSE stage command 166
   ROUTE stage command 168
   rows, VET option 224, 273
SAFE
   default 170
   named 170
   SAFE stage command
      creating named SAFEs, NULL messages 172
      description 170
      determining, named SAFE, example 171
      passing messages, REXX command list 172
      passing messages, second PIPE command 172
      taking snapshots, pipeline messages 299
      using named SAFE, example 172
      search strings, restrictions 23
      selecting messages, position, TOSTRING 257
   SEPARATE stage command
      breaking multiline messages, example 175
      description 174
      selecting single-line messages, multiline messages, example 175
      separating data lines, example 175
      separating multiline messages 255
      separation character, changing 23
      setting timeout value, asynchronous messages 49
      shortcut keys, keyboard ix
      single-line messages 39
      snapshots, pipeline messages 299
   SORT stage command
      description 176
      sorting, example 177
      sorting messages 176
   SPLIT stage command
      description 179
      dividing text 179
      splitting, examples 180
SQL
   accessing DB2 285
   concurrent SQL stages 291
   creating, loading, and querying tables 288
   defining unit of work 290
   diagnosing output 292
   querying databases and formatting results 289
SQL stage command 181
   SQLCODES stage command 188
   SQSELECT - Format a query 287
   stage commands
      < (From disk) 232
   stage commands (continued)
      $STEM 189
      $VAR 214
      APPEND 27
      BETWEEN 29
      CASEI 31
      CC 45
      CHANGE 33
      CHOP 35
      COLLECT 37
      COLOR 160
      CONSOLE 42
      CORRCMD 45
      CORRWAIT 49
      COUNT 54
      CPDOMAIN 58
      DELDUPES 60
      DIVERT 63
      DROP 64
      DUPLICAT 66
      EDIT 68
      ENVDATA 98
      EXPOSE 100
      FANIN 102
      FANINANY 104
      FANOUT 106
      HELMSG 108
      HOLE 110
      INSTORE 112
      INTERRUPT 115
      IPLOG 118
      JOINCONT 119
      KEEP 122
      LITERAL 124
      LOCATE 125
      LOGTO 127
      LOOKUP 129
      MEMLIST 133
      MVS 135
      NETVIEW 137
      NLOCATE 141
      NLS 143
      NOT 145
      PERSIST 147
      PICK 149
      PIPEND 152
      PPI 154
      PRESATTR 160
      QSAM 163
      REVERSE 166
      ROUTE 168
      SAFE 170
      SEPARATE 174
      SORT 176
      SPLIT 179
      SQL 181
      SQLCODES 188
      STEM 189
      STRIP 193
      SUBSYM 196
      summary 24
      TAKE 197
      TOSTRING 199
      TSO 201
      TSROUTE 207
      UNIX 209
      VAR 214

318  Tivoli NetView for z/OS Customization: Using Pipes
stage commands (continued)
VARLOAD 217
VET 222
VTAM 227
Xirate 230
STAGESEP parameter of PIPE, changing separation characters 20
STEM stage command
- comparing and setting variables, example 219
- copying task globals 219
- counting comment lines in files, example 233
- description 189
- saving records processed counts, example 192
- setting variables, example 219
- taking snapshots, pipeline messages 299
- update current group members 220
- writing, named variables, example 216
- writing, stemmed variables, example 192
STRIP stage command 193
subsequent stage, pipelines 6
SUBSYM stage command 196
SYSLOG, LOGTO option 127
System Query Language (SQL) 285

T
TAKE stage command
- description 197
- selecting last message, example 199
- selecting messages, position 263
timeout
- inserting return codes 51
- specifying intervals 51
Tivoli Customer Support viii
TOSTRING stage command
description 199
- ending a wait, example 200
TISO stage command 201
- discover TSO stacks serving a user 205
TSROUTE stage command 207
- automation table sample 208

U
UNIX stage command
- compiling and executing a Java sample 212
description 209
- executing commands, DSIPARM 212
- listing current UNIX working directory 212

V
VAR stage command 214
VARLOAD stage command 217
VET stage command
- command 273
description 222