IBM Tivoli NetView for z/OS
Version 6 Release 2

Programming: Pipes

IBM
IBM Tivoli NetView for z/OS
Version 6 Release 2

Programming: Pipes

IBM
Before using this information and the product it supports, read the information in "Notices" on page 327.

This edition applies to version 6, release 2 of IBM Tivoli NetView for z/OS (product number 5697-NV6) and to all subsequent versions, releases, and modifications until otherwise indicated in new editions.

This edition replaces SC27-2859-03.

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

Figures ........................................................................... vii

About this publication ................................................... ix

Intended audience ........................................................... ix
Publications ................................................................... ix
  IBM Tivoli NetView for z/OS library .............................. ix
  Related publications .................................................... xi
  Accessing terminology online ...................................... xi
  Using NetView for z/OS online help .............................. xii
  Accessing publications online ..................................... xii
  Ordering publications ................................................ xii
Accessibility ................................................................. xiii
Service Management Connect ....................................... xiii
Tivoli technical training ................................................ xiii
Tivoli user groups ....................................................... xiii
Downloads ..................................................................... xiii
Support information .................................................... xiv
Conventions used in this publication ............................... xiv
  Typeface conventions ................................................ xiv
  Operating system-dependent variables and paths. ............ xv
  Syntax diagrams ...................................................... xv

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

What Is a Pipeline .......................................................... 1
Pipeline Stages .............................................................. 2
PIPE Command .......................................................... 3
Stage Input and Output .................................................. 4
  First and Subsequent Stages ........................................ 6
Complex Pipelines ...................................................... 7
  Creating a Complex Pipeline ...................................... 7
  Processing a Complex Pipeline ................................. 9
Stages .......................................................................... 11
  Device Drivers ........................................................ 11
  Filters .................................................................... 12
Understanding NetView Pipelines .................................. 12
  How a Pipeline Begins ............................................. 12
  How a Pipeline Ends ................................................ 12
Online Help Facility ..................................................... 13
Getting Started with NetView Pipelines ......................... 13

Chapter 2. Pipeline Stages and Syntax ............................................ 19

PIPE (NCCF) ................................................................ 19
  PIPE Stages ........................................................... 23
PIPE APPEND .......................................................... 26
PIPE BETWEEN ....................................................... 28
PIPE CASEI ................................................................ 30
PIPE CHANGE ........................................................ 31
PIPE CHOP ................................................................ 34
PIPE COLLECT ......................................................... 36
PIPE CONSOLE ........................................................ 41
PIPE COREVENT .................................................... 44
PIPE COREVTDA .................................................... 45
PIPE CORRCMD ....................................................... 46
PIPE CORRWAIT ..................................................... 49
PIPE COUNT ........................................................... 54

© Copyright IBM Corp. 1997, 2014
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIPE CPDOMAIN</td>
<td>57</td>
</tr>
<tr>
<td>PIPE CZR</td>
<td>59</td>
</tr>
<tr>
<td>PIPE DELDUPES</td>
<td>61</td>
</tr>
<tr>
<td>PIPE DIVERT.</td>
<td>63</td>
</tr>
<tr>
<td>PIPE DROP</td>
<td>64</td>
</tr>
<tr>
<td>PIPE DUPLICAT</td>
<td>66</td>
</tr>
<tr>
<td>PIPE EDIT</td>
<td>67</td>
</tr>
<tr>
<td>PIPE ENVDATA</td>
<td>114</td>
</tr>
<tr>
<td>PIPE EXPOSE</td>
<td>115</td>
</tr>
<tr>
<td>PIPE FANIN.</td>
<td>117</td>
</tr>
<tr>
<td>PIPE FANINANY</td>
<td>118</td>
</tr>
<tr>
<td>PIPE FANOUT.</td>
<td>120</td>
</tr>
<tr>
<td>PIPE FMTPACK</td>
<td>121</td>
</tr>
<tr>
<td>PIPE HELDMSG</td>
<td>126</td>
</tr>
<tr>
<td>PIPE HOLE</td>
<td>127</td>
</tr>
<tr>
<td>PIPE INSTORE.</td>
<td>129</td>
</tr>
<tr>
<td>PIPE INTERPRT.</td>
<td>131</td>
</tr>
<tr>
<td>PIPE IPLOG</td>
<td>134</td>
</tr>
<tr>
<td>PIPE JOINCONT</td>
<td>135</td>
</tr>
<tr>
<td>PIPE KEEP</td>
<td>137</td>
</tr>
<tr>
<td>PIPE LITERAL</td>
<td>140</td>
</tr>
<tr>
<td>PIPE LOCATE</td>
<td>141</td>
</tr>
<tr>
<td>PIPE LOGTO</td>
<td>143</td>
</tr>
<tr>
<td>PIPE LOOKUP.</td>
<td>144</td>
</tr>
<tr>
<td>PIPE MEMLIST.</td>
<td>148</td>
</tr>
<tr>
<td>PIPE MVS</td>
<td>150</td>
</tr>
<tr>
<td>PIPE NETVIEW</td>
<td>152</td>
</tr>
<tr>
<td>PIPE NLOCATE</td>
<td>156</td>
</tr>
<tr>
<td>PIPE NILS</td>
<td>157</td>
</tr>
<tr>
<td>PIPE NOT</td>
<td>159</td>
</tr>
<tr>
<td>PIPE NPDAEV    D</td>
<td>160</td>
</tr>
<tr>
<td>PIPE PERSIST</td>
<td>161</td>
</tr>
<tr>
<td>PIPE PICK</td>
<td>164</td>
</tr>
<tr>
<td>PIPE PIPEND</td>
<td>166</td>
</tr>
<tr>
<td>PIPE PPI</td>
<td>168</td>
</tr>
<tr>
<td>PIPE PRESATTR</td>
<td>173</td>
</tr>
<tr>
<td>PIPE QSAM</td>
<td>176</td>
</tr>
<tr>
<td>PIPE REVERSE.</td>
<td>180</td>
</tr>
<tr>
<td>PIPE REVISRPT</td>
<td>182</td>
</tr>
<tr>
<td>PIPE ROUTE</td>
<td>182</td>
</tr>
<tr>
<td>PIPE SAFE</td>
<td>185</td>
</tr>
<tr>
<td>PIPE SEPARATE</td>
<td>188</td>
</tr>
<tr>
<td>PIPE SORT</td>
<td>190</td>
</tr>
<tr>
<td>PIPE SPLIT</td>
<td>192</td>
</tr>
<tr>
<td>PIPE SQL.</td>
<td>195</td>
</tr>
<tr>
<td>Performing a Query with SQL SELECT</td>
<td>198</td>
</tr>
<tr>
<td>Performing a Query with DESCRIBE SELECT</td>
<td>198</td>
</tr>
<tr>
<td>Loading Tables with SQL INSERT</td>
<td>199</td>
</tr>
<tr>
<td>Release statement, Set Connection, Set Current Degree, Set Current Package Set, Set Current Rules, or Send Current SQLID</td>
<td>199</td>
</tr>
<tr>
<td>Using SQL LISTREGS</td>
<td>199</td>
</tr>
<tr>
<td>Using SQL EXECUTE</td>
<td>200</td>
</tr>
<tr>
<td>Using Multiple Concurrent SQL Stages</td>
<td>200</td>
</tr>
<tr>
<td>Other Considerations When Using SQL</td>
<td>201</td>
</tr>
<tr>
<td>PIPE SQLCODES</td>
<td>201</td>
</tr>
<tr>
<td>PIPE STEM and PIPE $STEM</td>
<td>202</td>
</tr>
<tr>
<td>PIPE STRIP</td>
<td>206</td>
</tr>
<tr>
<td>PIPE SUBSYM</td>
<td>208</td>
</tr>
<tr>
<td>PIPE TAKE</td>
<td>209</td>
</tr>
<tr>
<td>PIPE TOSTRING</td>
<td>210</td>
</tr>
<tr>
<td>PIPE TSO.</td>
<td>212</td>
</tr>
</tbody>
</table>
Chapter 3. NetView Pipelines Device Drivers .................................................. 249
   Interfacing with the Task: CONSOLE, HELDMSG, LITERAL, LOGTO .............. 249
      Displaying Messages: CONSOLE ..................................................... 249
      Copying Held Messages into the Pipeline: HELDMSG ........................ 252
      Inserting Text into the Pipeline: LITERAL ..................................... 253
      Copying Pipeline Contents to a Log: LOGTO .................................. 255
   Interfacing with Other Applications: NETVIEW, VTAM .......................... 256
      Running a NetView Command: NETVIEW ....................................... 256
      Running a VTAM Command: VTAM .............................................. 259
   Working with DASD Data: < (From Disk) ........................................... 261
      Reading from DASD: (<) ......................................................... 261
   Accessing Variables within Command Procedures: VAR, STEM, SAFE .......... 262
      Reading from or Writing to Named Variables: VAR ......................... 262
      Reading from or Writing to Variables in a Stemmed Array: STEM .......... 263
      Reading from or Writing to a Command Procedure Message: SAFE .......... 265
   Building Large PIPE Commands: INTERPR ...................................... 267
      Using the INTERPR Stage ....................................................... 267

Chapter 4. NetView Pipeline Filters .............................................................. 269
   Manipulating Messages: SEPARATE, COLLECT .................................... 269
      Breaking Up an MLWTO: SEPARATE ............................................. 269
      Building an MLWTO: COLLECT .................................................. 270
   Selecting Messages by Content: LOCATE, NLOCATE, TOSTRING .............. 271
      Keeping or Discarding Matching Messages: LOCATE, NLOCATE ............ 272
      Selecting Messages Up to and Including a Message That Matches a Specified Text String: TOSTRING ................................. 275
   Selecting Messages by Position: TAKE, DROP .................................... 277
      Keeping the First or Last n Messages: TAKE ................................ 277
      Discarding the First or Last n Messages: DROP .............................. 278
   Emptying the Pipeline: HOLE ....................................................... 279
      Determining Correlation: HOLE .................................................. 279

Chapter 5. Full-Screen Automation ............................................................... 281
   What Is Full-Screen Automation ....................................................... 281
      A Simple Example ................................................................. 281
   Virtual OSTs .............................................................................. 284
      Dependent and Independent VOSTs .............................................. 284
   Attaching and Detaching Virtual OSTs ............................................. 285
      Attaching VOSTs ................................................................. 285
      Detaching VOSTs ............................................................... 286
   Interacting with Virtual OSTs ......................................................... 286
      VET First Stage ................................................................. 287
      A VET Command or Subsequent Stage ......................................... 287
   Handling Returned Messages .......................................................... 290
      ROWS Format ........................................................................ 291
      FIELDS Format ..................................................................... 291
      Other Messages ..................................................................... 294
Partial Screens ................................................................. 295
Debugging Full-Screen Automation Programs ......................... 295
NPDA Automation Example .................................................. 296

Chapter 6. Using Tivoli NetView for z/OS SQL Stages for Access to DB2 .................................................. 299
Accessing and Maintaining Relational Databases (SQL Tables) .......................................................... 299
  The SQL Stage ................................................................. 299
  Defining DB2 to the Tivoli NetView for z/OS program ............. 299
SQSELECT - Format a Query .................................................. 301
Creating, Loading, and Querying a Table ................................ 302
  Querying a Database and Formatting the Results .................. 303
  Working with Graphic Character Strings (DBCS) .................... 304
  Defining the Unit of Work ............................................... 304
  Using Secondary Output Streams with SQL ......................... 305
  Using Concurrent SQL Stages ......................................... 305
  Using CONSOLE DUMP to Diagnose SQL Output .................. 306

Chapter 7. REXX Access to VSAM Files .................................. 307
DSIVSMX: Generic Access to Keyed VSAM Files from REXX or Command Lists ............................................. 307
Using DSIVSMX with Alternate Index VSAM Files ................. 307
  Defining a VSAM File with Alternate Index ....................... 307
  Accessing VSAM Files Using Alternate Keys ...................... 308
  Deleting an Alternate Index File ..................................... 308
  Using the AUTOTOKE Value Provided by DSIVSMX ............... 308
DSIVSAM: Access to Keyed VSAM Files Defined by NetView DSTs ............................................................... 308
  Using the AUTOTOKE Value Provided by DSIVSAM ............... 309

Chapter 8. Debugging NetView Pipelines. ................................ 311
Understanding Error Messages ............................................. 311
Online Help ........................................................................ 311
Clogged Pipelines ............................................................. 311
Tracing Pipelines .................................................................. 313
  Displaying Stage Results ................................................. 313
  Displaying Data Stream Information (DEBUG) ...................... 313
Displaying Return Codes ..................................................... 316
Additional Troubleshooting Tips ........................................... 316

Appendix. Additional NetView Pipeline Examples ..................... 319
Displaying Part of a Lengthy Automated Message .................... 319
Transferring a Large Variable to Another Task ....................... 320
Searching for APARs and PTFs ............................................ 322
Displaying Task Information Summary .................................. 322
Displaying or Canceling a JES2 Job ...................................... 324

Notices ............................................................................. 327
Programming Interfaces ....................................................... 329
Trademarks ........................................................................ 329
Privacy policy considerations .............................................. 329

Index .............................................................................. 331
Figures

1. Stages within a Pipeline .................................................. 2
2. Messages Flowing through a Stage ...................................... 2
3. Messages Flowing through a LOCATE Stage ......................... 3
4. A PIPE Command Coded in Portrait Format ......................... 4
5. A PIPE Command Coded in Landscape Format ....................... 4
6. Messages Flowing through Multiple Stages ......................... 4
7. Input and Output Data Streams ......................................... 5
8. Examples of First and Subsequent Stages ......................... 7
9. Complex Pipeline .......................................................... 7
10. Complex Pipeline Example Output .................................... 9
11. Map of a Pipeline with Two Device Drivers ....................... 11
12. A Pipeline Invoked from a Command Procedure Called WISHCLST 17
13. Pipeline Output from WISHCLST Command Procedure ............ 18
14. TSO Command Flow ....................................................... 213
15. Simple Full-Screen Automation Example: REPTALRM .............. 282
16. Browse Changeable Configuration Panel ............................. 282
17. Example of Screen Returned for VET NEXT ROWS .................. 283
18. REPTALRM Results ......................................................... 284
19. Example Screen for VET CURRENT ROWS .......................... 291
20. Example Screen for VET NEXT FIELDS ............................. 293
21. Message and Full-Screen Returned to VET ......................... 295
22. Sample Partial DUMP of VOST Data ................................. 296
23. Alert History Automation Results ..................................... 296
24. Sample Full-Screen Automation Program — Capture Alert History 297
25. Job BLDTAPE Example .................................................. 319
26. Modified Job BLDTAPE Example ................................. 320
27. Transfer Send Results Screen ......................................... 321
28. Transfer Received Results Screen ................................. 321
29. Searching for APARs or PTFs with a PIPE command .............. 322
30. DSIGDS Task Summary Screen ........................................ 324
31. JES2JOB Display Command Output Example ....................... 326
32. JES2JOB Cancel Command Output Example ....................... 327
About this publication

The IBM® Tivoli® NetView® for z/OS® product provides advanced capabilities that you can use to maintain the highest degree of availability of your complex, multi-platform, multi-vendor networks and systems from a single point of control. This publication, IBM Tivoli NetView for z/OS Programming: Pipes, describes how to use NetView pipelines to customize your NetView installation.

Intended audience

This publication is for system programmers who customize the NetView program using NetView pipelines.

Publications

This section lists publications in the IBM Tivoli NetView for z/OS library and related documents. It also describes how to access Tivoli publications online and how to order Tivoli publications.

IBM Tivoli NetView for z/OS library

The following documents are available in the IBM Tivoli NetView for z/OS library:

- Administration Reference, SC27-2869, describes the NetView program definition statements required for system administration.
- Application Programmer’s Guide, SC27-2870, describes the NetView program-to-program interface (PPI) and how to use the NetView application programming interfaces (APIs).
- Automation Guide, SC27-2846, describes how to use automated operations to improve system and network efficiency and operator productivity.
- Command Reference Volume 1 (A-N), SC27-2847, and Command Reference Volume 2 (O-Z), SC27-2848, describe the NetView commands, which can be used for network and system operation and in command lists and command procedures.
- Customization Guide, SC27-2849, describes how to customize the NetView product and points to sources of related information.
- Data Model Reference, SC27-2850, provides information about the Graphic Monitor Facility host subsystem (GMFHS), SNA topology manager, and MultiSystem Manager data models.
- Installation: Configuring Additional Components, GC27-2851, describes how to configure NetView functions beyond the base functions.
- Installation: Configuring Graphical Components, GC27-2852, describes how to install and configure the NetView graphics components.
- Installation: Configuring the GDPS Active/Active Continuous Availability Solution, SC14-7477, describes how to configure the NetView functions that are used with the GDPS Active/Active Continuous Availability solution.
- Installation: Configuring the NetView Enterprise Management Agent, GC27-2853, describes how to install and configure the NetView for z/OS Enterprise Management Agent.
- Installation: Getting Started, GI11-9443, describes how to install and configure the base NetView program.
Installation: Migration Guide, GC27-2854, describes the new functions that are provided by the current release of the NetView product and the migration of the base functions from a previous release.

IP Management, SC27-2855, describes how to use the NetView product to manage IP networks.

Messages and Codes Volume 1 (AAU-DSI), GC27-2856, and Messages and Codes Volume 2 (DUI-IHS), GC27-2857, describe the messages for the NetView product, the NetView abend codes, the sense codes that are included in NetView messages, and generic alert code points.

Programming: Assembler, SC27-2858, describes how to write exit routines, command processors, and subtasks for the NetView product using assembler language.

Programming: Pipes, SC27-2859, describes how to use the NetView pipelines to customize a NetView installation.

Programming: PL/I and C, SC27-2860, describes how to write command processors and installation exit routines for the NetView product using PL/I or C.

Programming: REXX and the NetView Command List Language, SC27-2861, describes how to write command lists for the NetView product using the Restructured Extended Executor language (REXX) or the NetView command list language.

Resource Object Data Manager and GMFHS Programmer’s Guide, SC27-2862, describes the NetView Resource Object Data Manager (RODM), including how to define your non-SNA network to RODM and use RODM for network automation and for application programming.

Security Reference, SC27-2863, describes how to implement authorization checking for the NetView environment.

SNA Topology Manager Implementation Guide, SC27-2864, describes planning for and implementing the NetView SNA topology manager, which can be used to manage subarea, Advanced Peer-to-Peer Networking, and TN3270 resources.

Troubleshooting Guide, GC27-2865, provides information about documenting, diagnosing, and solving problems that occur in the NetView product.


User’s Guide: Automated Operations Network, SC27-2866, describes how to use the NetView Automated Operations Network (AON) component, which provides event-driven network automation, to improve system and network efficiency. It also describes how to tailor and extend the automated operations capabilities of the AON component.

User’s Guide: NetView, SC27-2867, describes how to use the NetView product to manage complex, multivendor networks and systems from a single point.


Licensed Program Specifications, GC31-8848, provides the license information for the NetView product.

Program Directory for IBM Tivoli NetView for z/OS US English, GI11-9444, contains information about the material and procedures that are associated with installing the IBM Tivoli NetView for z/OS product.

Program Directory for IBM Tivoli NetView for z/OS Japanese, GI11-9445, contains information about the material and procedures that are associated with installing the IBM Tivoli NetView for z/OS product.
• Program Directory for IBM Tivoli NetView for z/OS Enterprise Management Agent, GI11-9446, contains information about the material and procedures that are associated with installing the IBM Tivoli NetView for z/OS Enterprise Management Agent.

• IBM Tivoli NetView for z/OS V6R2 Online Library, LCD7-4913, contains the publications that are in the NetView for z/OS library. The publications are available in PDF and HTML formats.

Related publications
You can find additional product information on the NetView for z/OS web site at http://www.ibm.com/software/tivoli/products/netview-zos/

For information about the NetView Bridge function, see Tivoli NetView for OS/390 Bridge Implementation, SC31-8238-03 (available only in the V1R4 library).

Accessing terminology online
The IBM Terminology web site consolidates the terminology from IBM product libraries in one convenient location. You can access the Terminology web site at http://www.ibm.com/software/globalization/terminology/

For NetView for z/OS terms and definitions, see the IBM Terminology web site.

The following terms are used in this library:

NetView
For the following products:
• Tivoli NetView for z/OS version 6 release 2
• Tivoli NetView for z/OS version 6 release 1
• Tivoli NetView for z/OS version 5 release 4
• Tivoli NetView for z/OS version 5 release 3
• Tivoli NetView for OS/390® version 1 release 4
• NetView releases that are no longer supported

CNMCMDC
For the CNMCMDC member and the members that are included in it using the %INCLUDE statement

CNMSTYLE
For the CNMSTYLE member and the members that are included in it using the %INCLUDE statement

DSIOPF
For the DSIOPF member and the members that are included in it using the %INCLUDE statement

PARMLIB
For SYS1.PARMLIB and other data sets in the concatenation sequence

MVS™ For z/OS operating systems

MVS element
For the base control program (BCP) element of the z/OS operating system

VTAM®
For Communications Server - SNA Services

IBM Tivoli Network Manager
For either of these products:
• IBM Tivoli Network Manager
• IBM Tivoli OMNIbus and Network Manager
IBM Tivoli Netcool/OMNIbus
For either of these products:
• IBM Tivoli Netcool/OMNIbus
• IBM Tivoli OMNIbus and Network Manager

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

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

Using NetView for z/OS online help
The following types of NetView for z/OS mainframe online help are available, depending on your installation and configuration:
• General help and component information
• Command help
• Message help
• Sense code information
• Recommended actions

Accessing publications online
The documentation DVD, IBM Tivoli NetView for z/OS V6R2 Online Library contains the publications that are in the product library. The publications are available in PDF and HTML formats. Refer to the readme file on the DVD for instructions on how to access the documentation.

IBM posts publications for this and all other Tivoli products, as they become available and whenever they are updated, to the Tivoli Documentation Central website at https://www.ibm.com/developerworks/mydeveloperworks/wikis/home/wiki/Tivoli%20Documentation%20Central

Note: If you print PDF documents on other than letter-sized paper, set the option in the File > Print window that enables Adobe Reader to print letter-sized pages on your local paper.

Ordering publications
You can order many Tivoli publications online at http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss

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, contact your software account representative to order Tivoli publications. To locate the telephone number of your local representative, perform the following steps:
2. Select your country from the list and click Go.
3. Click About this site to see an information page that includes the telephone number of your local representative.
Accessibility

Accessibility features help users with a physical disability, such as restricted mobility or limited vision, to use software products successfully. 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.

For additional information, see the Accessibility appendix in the User’s Guide: NetView.

Service Management Connect

Connect, learn, and share with Service Management professionals: product support technical experts who provide their perspectives and expertise.


- Become involved with transparent development, an ongoing, open engagement between other users and IBM developers of Tivoli products. You can access early designs, sprint demonstrations, product roadmaps, and prerelease code.
- Connect one-on-one with the experts to collaborate and network about Tivoli and the NetView community.
- Read blogs to benefit from the expertise and experience of others.
- Use wikis and forums to collaborate with the broader user community.

Tivoli technical training

For Tivoli technical training information, refer to the following IBM Tivoli Education website at [http://www.ibm.com/software/tivoli/education](http://www.ibm.com/software/tivoli/education)

Tivoli user groups

Tivoli user groups are independent, user-run membership organizations that provide Tivoli users with information to assist them in the implementation of Tivoli Software solutions. Through these groups, members can share information and learn from the knowledge and experience of other Tivoli users.

Access the Tivoli Users Group at [http://www.tivoli-ug.org](http://www.tivoli-ug.org)

Downloads

Clients and agents, NetView product demonstrations, and several free NetView applications can be downloaded from the NetView for z/OS support web site:


In the “Support shortcuts” pane, expand Tivoli NetView for z/OS, and click Fixes (downloads) to go to a page where you can search for or select downloads.

These applications can help with the following tasks:
• Migrating customization parameters and initialization statements from earlier releases to the CNMSTUSR member and command definitions from earlier releases to the CNMCMDU member.
• Getting statistics for your automation table and merging the statistics with a listing of the automation table
• Displaying the status of a job entry subsystem (JES) job or canceling 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 Time Sharing Option (TSO) commands and receiving responses

Support information

If you have a problem with your IBM software, you want to resolve it quickly. IBM provides the following ways for you to obtain the support you need:

**Online**


**IBM Support Assistant**

The IBM Support Assistant is a free local software serviceability workbench that helps you resolve questions and problems with IBM software products. The Support Assistant provides quick access to support-related information and serviceability tools for problem determination. To install the Support Assistant software, go to [http://www.ibm.com/software/support/isa/](http://www.ibm.com/software/support/isa/).

**Troubleshooting information**

For more information about resolving problems with the NetView for z/OS product, see the *IBM Tivoli NetView for z/OS Troubleshooting Guide*. Additional support for the NetView for z/OS product is available through the NetView user group on Yahoo at [http://groups.yahoo.com/group/NetView/](http://groups.yahoo.com/group/NetView/). This support is for NetView for z/OS customers only, and registration is required. This forum is monitored by NetView developers who answer questions and provide guidance. When a problem with the code is found, you are asked to open an official problem management record (PMR) to obtain resolution.

Conventions used in this publication

This section describes the conventions that are used in this publication.

**Typeface conventions**

This publication uses the following typeface conventions:

**Bold**

• Lowercase commands and mixed case commands that are otherwise difficult to distinguish from surrounding text
• Interface controls (check boxes, push buttons, radio buttons, spin buttons, fields, folders, icons, list boxes, items inside list boxes, multicolumn lists, containers, menu choices, menu names, tabs, property sheets), labels (such as **Tip:**; and **Operating system considerations:**)
- Keywords and parameters in text

**Italic**
- Citations (examples: titles of publications, diskettes, and CDs
- Words defined in text (example: a nonswitched line is called a *point-to-point line*)
- Emphasis of words and letters (words as words example: “Use the word *that* to introduce a restrictive clause.”; letters as letters example: “The LUN address must start with the letter *L*.”)
- New terms in text (except in a definition list): a *view* is a frame in a workspace that contains data.
- Variables and values you must provide: ... where *myname* represents...

**Monospace**
- Examples and code examples
- File names, programming keywords, and other elements that are difficult to distinguish from surrounding text
- Message text and prompts addressed to the user
- Text that the user must type
- Values for arguments or command options

### Operating system-dependent variables and paths

For workstation components, this publication uses the UNIX convention for specifying environment variables and for directory notation.

When using the Windows command line, replace `$variable` with `%variable%` for environment variables and replace each forward slash (`/`) with a backslash (`\`) in directory paths. The names of environment variables are not always the same in the Windows and UNIX environments. For example, `%TEMP%` in Windows environments is equivalent to `$TMPDIR` in UNIX environments.

**Note:** If you are using the bash shell on a Windows system, you can use the UNIX conventions.

### Syntax diagrams

The following syntax elements are shown in syntax diagrams. Read syntax diagrams from left-to-right, top-to-bottom, following the horizontal line (the main path).

- “Symbols”
- “Parameters” on page xvi
- “Punctuation and parentheses” on page xvi
- “Abbreviations” on page xvii

For examples of syntax, see “Syntax examples” on page xvii.

#### Symbols

The following symbols are used in syntax diagrams:

- `►►` Marks the beginning of the command syntax.
- `►` Indicates that the command syntax is continued.
- `|` Marks the beginning and end of a fragment or part of the command syntax.
- `►◄` Marks the end of the command syntax.
Parameters
The following types of parameters are used in syntax diagrams:

Required
Required parameters are shown on the main path.

Optional
Optional parameters are shown below the main path.

Default
Default parameters are shown above the main path. In parameter descriptions, default parameters are underlined.

Syntax diagrams do not rely on highlighting, brackets, or braces. In syntax diagrams, the position of the elements relative to the main syntax line indicates whether an element is required, optional, or the default value.

When you issue a command, spaces are required between the parameters unless a different separator, such as a comma, is specified in the syntax.

Parameters are classified as keywords or variables. Keywords are shown in uppercase letters. Variables, which represent names or values that you supply, are shown in lowercase letters and are either italicized or, in NetView help, displayed in a differentiating color.

In the following example, the USER command is a keyword, the user_id parameter is a required variable, and the password parameter is an optional variable.

►►USER—user_id—password—◄◄

Punctuation and parentheses
You must include all punctuation that is shown in the syntax diagram, such as colons, semicolons, commas, minus signs, and both single and double quotation marks.

When an operand can have more than one value, the values are typically enclosed in parentheses and separated by commas. For a single value, the parentheses typically can be omitted. For more information, see “Multiple operands or values” on page xviii.

If a command requires positional commas to separate keywords and variables, the commas are shown before the keywords or variables.

When examples of commands are shown, commas are also used to indicate the absence of a positional operand. For example, the second comma indicates that an optional operand is not being used:

COMMAND_NAME opt_variable_1,,opt_variable_3

You do not need to specify the trailing positional commas. Trailing positional and non-positional commas either are ignored or cause a command to be rejected. Restrictions for each command state whether trailing commas cause the command to be rejected.
Abbreviations
Command and keyword abbreviations are listed in synonym tables after each command description.

Syntax examples
The following examples show the different uses of syntax elements:

- “Required syntax elements”
- “Optional syntax elements”
- “Default keywords and values”
- “Multiple operands or values” on page xviii
- “Syntax that is longer than one line” on page xviii
- “Syntax fragments” on page xviii

Required syntax elements:
Required keywords and variables are shown on the main syntax line. You must code required keywords and variables.

►►REQUIRED_KEYWORD—required_variable◄◄

A required choice (two or more items) is shown in a vertical stack on the main path. The items are shown in alphanumeric order.

►►REQUIRED_OPERAND_OR_VALUE_1
  REQUIRED_OPERAND_OR_VALUE_2◄◄

Optional syntax elements:
Optional keywords and variables are shown below the main syntax line. You can choose not to code optional keywords and variables.

►►OPTIONAL_OPERAND◄◄

A required choice (two or more items) is shown in a vertical stack below the main path. The items are shown in alphanumeric order.

►►OPTIONAL_OPERAND_OR_VALUE_1
  OPTIONAL_OPERAND_OR_VALUE_2◄◄

Default keywords and values:
Default keywords and values are shown above the main syntax line in one of the following ways:

- A default keyword is shown only above the main syntax line. You can specify this keyword or allow it to default. The following syntax example shows the default keyword KEYWORD1 above the main syntax line and the rest of the optional keywords below the main syntax line.

- If an operand has a default value, the operand is shown both above and below the main syntax line. A value below the main syntax line indicates that if you specify the operand, you must also specify either the default value or another value shown. If you do not specify the operand, the default value above the main syntax line is used. The following syntax example shows the default values for operand OPTION=* above and below the main syntax line.
Multiple operands or values:
An arrow returning to the left above a group of operands or values indicates that more than one can be selected or that a single one can be repeated.

Syntax that is longer than one line:
If a diagram is longer than one line, each line that is to be continued ends with a single arrowhead and the following line begins with a single arrowhead.

Syntax fragments:
Some syntax diagrams contain syntax fragments, which are used for lengthy, complex, or repeated sections of syntax. Syntax fragments follow the main diagram. Each syntax fragment name is mixed case and is shown in the main diagram and in the heading of the fragment. The following syntax example shows a syntax diagram with two fragments that are identified as Fragment1 and Fragment2.
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 might 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 name as described in Chapter 2, “Pipeline Stages and Syntax,” on page 19. A stage name 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 might help you to understand pipelines if you think of them as a plumbing pipeline. In Table 1 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.
Imagine a stage as a small black box inserted into the plumbing pipeline described in Table 1 on page 1. 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 which performs some action on the data. In Figure 1 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.

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

Figure 2 shows an example of a stage processing messages. Unprocessed messages enter from the left, the stage reads and processes them, and the output appears on the right. This is analogous to the operation of one black box in our earlier plumbing metaphor. Just as in the earlier metaphor, you can string together several stages, each one driven by the output of a preceding stage and each one performing some unique operation on your data.

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 that is being used. Many stages generate one output message for each input message; some commands do not. In this example, three messages
went in, but only two came out. Without knowing exactly what stage 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 3 shows a more explicit example. In this case, the stage specification is:

```
LOCATE /Bob/
```

LOCATE is a stage 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.

![Figure 3. Messages Flowing through a LOCATE Stage](image)

**PIPE Command**

You can issue the NetView PIPE command anywhere you use a 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 4 on page 4. In landscape format, parameters are strung horizontally as shown in Figure 5 on page 4. When entering a PIPE command from the command line, you might prefer landscape form. When issuing a PIPE command from a command procedure, you might 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.

For more information, see Chapter 2, “Pipeline Stages and Syntax,” on page 19.

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 3 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 6 shows how messages flow through several stages. The output of the LOCATE stage becomes the input to the TAKE stage.

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 can 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 7 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 7. Input and Output Data Streams
```

**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.
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 can 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 8 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.
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 3 on page 3 and Figure 6 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 9.

The pipeline shown in Figure 9 is called a complex pipeline. A complex pipeline is made up of simple pipelines connected with labels, such as the one shown in Figure 6 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.
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 example is a label:

\[
\ldots | B: \text{LOCATE} /X/ | \ldots
\]

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:

\[
\text{PIPE} \ (\text{END} \ %) \\
\text{< NAMES} \\
\quad | A: \text{LOCATE} /BOB/ \\
\quad | \text{CHANGE} //\text{HERE IS A NAME CONTAINING 'BOB' ==>}\ \\
\quad | \text{CONSOLE} \\
\quad | %A: \\
\quad | \text{CONSOLE}
\]

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:. The end of the simple pipeline and the beginning of the second is indicated by the end character, which is defined as a % symbol in (END %).

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.
  
  \[
  \text{< NAMES} \\
  \quad | A: \text{LOCATE} /BOB/ \\
  \quad | \text{CHANGE} //\text{HERE IS A NAME CONTAINING 'BOB' ==>}\ \\
  \quad | \text{CONSOLE} \\
  \]

- 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 10.

![Figure 10. Complex Pipeline Example Output](image)

Note:
1. You can have as many simple pipelines within a complex pipeline specification as you need. Each stage with multiple outputs can pass data to different connectors. Or, multiple stages can pass data to a single connector.
2. Each pipeline stream acts as independently as possible. For example, in Figure 10 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 */
' | ~ C:', /* Pick up excess names from TAKE */
' | COLOR RED', /* Names past twelfth should be red */
' | R:', /* give these to FANIN's secondary */
```
The first 12 names are displayed on the console in the default color. The remaining names are displayed in red.

**Stages**

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

Device drivers are stages 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 11.

![Figure 11. Map of a Pipeline with Two Device Drivers](image)

This PIPE command performs the functions shown in Figure 11.

```
PIPE < TESTDATA | CONSOLE
```

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 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 stages 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,” on page 19 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 stages 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 name, enter:

HELP PIPE stage_name

Where: stage_name is any NetView PIPE stage.

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 might 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,” on page 269. For information on device drivers, see Chapter 3, “NetView Pipelines Device Drivers,” on page 249.

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 ddbname. 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:

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NET VIEW</th>
<th>03/26/10 13:10:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN19 PIPE &lt; WISHLIST CONSOLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 CAIRO, EGYPT AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 CASABLANCA, MOROCCO AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 KRUGER PARK AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 NILE RIVER AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 BANGKOK, THAILAND ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 GREAT WALL, CHINA ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 TOKYO, JAPAN ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 YANGTZE RIVER ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 CASABLANCA, MOROCCO AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 CAIRO, EGYPT AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 KRUGER PARK AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 NILE RIVER AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 BANGKOK, THAILAND ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 GREAT WALL, CHINA ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 TOKYO, JAPAN ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 YANGTZE RIVER ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 CASABLANCA, MOROCCO AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 CAIRO, EGYPT AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 KRUGER PARK AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 NILE RIVER AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 BANGKOK, THAILAND ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 GREAT WALL, CHINA ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 TOKYO, JAPAN ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 YANGTZE RIVER ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 CASABLANCA, MOROCCO AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 CAIRO, EGYPT AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 KRUGER PARK AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 NILE RIVER AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 BANGKOK, THAILAND ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 GREAT WALL, CHINA ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 TOKYO, JAPAN ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 YANGTZE RIVER ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 CASABLANCA, MOROCCO AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 CAIRO, EGYPT AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 KRUGER PARK AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 NILE RIVER AFRICA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 BANGKOK, THAILAND ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 GREAT WALL, CHINA ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 TOKYO, JAPAN ASIA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN19 YANGTZE RIVER ASIA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 had entered 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 might 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 might 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 is selected from the input stream and all other messages are 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         PETE 03/26/10 13:50:00
* CMN19     PIPE < WISHLIST LOCATE /RIVER/ /PARK/ NLOCATE /A
   /AUSTRALIA/ TAKE FIRST 1 LITERAL /OUR FINAL VACATION CHOICE/
   COLLECT CONSOLE
   CMN19
   OUR FINAL VACATION CHOICE
   RHINE RIVER EUROPE
```

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 12 on page 17. The output from the procedure in Figure 12 on page 17 is shown in Figure 13 on page 18.
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
*********************************************************************
** BOTTOM OF DATA **************

Figure 12. A Pipeline Invoked from a Command Procedure Called WISHCLST
WISHCMD is processing
/*REXX WISHCMD */
/* This procedure uses three PIPE commands
- The first writes a message to a cleared screen
- The second writes a message to a REXX variable.
- The third read the variable and display the result.
------------------------------------------------------------------*/
ADDRESS NETVASIS /* Prevent literals from being upper cased */

'PIPE (NAME WISHCMD1)', /* Simple message to operator */
' LITERAL /WISHCMD 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 (NAME WISHread)',
' < 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 (NAME WISHOUT)',
' VAR VACVAR',
' LITERAL /Vacation Choice/','
' COLLECT',
' CONSOLE'
EXIT 0
/*---------------------------------------------------------- End of WISHCMD */

Figure 13. Pipeline Output from WISHCLST Command Procedure. Created by the pipeline in Figure 12 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.

**PIPE (NCCF)**

**Syntax**

```plaintext
PIPEREF

PIPE (Pipe Options)

(name: (DEBUG))

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.
Pipeline Stages and Syntax

You determine the function of each stage by coding a stage 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 parameters, controls other run characteristics that are 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 stages 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, enter:

```
HELP PIPE stage_name
```

Where:

- `stage_name` is any NetView PIPE stage.

See "PIPE Stages" on page 23 for an alphabetic listing and brief summary of each PIPE stage.

**Operands 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 items 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 can 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 311.

**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 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 examples:

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

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

The following text is displayed on the console:

```
MY END CHARACTER IS ?
```

**ESC**

Indicates that the character following the specified character is treated literally 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
```

Results in the display of the following text:

```
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 7.

**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 pre-empt pipeline processing.

LOWQENAB affects 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.
Pipeline Stages and Syntax

**pipename**

The value is 1 - 8 alphanumeric characters. The default value is PIPE.

**stage_specification**

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

- If running DBCS, be careful of what STAGESEP character is used. The NetView program determines the hexadecimal equivalent of the STAGESEP character and then scans the string until it finds it again. The scan does not determine if the string is part of a DBCS character. The next occurrence of the string ends the stage command. See “PIPE VAR and PIPE $VAR” on page 222 for more information.

- See the individual stages for other usage notes 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 separator character was used improperly.</td>
</tr>
<tr>
<td>16</td>
<td>A stage specification error occurred.</td>
</tr>
</tbody>
</table>
A storage failure occurred during PIPE interpretation.

The pipeline is clogged. A deadlock occurred during pipeline processing.

For information on debugging clogged pipeline conditions see Chapter 8, “Debugging NetView Pipelines,” on page 31.

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

A RESET condition occurred.

Example: Changing the Separation Character with STAGESEP

To change the stage 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 separation character from the default value of a vertical bar (|) to a period (.), use double quotation marks (") 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 separator.

PIPE Stages

Table 2 contains an alphabetic summary of the stages in the PIPE command and shows the minimum synonym allowed for each stage in a pipeline specification.

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

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Task Performed</th>
<th>Synonym</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND</td>
<td>Defines a pipeline that runs after other stages are complete.</td>
<td>APPEND</td>
<td>“PIPE APPEND” on page 26</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>Divides message streams into sections.</td>
<td>BETWEEN</td>
<td>“PIPE BETWEEN” on page 28</td>
</tr>
<tr>
<td>CASEI</td>
<td>Compares character strings without respect to case.</td>
<td>CAS</td>
<td>“PIPE CASEI” on page 30</td>
</tr>
<tr>
<td>CHANGE</td>
<td>Replaces occurrences of one string with another.</td>
<td>CHAN</td>
<td>“PIPE CHANGE” on page 31</td>
</tr>
<tr>
<td>CHOP</td>
<td>Truncates lines after a specified character, column, or string.</td>
<td>CHOP</td>
<td>“PIPE CHOP” on page 34</td>
</tr>
<tr>
<td>COLLECT</td>
<td>Creates a multiline message, or messages, from input lines.</td>
<td>COL</td>
<td>“PIPE COLLECT” on page 36</td>
</tr>
<tr>
<td>CONSOLE</td>
<td>Displays the contents of the pipeline.</td>
<td>CON</td>
<td>“PIPE CONSOLE” on page 41</td>
</tr>
</tbody>
</table>
Table 2. Stages of the PIPE Command (continued)

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Task Performed</th>
<th>Synonym</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>COREVENT</td>
<td>Extracts information from messages for state correlation.</td>
<td>COREVENT</td>
<td>“PIPE COREVENT” on page 44</td>
</tr>
<tr>
<td>COREVTDA</td>
<td>Takes messages with event data, creates new MWLTO messages from the input stage, and places them on the secondary output stream.</td>
<td>COREVTDA</td>
<td>“PIPE COREVTDA” on page 45</td>
</tr>
<tr>
<td>CORRCMD</td>
<td>Runs commands and adds timer and termination stages.</td>
<td>CC</td>
<td>“PIPE CORRCMD” on page 46</td>
</tr>
<tr>
<td>CORRWAIT</td>
<td>Allows asynchronous messages into the pipeline.</td>
<td>CORR, WAIT</td>
<td>“PIPE CORRWAIT” on page 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>“PIPE COUNT” on page 54</td>
</tr>
<tr>
<td>CPDOMAIN</td>
<td>Converts control point (CP) names to NetView domain names.</td>
<td>CPD</td>
<td>“PIPE CPDOMAIN” on page 57</td>
</tr>
<tr>
<td>DELDUPES</td>
<td>Deletes duplicate messages.</td>
<td>DELDUP</td>
<td>“PIPE DELDUPES” on page 61</td>
</tr>
<tr>
<td>DIVERT</td>
<td>Routes primary input to primary or secondary output.</td>
<td>DIVERT</td>
<td>“PIPE DIVERT” on page 63</td>
</tr>
<tr>
<td>DROP</td>
<td>Specifies the number of messages to be discarded from the pipeline.</td>
<td>DROP</td>
<td>“PIPE DROP” on page 64</td>
</tr>
<tr>
<td>DUPLICAT</td>
<td>Copies messages in the input stream.</td>
<td>DUP</td>
<td>“PIPE DUPLICAT” on page 66</td>
</tr>
<tr>
<td>EDIT</td>
<td>Creates or reformats messages.</td>
<td>EDIT</td>
<td>“PIPE EDIT” on page 67</td>
</tr>
<tr>
<td>ENVDATA</td>
<td>Outputs environment data.</td>
<td>ENV</td>
<td>“PIPE ENVDATA” on page 114</td>
</tr>
<tr>
<td>EXPOSE</td>
<td>Causes messages to be exposed for automation and logging.</td>
<td>EXPOSE</td>
<td>“PIPE EXPOSE” on page 115</td>
</tr>
<tr>
<td>FANIN</td>
<td>Merges multiple input streams, in stream order, into a single output stream.</td>
<td>FANIN</td>
<td>“PIPE FANIN” on page 117</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>“PIPE FANINANY” on page 118</td>
</tr>
<tr>
<td>FANOUT</td>
<td>Passes a single input stream to multiple output streams.</td>
<td>FANOUT</td>
<td>“PIPE FANOUT” on page 120</td>
</tr>
<tr>
<td>FMTPACKT</td>
<td>Takes raw TCPIP packet data, converts it into readable form, and generates reports that are passed to the primary output stream.</td>
<td>FMT</td>
<td>“PIPE FMTPACKT” on page 121</td>
</tr>
<tr>
<td>HELDMSG</td>
<td>Reads a copy of the held messages queue into the pipeline.</td>
<td>HELD</td>
<td>“PIPE HELDMSG” on page 126</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>“PIPE HOLE” on page 127</td>
</tr>
<tr>
<td>INSTORE</td>
<td>Adds, deletes, or replaces in-storage members.</td>
<td>INSTORE</td>
<td>“PIPE INSTORE” on page 129</td>
</tr>
<tr>
<td>INTERPRRT</td>
<td>Builds and runs stages from input command data. Facilitates long pipe commands.</td>
<td>INT</td>
<td>“PIPE INTERPRT” on page 131</td>
</tr>
<tr>
<td>JOINCONT</td>
<td>Joins consecutive messages that match a specified string.</td>
<td>JOINCONT</td>
<td>“PIPE JOINCONT” on page 135</td>
</tr>
<tr>
<td>Stage Name</td>
<td>Task Performed</td>
<td>Synonym</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>KEEP</td>
<td>Defines a task global place to store messages.</td>
<td>KEEP</td>
<td>“PIPE KEEP” on page 137</td>
</tr>
<tr>
<td>LITERAL</td>
<td>Inserts text into the pipeline.</td>
<td>LIT</td>
<td>“PIPE LITERAL” on page 140</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>“PIPE LOCATE” on page 141</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>“PIPE LOGTO” on page 143</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>Matches data within a pipeline.</td>
<td>LOOKUP</td>
<td>“PIPE LOOKUP” on page 144</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>“PIPE MEMLIST” on page 148</td>
</tr>
<tr>
<td>MVS</td>
<td>Runs specified MVS commands.</td>
<td>MVS</td>
<td>“PIPE MVS” on page 150</td>
</tr>
<tr>
<td>NETVIEW</td>
<td>Runs specified NetView commands.</td>
<td>NETV</td>
<td>“PIPE NETVIEW” on page 152</td>
</tr>
<tr>
<td>NLOCATE</td>
<td>Discards messages that match a specified character string.</td>
<td>NLOC</td>
<td>“PIPE NLOCATE” on page 156</td>
</tr>
<tr>
<td>NLS</td>
<td>Converts input messages to their translated versions.</td>
<td>NLS</td>
<td>“PIPE NLS” on page 157</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>“PIPE NOT” on page 159</td>
</tr>
<tr>
<td>NPDAEVD</td>
<td>Outputs NPDA event-detailed text messages and recommended actions.</td>
<td>NPDAEVD</td>
<td>“PIPE NPDAEVD” on page 160</td>
</tr>
<tr>
<td>PERSIST</td>
<td>Specifies the disposition for correlated output after a pipeline ends.</td>
<td>PERSIST</td>
<td>“PIPE PERSIST” on page 161</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>“PIPE PICK” on page 164</td>
</tr>
<tr>
<td>PIPEND</td>
<td>Causes a pipeline to end and return a return code.</td>
<td>PIPEEND</td>
<td>“PIPE PIPEND” on page 166</td>
</tr>
<tr>
<td>PPI</td>
<td>Passes data to a PPI receiver.</td>
<td>PPI</td>
<td>“PIPE PPI” on page 168</td>
</tr>
<tr>
<td>PRESATTR</td>
<td>Changes the way messages are displayed on the NetView console.</td>
<td>COLOR, COLOUR</td>
<td>“PIPE PRESATTR” on page 173</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>“PIPE QSAM” on page 176</td>
</tr>
<tr>
<td>REVERSE</td>
<td>Changes the order of message text and message lines.</td>
<td>REV</td>
<td>“PIPE REVERSE” on page 180</td>
</tr>
<tr>
<td>REVISRPT</td>
<td>Creates a report of the action of the active message revision table.</td>
<td>REVISRPT</td>
<td>“PIPE REVISRPT” on page 182</td>
</tr>
<tr>
<td>ROUTE</td>
<td>Sends messages to another task.</td>
<td>ROUTE</td>
<td>“PIPE ROUTE” on page 182</td>
</tr>
<tr>
<td>SAFE</td>
<td>Reads or writes messages to a command procedure message queue.</td>
<td>SAFE</td>
<td>“PIPE SAFE” on page 185</td>
</tr>
<tr>
<td>SEPARATE</td>
<td>Breaks MLWTOs into multiple single-line messages.</td>
<td>SEP</td>
<td>“PIPE SEPARATE” on page 188</td>
</tr>
<tr>
<td>SORT</td>
<td>Sorts input stream messages.</td>
<td>SORT</td>
<td>“PIPE SORT” on page 190</td>
</tr>
<tr>
<td>SPLIT</td>
<td>Divides a line of text into multiple lines.</td>
<td>SPLIT</td>
<td>“PIPE SPLIT” on page 190</td>
</tr>
<tr>
<td>SQL</td>
<td>Queries DB2® tables, inserts rows into DB2 tables, and issues DB2 commands.</td>
<td>SQL</td>
<td>“PIPE SQL” on page 195</td>
</tr>
<tr>
<td>Stage Name</td>
<td>Task Performed</td>
<td>Synonym</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>SQLCODES</td>
<td>Used for diagnostics when using the SQL stage</td>
<td>SQLCODES</td>
<td>“PIPE SQLCODES” on page 201</td>
</tr>
<tr>
<td>STEM</td>
<td>Reads or writes records to or from command procedure variables.</td>
<td>STEM</td>
<td>“PIPE STEM and PIPE $STEM” on page 202</td>
</tr>
<tr>
<td>STRIP</td>
<td>Removes characters from the beginning or end of a message.</td>
<td>STRIP</td>
<td>“PIPE STRIP” on page 206</td>
</tr>
<tr>
<td>SUBSYM</td>
<td>Substitutes MVS and user-defined system symbols in messages in the pipeline.</td>
<td>SUBS</td>
<td>“PIPE SUBSYM” on page 208</td>
</tr>
<tr>
<td>TAKE</td>
<td>Specifies the number of messages to be kept in the pipeline.</td>
<td>TAKE</td>
<td>“PIPE TAKE” on page 209</td>
</tr>
<tr>
<td>TOSTRING</td>
<td>Ends the data stream when a specific character string is located.</td>
<td>TOS</td>
<td>“PIPE TOSTRING” on page 210</td>
</tr>
<tr>
<td>TSO</td>
<td>Runs specified TSO commands.</td>
<td>TSO</td>
<td>“PIPE TSO” on page 212</td>
</tr>
<tr>
<td>TSRROUTE</td>
<td>Passes data to Topology Display Servers.</td>
<td>TSR</td>
<td>“PIPE TSROUTE” on page 217</td>
</tr>
<tr>
<td>UNIX</td>
<td>Runs specified UNIX commands.</td>
<td>UNIX</td>
<td>“PIPE UNIX” on page 218</td>
</tr>
<tr>
<td>VAR</td>
<td>Reads or writes records to or from command procedure variables.</td>
<td>VAR</td>
<td>“PIPE VAR and PIPE $VAR” on page 222</td>
</tr>
<tr>
<td>VARLOAD</td>
<td>Sets variables to a specified value.</td>
<td>VARLOAD</td>
<td>“PIPE VARLOAD” on page 225</td>
</tr>
<tr>
<td>VET</td>
<td>Reads or writes data to, or from, a virtual screen belonging to a virtual OST (VOST).</td>
<td>VOSTIO</td>
<td>“PIPE VET” on page 230</td>
</tr>
<tr>
<td>VTAM</td>
<td>Runs specific VTAM commands in a local or remote domain.</td>
<td>VTAM</td>
<td>“PIPE VTAM” on page 235</td>
</tr>
<tr>
<td>XCFMSG</td>
<td>Sends and receives z/OS XCF service messages.</td>
<td>XCFMSG</td>
<td>“PIPE XCFMSG” on page 238</td>
</tr>
<tr>
<td>XCFQUERY</td>
<td>Retrieves z/OS XCF service data for group members.</td>
<td>XCFQUERY</td>
<td>“PIPE XCFQUERY” on page 239</td>
</tr>
<tr>
<td>XCFTABLE</td>
<td>Retrieves or sets the state field maintained by the z/OS XCF service for a group member.</td>
<td>XCFTABLE</td>
<td>“PIPE XCFTABLE” on page 241</td>
</tr>
<tr>
<td>XIMATE</td>
<td>Translates uppercase, lowercase, ASCII, and EBCDIC characters.</td>
<td>XIMATE</td>
<td>“PIPE XIMATE” on page 243</td>
</tr>
<tr>
<td>$STEM</td>
<td>Same as STEM, plus reads or writes VIEW attribute variables associated with specified data variables.</td>
<td>$STEM</td>
<td>“PIPE STEM and PIPE $STEM” on page 202</td>
</tr>
<tr>
<td>$VAR</td>
<td>Same as VAR, plus reads or writes VIEW attribute variables associated with specified data variables.</td>
<td>$VAR</td>
<td>“PIPE VAR and PIPE $VAR” on page 222</td>
</tr>
<tr>
<td>&lt; (From Disk)</td>
<td>Reads data from DASD into the pipeline.</td>
<td>&lt;</td>
<td>“PIPE &lt; (From Disk)” on page 245</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 is treated as a first stage and the last stage generated by APPEND has 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, see “PIPE (NCCF)” on page 19.

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 uses ||||.

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.

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:

```bash
/* Route SWITCH and LIST DST output to operator with attributes*/

'PIPE (NAME APPNXMP)'
```
PIPE APPEND

| 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 cmds yellow*/ |
| CONSOLE' /* send command outputs to console */ |

PIPE BETWEEN

Syntax

BETWEEN:

\[
\begin{align*}
\text{BETWEEN} & \quad \text{INCL} \\
\text{INCLFRST} & \quad \text{INCLLAST} \\
\text{NOINCL} & \\
\text{number} & \quad \text{NOT} \\
\text{pos.len} & \quad \text{NOT} \\
\end{align*}
\]

Command Description

The BETWEEN stage 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 is included in the output. A message that matches a count as a second condition is always included.

INCLFRST

Specifies that the message or messages that match the first string is included in
the output. The second string, if specified, does not have its matching message included in the output. A message that matches a count as a second condition is always included.

**INCLLAST**
Specifies that the message or messages that match the second string is included in the output. The first string does not have its matching message 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 is not 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 begins. When you specify a wild card (*) for len, the remainder of each line is searched. When you do not specify pos.len, each entire line is searched. You can specify the letter $ for the length if the specification is followed by a delimited string. The BETWEEN stage replaces the letter with the length of that delimited string.

**/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**.

**NOT**
Indicates that a break occurs if the specified string is not found.

**/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.

**Example: Dividing Messages**
The following divides a message stream into groups, as shown in the sample output:

```
PIPE < CNMPNL1.CNM63
    BETWEEN 1.5/:XMP./1.6/:EXMP./
    CONSOLE

:XMP.
    CNM630I NETVIEW SUBSYSTEM INTERFACE IS NOT INITIALIZED
    :EXMP.
    :XMP.
    CNM631I NETVIEW PROGRAM TO PROGRAM INTERFACE IS ACTIVE ON xxxx
    :EXMP.
    :XMP.
    CNM632I NETVIEW PROGRAM TO PROGRAM INTERFACE IS BEING TERMINATED
    :EXMP.
```
PIPE CASEI

Syntax

CASEI:

---CASEI--- stage_specification

Synonyms

Stage Name  Synonym

CASEI  CAS

Command Description

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

stage_specification

The stage specification being modified, including its operands.

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.
NETVASIS 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.

### PIPE CHANGE

#### Syntax

**CHANGE (with specifications as arguments):**

```
| CHANGE | 1.* | /string1/string2/ | numchg |
|        |    | /string1/ /string2/ |        |
| position.length |        | | numchg |
```

**CHANGE (with parameters from secondary input):**

```
| CHANGE | numchg |
```

#### Synonyms

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

#### Command Description

The CHANGE stage 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 the first string is not found) are passed to the secondary output stream. If either output stream is defined, but disconnected, messages usually 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.

You can also specify a secondary input stream to the CHANGE stage (see the previous syntax diagram labeled CHANGE (with parameters from secondary input)). Because this method provides an alternative means of providing parameters for input (rather than through the use of arguments), using secondary input is particularly useful when you must make many changes, or a number of changes that are computed (rather than predetermined). This also provides a means of specifying strings longer than 255, or strings that contain characters that might be interpreted as delimiters.

Consider a pipe that might be coded like this:
The contents of `msgSource` is the source of messages to be changed. The contents of the `changeSpecs.stem` is the key to the advanced functions. The CHANGE stage expects the lines in the `changeSpecs.stem` to be in groups of three, where each group of lines has the following syntax:

**First line::**

```
  position.length
```

**Second line::**

```
  string1
```

**Third line::**

```
  string2
```

See “Example: Using CHANGE with parameters from secondary input” on page 34 for an example of using a secondary input stream.

**Note:**

1. The forward slash characters (/) in the syntax diagrams are for illustration. Any character except alphanumeric characters, parentheses, blanks, nulls, and national characters (@, #, and $) can be used instead of the slash character to delimit the strings.

2. Changes are applied in the order in which they are supplied, so some strings inserted by early changes might be modified by later changes. There is still a maximum that you can specify for the number of changes to any one message; that maximum is still specified as an argument on the CHANGE 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**

The CHANGE stage ends when the primary input stream disconnects or when both output streams disconnect.

**Operand Descriptions**

`numchg`

The maximum number of substitutions to be made in each message. Note that
there can be multiple lines per message. If \texttt{numchg} is not specified, the default is to change all occurrences in each message.

\textit{position.length}

Specifies the character position in each message where searching begins, and the length of the search. If you specify a \texttt{length} of *, the remainder of each line is searched. If you do not specify a \texttt{position.length}, the entire line is searched. For the \textit{with specifications as arguments} format only, you can specify the letter $S$ for the length if the specification is followed by a delimited string. When this is done, the CHANGE stage replaces the letter $S$ with the length of that delimited string.

\texttt{R} Specifies that the case of \texttt{string1} is to be respected. This parameter is only valid for the \textit{with parameters from secondary input} format, and is the default. For the \textit{with specifications as arguments} format, case is respected.

\texttt{I} Specifies that the case of \texttt{string1} is to be ignored. This parameter is only valid for the \textit{with parameters from secondary input} format. For the \textit{with specifications as arguments} format, specify the CHANGE stage on the CASEI stage.

\texttt{string1}

The character string to be replaced in input lines.

\texttt{string2}

The character string to replace \texttt{string1}.

\textbf{Usage Notes}

- The CHANGE stage cannot be a first stage.
- If you code a pipeline using a string that includes special characters, remember that these characters might be interpreted as delimiters. To avoid this problem, code the CHANGE stage using parameters from secondary input.

\textbf{Example: Changing All Occurrences of a String}

The following example changes all occurrences of 'AAA' to 'ZZ'.

\begin{verbatim}
PIPE LITERAL /AAABBBAAA/
 | CHANGE /AAA/ZZ/
 | CONSOLE
\end{verbatim}

The following output is produced:

\begin{verbatim}
ZZBBBZZ
\end{verbatim}

\textbf{Example: Inserting Constants}

The following example inserts a constant at the start of each line.

\begin{verbatim}
PIPE NETV DISCONID
 | CHANGE //DISCONID: /
 | CONSOLE
\end{verbatim}

Output similar to the following output is produced:

\begin{verbatim}
DISCONID: CNM4921 OPERATOR ID CONSOLE ID CONSOLE NAME
DISCONID: CNM4921 ---------- ---------- ----------
DISCONID: CNM4921 NTVFOPPT EXTENDED NTVFTF0
DISCONID: CNM4921 AUTOAON EXTENDED AUTONF0
DISCONID: CNM4921 END DISPLAY
\end{verbatim}
Example: Using CHANGE with CASEI Stage

Starting from column 2, the following example changes up to 4 occurrences of 'A' or 'a' to 'Zz'.

NETVASIS PIPE LITERAL /AaBbCcAa/
CASEI CHANGE 2.* /a/Zz/ 4
CONSOLE

The following output is produced:
AZzBbZzZzCcZza

Example: Using CHANGE with parameters from secondary input

As an example of using parameters from secondary input, consider the following REXX program:

/* Example program to show results of using a secondary input */
/* stream on the CHANGE stage */
changeSpec.1 = '1.*'
changeSpec.2 = 'CNM492I'
changeSpec.3 = 'DISCONID output:'
changeSpec.4 = '22.4'
changeSpec.5 = '-----'
changeSpec.6 = 'opid'
changeSpec.7 = '35.6'
changeSpec.8 = '------'
changeSpec.9 = 'consid'
changeSpec.10 = '49.8'
changeSpec.11 = '---------'
changeSpec.12 = 'consname'
changeSpec.0 = 12
'PIPE (END %) NETV DISCONID' ,
| A: CHANGE' ,
| 'CONSOLE' ,
| '% STEM changeSpec.' ,
| 'A:'

Output similar to the following output is produced:

DISCONID output: OPERATOR ID CONSOLE ID CONSOLE NAME
DISCONID output: ---opid--- --consid-- --consname--
DISCONID output: NTVF0PPT EXTENDED NTVFTF0
DISCONID output: AUTOAON EXTENDED AUTOF0
DISCONID output: END DISPLAY

PIPE CHOP

Syntax

CHOP:

```
CHOP width column 0 BEFORE ANYof STRING /string/
```

Command Description

The CHOP stage 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 searches 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 value.

**column**

The column after which truncation is to occur.

**NOT**

Specifies that truncation is relative to the first character or string that does not
match the specified target.

**offset**

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

**STRING**

Specifies that CHOP searches 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.

**Example: Truncating Lines**

The following example truncates all lines after column 44.

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

PIPE COLLECT

Syntax

```
COLLECT:
```

```
COLLECT [number] FREEFORM [MAX]
```

```
ASBEFORE [BREAK [ONCHANGE] pos.len]
```

```
BREAK [AT] [AFTER] [BEFORE] [pos.len] [NOT] [BLANK] [NULL]
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLLECT</td>
<td>COL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Operands</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASBEFORE</td>
<td>ASB4</td>
</tr>
<tr>
<td>FREEFORM</td>
<td>FF</td>
</tr>
</tbody>
</table>
Command Description

The COLLECT stage 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 recognizes only the first six lines as label lines. The remainder are treated as data lines.

- Unless FREEFORM is specified, all lines, except the last, read from the primary input stream os 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 is used as an end line.

If a secondary input stream is not defined for the COLLECT state and FREEFORM is specified:

- The line type is not validated or modified for any line.

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

- If the first line of a multiline message is a data line, it is handled as a control line.
- For each multiline message produced, COLLECT l accepts 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.
- 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 creates one multiline message containing all input lines, and must delay its output until the previous stage disconnects.
**PIPE COLLECT**

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

**ASBEFORE**
Specifies that the output is 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 value.

**BEFORE**
Specifies that when the message containing the matching string is detected, it is kept to begin a new multiline message.

**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.

**FREEFORM**
No validation of the existing line types of input message is performed. As a result, the line types are not modified by the COLLECT stage. For example, if a message with a line type of Control follows a message with a line type of Data, and a message with a line type of Data follows a message with a line type of End, the collected multiline message is built with this erroneous sequence.

**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 attempts to find the specified delimited string. You can specify the letter $ for the length if the specification is followed by a delimited string. The COLLECT stage replaces the letter with the length of that delimited string.

**NOT**
When specified on the first string, indicates that the selection begins if the string is not found. When specified on the second string, indicates that the selection ends if the string is not found.

**ONCHANGE**
Specifies that when the message being examined is different from the previous message in the range indicated by *pos.len*, then any previously accumulated lines are output and a new multiline message is begun with the message being examined. Only the first line of a multiline input message is examined.

*/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.

**BLANK**

Specifies that the character string for which to search contains only blanks. The search occurs in the range specified by the `pos.len` parameter, but if the data contains only blanks, a match is recognized regardless of the length specified.

**NULL**

Specifies that the stage is to search for no data whatsoever, that is, null data (not even blanks). This means that a match is recognized only when the data is shorter than the number specified for position in the `pos.len` parameter.

**Usage Notes**

- COLLECT cannot be a first stage.
- Each of the COLLECT parameters (except for FREEFORM) 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 stage following a STEM stage, see the description of the COLLECT operand of the STEM stage 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 does 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 bears 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 the NetView program.
- COLLECT can cause a complex pipeline to become deadlocked or clogged. For information on resolving clogged pipelines, see "Clogged Pipelines" on page 311.

**Example: Converting Single-Line Messages to Multiline**

To present the output of LIST STATUS=_TASKS as a multiline message, enter:

```
PIPE NETVIEW LIST STATUS=TABLES
    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 stage 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 CORRCMD D NET,LINES
    SEPARATE
    NLOCATE 1.7 /IST350I/
    COLLECT ASBEFORE
    CONSOLE

    ---> IST097I DISPLAY ACCEPTED
    --->
    IST354I PU T4/5 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 stage 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.

```
/* 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 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 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 D 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 are still label lines.

The source for this example is in CNMS1101.

```rexx
/* REXX Example */
PIPE (NAME DISPAL END \'', /* generate data */
  CORCMD MYS D A,L', /* handle lines individually */
  /* SEP preserves the line type {ctl, label, data, end} */
  A: TAKE 3', /* first 3 to primary, rest 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 "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:**

```

```

**Synonyms**

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

**Command Description**

The CONSOLE stage 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 memory 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 HELDMG stage.

An exposed message is a message that fits all of the following criteria:
• Passed to DSIEX02A
• Copied or routed to satisfy ASSIGN stage 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 stages. 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 43.

Reference: See the IBM Tivoli NetView for z/OS Programming: 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.
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 the NetView program from the MVS environment can have a stage 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.

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 00DC0100 00C90024 | ü* I *|
040AE270 1127520C D5E35F7 C5AO4040 00000000 | **â=NTV7E |
040AE280 00000000 E0D6440 40404040 0017A000 | *µ|
040AE290 00000624 007FD068 00000000 | ** *)Ç+5 *Çâ|
040AE2A0 04486658 40404040 00003000 00000000 | *Çâì** *|
040AE2B0 00000000 00000000 00000000 | *Çâì** *|
040AE2C0 00000000 00000000 00E20010 | TOM
040AE2D0 00000000 00000000 00000000 00000000 | *Çâì** *
040AE2E0 00000000 00000000 00E20010 | TOM
040AE2F0 D4E2740 00000000 00000000 | MSG
040AE300 00000000 0080011F 007FD068 00000624 | * *5 *)Ç  **
040AE310 01000002 C6F7F9D4 E5E24040 00000000 | * *F7MVS |
040AE320 00000000 00000000 00000000 00000000 | *F7MVS |
040AE330 E2E353F0 F0F0F0F5 C5F3F2F0 C57E50E5 | STC0005E320EGNV
040AE340 00000000 00000000 AC01B383 B8A3C1B3 | ÔÇ=**,****
040AE350 E3D0440 40404040 00100010 000C0011 | TOM  ** * * *
040AE360 00000000 044E80B0 | **Y* |
- NTV7E TOM

-------- Aifro (GDS) ---------------------
040AE800 00E20010 000C0011 | S  ** *
040AE810 00000000 00000000 00020006 00CE0007 | K  * *
040AE820 D4C4240 00000001 00300001 00000624 | MOD  **  * *
040AE830 F1F14BFD F74BF5F5 4BF3F2FD 00F9F59F | 11.27.52.32 1995
040AE840 F3F2F500 00000000 00000000 00000000 | 325
040AE850 C6F7F9D4 E5E24040 E2E3C3F0 F0F0F0F5 | F7MVS STC0005
040AE860 008A0002 00000004 D4E5E240 C8C2C2F5 | 11.27.52.32 1995
040AE870 F5F1F640 00000000 00000000 00000000 | 510
```

Chapter 2. Pipeline Stages and Syntax
PIPE CONSOLE

040AE880 00000000 08000000 60000000 000001F5 | * * - *5
040AE899 007FD068 00000000 00001800 E2E3C3F0 *}Ç * STC0
040AE8A0 F0F0F0F5 40404040 40404040 00000000 0005
040AE8B0 00000000 AC0183B3 327B9325 01000002 00C.*¢** *
040AE8C0 00000000 00000000 00000000 00000001 *
040AE8D0 00000000 00000000 0000C5F3 F2F0C5C7 0005
040AE8E0 D5E5D3D6 C3C1D340 4040 NVLOCAL
- NV7E TOM

04486650 004C007A 00C5002E 00AC0183B3 327B9325 01000002 00C.*¢** *
04486660 1127520C D5E3E5F7 C5404040 00000000 E3D6D440 40404040 00570004
04486670 000100F4 00E4C9C5 C9400306 C9400306 004C007A 00C5002E 004C007A 00C5002E
04486680 F5F2F4C4 C1E3C57E 00E4C9C5 C9400306 C9400306 00E4C9C5 C9400306 00E4C9C5 C9400306
04486690 F5F2F4C4 C1E3C57E 00E4C9C5 C9400306 C9400306 00E4C9C5 C9400306 00E4C9C5 C9400306
- NV7E TOM

PIECE COREVENT

Syntax

COREVENT:

COREVENT— event_type— timer— retry_interval

Command Description

The COREVENT stage takes a message or MSU from the primary input stream and takes information from it that can be used for state correlation. A correlation event is built from the message or MSU and sent from the pipe stage to the DSICORSV subtask for correlation processing. The original message or MSU is passed onto the primary output stage of the pipe. In most cases, the pipe discards the original message or MSU after it has been sent off for correlation processing. If the event cannot be sent to the DSICORSV task, error message BNH785I is generated and placed on 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

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

Operand Descriptions

**event_type:**

Indicates a name that is used to describe the resulting correlation event. This name is used in the correlation rule base to indicate that a rule applies to this event.
timer
Indicates (in seconds) how long to wait for a correlation reply before
discarding the event. This parameter overrides the PURGETIMER value in the
CNMSTYLE member.

retry_interval
Indicates (in seconds) how long the DSICORSV task keeps retrying to send the
event to the correlation engine in UNIX System Services if there are
communication problems with the correlation engine. This parameter overrides
the RETRYTIMER value in the CNMSTYLE member.

Usage Notes
• The following information from messages is used by default when building an
event:
  – The date of the message
  – The message ID
  – The message text (including additional lines in an MLWTO)
  – A count of the number of text lines in the messages (1 for single messages, >1
    for MLWTOs)
  – The job name of the message originator
  – The job number of the message originator
  – The NetView host where the message was created

Example: Creating a Correlation Event

To create a correlation event type of sample and wait five minutes for a response,
enter:
PIPE COREVENT SAMPLE 300

PIPE COREVTDA

Syntax
COREVTDA:

Command Description
The COREVTDA stage takes a message that has event data attached to it, creates a
new MLWTO message from the input stage, and places it on the secondary output
stream for processing in the pipe. Each line of the message contains a name/value
pair, such as MSGID=DSI002I. The input message is copied 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>
**PIPE COREVTDA**

**Termination Conditions**

COREVTDA terminates when the primary input stream or both of the output streams are disconnected.

---

**PIPE CORRCMD**

**Syntax**

CORRCMD:

```
CORRCMD ( CGI NOPANEL MOE numbe)---cmdlabel:cmdtext
```

**Synonyms**

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

**Command Description**

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

If you use the *label syntax*, CORRCMD automatically synchronizes the output of the stage 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 stage 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 stage as the target stage 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 see the *IBM Tivoli NetView for z/OS User's Guide: NetView*.

**Usage note:** Commands queued to the task running the pipe (such as with a null label) are run at high priority because they must interrupt the pipe itself. Therefore, subsequent commands invoked using the same CORRCMD pipe stage (when the CORRCMD stage is not a first stage) might interrupt previous commands. If command procedures are queued by the CORRCMD stage using a null label and if any procedure invokes the UNIQUE command, then one or more procedures can be unexpectedly affected by the UNIQUE processing. See the description of the UNIQUE command in the *IBM Tivoli NetView for z/OS Customization Guide* for more information.

**cmdtext**
Specifies the command to be processed. If cmdtext consists of a label only and CORRCMD is not the first stage, then CORRCMD routes commands in the input stream as specified by the label. Such commands cannot contain a form feed character (OC'X) unless the command being routed is PIPE.

**Note:** A PIPE command can contain any character. If your command contains the form feed character (OC'X), 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 stage. If the return code is not zero, it inserts message DWO369I containing the return code into the stream after any messages the stage might have returned. The return codes in the message are from either the command or the implied wait. If you do not specify MOE, return codes from stages are ignored.

**NOPANEL**
When NOPANEL is specified, the stage is not allowed to display a full-screen panel. If it attempts to do so, message BNH113W is inserted into the pipeline and the stage receives 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 overrides the value specified for the stage by CCDEF. Valid values are 1 - 10000000. The default value is 60, unless another value is defined with CCDEF.
**Usage Notes**

- The timeouts and termination conditions to be used are defined by the customer using the CCDEF stage.
- When RMTCMD or service point stages are entered using CORRCMD, termination conditions are inherent in the stage. Other termination conditions defined by CCDEF are redundant.
- When the target stage (following the RMTCMD label syntax) is PIPE, the CORRCMD stage does not force output to be collected into one multiline message. You can add a COLLECT stage to your pipeline specification for better performance.
- When you use the RMTCMD label syntax to transfer stages to a NetView V2R3 domain, the target command must be PIPE. However, you can use the pipeline to run other stages. See "PIPE NETVIEW" on page 152 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 IBM Software Support.</td>
</tr>
<tr>
<td>-108</td>
<td>Stage is type=I or type=P.</td>
</tr>
<tr>
<td>-112</td>
<td>Stage search failed. This is usually because the stage 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.

**Example: Issuing a VTAM Command**

To issue the VTAM stage 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 stage to a remote domain (by the RMTCMD stage) 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 stage is arranged.
• At the remote domain (CNM18):
  – The VARY stage is run.
  – An appropriate wait time for the response to the stage is arranged.
  – The wait is terminated when the expected response is received.
  – Output from the stage 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 stages at a remote domain. Commands are passed on the input stream. For example the following stages executes each stage stored in the stem SOMECMDS. and stores all the responses from the stages in the RESPONSES. stem:

```
PIPE STEM SOMECMDS. | CORRCMD CNM02: | STEM RESPONSES.
```

**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 is used by the USEDAT stage. The remote stage conversation is also shared with others by using the RMT02 autotask. The stage, with the data, is 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 stage.

---

**PIPE CORRWAIT**

**Syntax**

```
CORRWAIT:
```

```
CORRWAIT  [ MDE ]  [ NOSLOGR ]  [ interval ]  [ seqWait ]
```

**Synonyms**

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

The CORRWAIT stage allows time to pass while asynchronous messages generated by the previous 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 \textit{interval} parameter.

Asynchronous or delayed messages are those which return to the NetView program from commands running in another task (SWITCH, for example), or at another NetView system. When asynchronous responses are expected from any command issued in a pipe NETVIEW or VTAM stage, the next stage must be CORRWAIT. Otherwise, the stages in between do not see the asynchronous messages, and results are unpredictable.

The CORRWAIT stage can be used as a first stage (to provide a definite waiting period) or following a stage generating asynchronous 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</td>
</tr>
</tbody>
</table>

Termination Conditions

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

\texttt{PIPE NETV RMTCMD LU=CNM02,LIST XXX|CORRWAIT 60|STEM RMTDATA.}

RMTCMD completes its local processing in less than a second, but processing of the stage 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 stage completes, CORRWAIT is notified and the wait ends immediately. Functionally, it is as if the preceding stage did not complete processing until the stage at the remote NetView system 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 stage. For example, if you create a stem variable containing a series of LIST DST= commands for all your DSTs and then issue:

\texttt{PIPE STEM LISTCMDS.[NETVIEW]CORRWAIT 20| ...}

Each LIST stage 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 many 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 10-character, right-aligned, 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.

### Completion Code

**Event**

+0000000000  
A command thread completed (see note).

+0000000008  
A timeout occurred.

+0000000012  
A GO command was issued.

+0000000016  
A task executing a thread terminated.

+0000000032  
An ABEND occurred on a task that was executing a thread.

-0000000005  
A RESET occurred on a remote task that was executing a thread.

**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 - 10,000,000. The default is 1.

- An asterisk (*) can be specified for `interval`. When an asterisk is specified, CORRWAIT never times out. The following conditions end the wait:
  - A GO command
  - A RESET command
PIPE CORRWAIT

- A PIPEND pipe stage
- Secondary output disconnect
- Other conditions that end a wait

**seqWait**
The second number (after interval) specifies the time in tenths of a second to wait between messages, after the first message. This is to allow the timeout to change after responses have begun to flow. Valid values are in the range of 0 - 10,000,000. The default is the same time period (ten times the value of the argument) as the interval. Note that, for purposes of determining the "first" message, command echos are ignored, as are certain (unseen) internal NetView flows.

**aftMLWait**
The third number (after seqWait) specifies the time in tenths of a second to wait following receipt of the first multi-line message. This allows you to change the wait period when you know the pattern of responses. Valid values are in the range of 0 - 10,000,000. The default is the same time period as was specified for the seqWait.

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

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

**NOSLOGR**
Suppresses the recording of command responses in the system log and the Canzlog log. Any response that is received after the wait is terminated (such as by timeout) is not affected.

**Usage Notes**
- The display of W 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 (using 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 containing terminating conditions (for example, TOSTRING or TAKE FIRST) after a CORRWAIT stage. 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>
</tbody>
</table>
12 A GO command was entered.
16 A task executing a thread terminated.
32 An ABEND occurred on a task executing a thread.

**Example: Causing a Wait with CORRWAIT**

To display *We will now wait 9 seconds*. and wait for nine seconds, enter:

```
PIPE CORRWAIT 9
   LITERAL /We will now wait 9 seconds./
   CONSOLE
```

**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 IST061I VARY ACT...FAILED message, but if good responses are received, we must count them.

**Note:** Certain VTAM message IDs are release dependent.

```
PIPE VTAM VTNET,ACT,SCOPE=ALL,ID=NTFELN7E
   CORRWAIT 100
   TOSTRING 1.7 /IST061I/
   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 PIPEND 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 can examine the incoming messages one at a time and can create a named SAFE, if necessary. Like the DROP stage in this example, your procedure can produce a message to drive PIPEND when it is appropriate to end the wait.
### Example: Early timeout following an expected message flow

This pipe waits three seconds for responses to begin and one second following any (unexpected) single-line message(s). Following receipt of an MLWTO (the expected IEE114I), it times out immediately.

```
PIPE MVS 0 A,L | WAIT 3 I 0 | CONS
```

### Example: Using a Secondary Output Stream with CORRWAIT

CORRWAIT can be coded to end when one of the following situations occurs:
- 60 seconds has elapsed.
- The operator entered the GO command.
- A task end or ABENDS.

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

```
... 
A: CORRWAIT 60
... 
% 
A: 
| NLOCATE 1.11 /*0000000000*/  
| TAKE 1  
| HOLE  
... 
```

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 +0000000000 is processed, CORRWAIT terminates.

### PIPE COUNT

#### Syntax

**COUNT:**

- `COUNT MESSAGES`  
- `COUNT TIMES`  
- `COUNT EACHLINE`  
- `COUNT EACHMSG`  
- `COUNT LINES`  
- `COUNT MAXLINE`  
- `COUNT MINLINE`  
- `FROM 0`  
- `FROM number`

#### Synonyms

<table>
<thead>
<tr>
<th>Stage Operands</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>
</tbody>
</table>
Stage Operands | Synonym
---|---
MINLINE | MINL

### 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 can change the line count attributes.

See “Example: Line Counts Reset by Command” on page 57 for information on modification of set line attributes, see HELP PIPE EDIT for information on the EDIT stage, and HELP PIPE CONSOLE for information on the CONSOLE stage.

**Note:** Do not use EACHLINE 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.
PIPE COUNT

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 counts each message line passed to it and returns 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 passed to the primary output stream.

Each multiline message is counted as one message.

MINLINE
Specifies that COUNT counts each message line passed to it and returns 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.
- 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.

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 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 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 are preserved.

```
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 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 INFILE 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:
```

**Synonyms**

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

**Command Description**

The CPDOMAIN stage 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 DSI6DST must be active on both the local node and target node.
- The program-to-program interface (PPI), as well as the VTAM receiver, ISTMTRCV, must be active on the target node.
- CNMI=YES is specified in the CNMSTYLE member at both the local node and the target node.
- Because CPDOMAIN uses an asynchronous request, it must be followed by the CORRWAIT stage.
- CPDOMAIN produces a DWO969I message for each successful conversion and a DWO843I message for each asynchronous failure (return codes are for synchronous failures). DWO843I messages are usually accompanied by other NetView or VTAM messages which helps diagnose the problem.

**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 produces return code +0000000608 CPX.

**Return Code**

**Meaning**

+0000000008  
The input is not valid.

+00000006xx  
DSIMQS to DSI6DST failed. The return code is 600 plus the DSIMQS return code.
Example: Converting a Hardcoded CP Name

The following example converts a CP name that is hardcoded:

\[
\text{PIPE (END ;)} \text{ a: CPD USIBMNT.NTFEMVS | WAIT 10 | CONS; a:| COLOR WHI | CONS}
\]

Example: Converting a CP Name Specified In a Variable

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

\[
\text{PIPE (END ;) VAR var1| a: CPD | WAIT 10 | CONS; a:| COLOR PIN | CONS}
\]

PIPE CZR

Syntax

\[
\text{CZR}
\]

Alternate syntax

\[
\text{CZR}
\]

Command Description

The CZR stage retrieves messages from the Canzlog database. Messages retrieved are identical with the original message at the time it was logged. Use the CZR stage in conjunction with the CNMECZFS command, so that filtering can be applied prior to message identification. The CZR stage returns only messages that match the current filter established by CNMECZFS; the input time or message identifier serves as a starting point for a search of Canzlog data for messages matching the filter.

If a secondary output is defined and connected, a single line is written there at the completion of operations. This single line consists of three numbers, as follows:

1. Number of messages actually produced by the CZR stage on its primary output
2. The Canzlog database ID (CzID) of the next message from which your search can resume, if you want to continue the search. If the search reached the end of data, a minus one (-1) is returned.
3. A zero (0), if all requested messages were produced; otherwise, an indication of how much time the search consumed in units of 0.004096 seconds.

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**

CZR terminates when any of these conditions is met:
- When its primary output stream is disconnected.
- When the time for searching defined by DEFAULTS CZBRWAIT has expired.
- When the last record in the direction indicated has been examined.

**Operand Descriptions**

`count`

The number of messages wanted. The CZR stage continues to search for messages in the direction indicated.

`CzID`

Indicates that Canzlog database ID is supplied as a starting point. The association of a CzID with a message can last from a few minutes to a few hours, depending on usage. Your usage of the CzID is limited to values obtained during a program run, as from the secondary output of a previous invocation of the CZR stage, and continuing without any wait or suspension.

`decimalCzID`

A CzID obtained from a previous invocation, from examination of a message using CzID edit stage, or from a DISPMSG display. Be careful to use fresh (meaning recently obtained) numbers, as all CzIDs expire at some point. Use this with CzID keyword.

`direction`

A plus sign (+) or a minus sign (-) indicating the direction of search. The CZR stage searches for messages matching the current filter. Because of the search, the input time or CzID can be approximate. The default minus (-) means to search UP toward older messages. A plus sign (+) means search DOWN toward more recent messages.

`STCKvalue`

A store-clock value in EBCDIC, where CZR should begin to search for your message. Use this with the TIME keyword. If fewer than 16 characters are given, the value is padded on the right with zeroes. The CZR stage can extrapolate from FFFFFFFF to give you the most recent message, but when searching upward (with minus sign), use a date that is within the range of the available data to avoid I/O errors.

`TIME`

Indicates that a time in STCK format is supplied as a starting point. Note that times in U.S. standard format can be converted to STCK format using the edit orders DTS and C2X together. Be careful not to specify a time earlier than any record in the Canzlog data, because this causes an error.

**Command Description**

Requested messages are output on the primary output stream. If a secondary output is connected, a message with three numbers is displayed, indicating:
- The number of messages returned.
- A CzID that can be used to continue search in the same direction.
- An indication of how long the operation took in units of about 1/244 seconds (more precisely, 1.048576/256 second units). The number returned can be less than the number requested due to RESET, excessive errors (skipped segments),
or excessive time. The suggested starting point for continued search can be minus one (-1); this means that there are no more messages in the indicated direction.

**Usage Notes**

- A CzID is a direct index to a message and is the fastest way to locate a message. Depending on system demands, CzIDs might be stable for several hours or for only several seconds. If a wait occurs between obtaining a CzID and its use, or if an unexpected message is obtained, start your search again using the last known good store-clock value.
- You can use CZR to copy a subset of the Canzlog data, by choosing an appropriate filter and sending the output of CZR to another data set or alternate destination.
- When the time in the TIME argument is later than any record and the search is upward (-), then the CZR stage starts with the actual latest record. This search yields the most recent records. However, if the time is earlier than any record in the Canzlog log, even with a downward search (+), an error is reported.

### PIPE DELDUPES

**Syntax**

DELDUPES:

```
DELDUPES:  DEELDUPES  PAD '00'X  KEEPFIRST
            PAD  'xx'X  KEEPLAST
                   ALL
                   I.*
                   position.length
```

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELDUPES</td>
<td>DELDUP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Operands</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<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 cause length to default to 1.

If length is larger than the available characters, all available characters are used and the compared field is 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.
Example: Display Last Logtime

This 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 11/02/18 13:25:04
Smith, Fred 11/02/18 13:29:21
Collins, Mary 11/02/23 17:01:55
Doe, John 11/02/23 09:00:00
Howe, Tom 11/02/23 04:14:20
Jones, Fred 11/02/23 11:16:44
Collins, Mary 11/03/01 10:15:40

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

Collins, Mary 11/03/01 10:15:40
Doe, John 11/02/23 09:00:00
Howe, Tom 11/02/23 04:14:20
Jones, Fred 11/02/23 11:16:44
Smith, Fred 11/02/18 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.

PIPE DIVERT

Syntax

```
DIVERT:
```

| DIVERT | COLLECT |

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 is concatenated with the message being written to the output.

Usage Notes
• DIVERT requires at least two input streams.
• 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, see sample CNMSRPLY.

PIPE DROP

Syntax

DROP:

\[
\text{DROP:} \quad \\
\text{FIRST} \quad 1 \quad \text{MSGS} \\
\text{LAST} \quad \text{count} \quad \text{LINES} \\
\]

Synonyms

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

Command Description

The DROP stage 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 - 10,000,000. The default is 1.

\( \text{FIRST|LAST} \)

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

\( \text{LINES} \)

 Specifies that \( count \) is the number of lines within the input stream to be dropped. If LINES is not specified, DROP discards the number of messages indicated by \( count \).

\( \text{MSGS} \)

 Specifies that \( count \) is the number of messages within the input stream to be dropped.

Usage Notes

DROP cannot be the first stage.

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 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:

```
DUPLICAT 1
    number
    *
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Name</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

number

Specifies the number of copies to make in addition to the original message. If number is zero (0), then the input message is written to the output stream and no additional copies are made. If 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.

Example: Creating Four Consecutive Utilization Reports

In the following example, AUTO1 executes 4 TASKUTIL commands and returns the results to the console.

```
PIPE LITERAL /TASKUTIL/ | DUP 3 | CORRCMD /AUTO1: | CONSOLE
```
Additional Examples

See also “Example: Discover TSO Stacks Serving a User” on page 216 for an example of how to use the asterisk (*) DUPLICAT option.

**PIPE EDIT**

**Syntax**

**EDIT:**

```
EDIT - Global Order - Edit Phrase
```

**Global Order:**
Edit Phrase:

Input Order | Conversion Order | Output Order

Input Order:
<table>
<thead>
<tr>
<th>Location:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Conversion Order:
### PIPE EDIT

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASTYPE</td>
<td></td>
</tr>
<tr>
<td>B2C</td>
<td></td>
</tr>
<tr>
<td>CHKEY</td>
<td></td>
</tr>
<tr>
<td>C2B</td>
<td></td>
</tr>
<tr>
<td>C2D</td>
<td></td>
</tr>
<tr>
<td>C2F</td>
<td></td>
</tr>
<tr>
<td>C2G</td>
<td></td>
</tr>
<tr>
<td>C2GV</td>
<td></td>
</tr>
<tr>
<td>C2VG</td>
<td></td>
</tr>
<tr>
<td>C2P</td>
<td>scale</td>
</tr>
<tr>
<td>C2S</td>
<td></td>
</tr>
<tr>
<td>C2V</td>
<td></td>
</tr>
<tr>
<td>C2X</td>
<td></td>
</tr>
<tr>
<td>CNVDT</td>
<td>(from-template to-template)</td>
</tr>
<tr>
<td>CNVDTO</td>
<td></td>
</tr>
<tr>
<td>D2C</td>
<td></td>
</tr>
<tr>
<td>D2X</td>
<td></td>
</tr>
<tr>
<td>DT</td>
<td></td>
</tr>
<tr>
<td>DTS</td>
<td></td>
</tr>
<tr>
<td>ETIME</td>
<td></td>
</tr>
<tr>
<td>FOUND</td>
<td></td>
</tr>
<tr>
<td>F2C</td>
<td></td>
</tr>
<tr>
<td>G2C</td>
<td></td>
</tr>
<tr>
<td>GV2C</td>
<td></td>
</tr>
<tr>
<td>VG2C</td>
<td></td>
</tr>
<tr>
<td>JOBNAME</td>
<td></td>
</tr>
<tr>
<td>LEFT</td>
<td>number</td>
</tr>
<tr>
<td>LEFT</td>
<td>keep.discard</td>
</tr>
<tr>
<td>OPDT</td>
<td></td>
</tr>
<tr>
<td>P2C</td>
<td>scale</td>
</tr>
<tr>
<td>PREFIX</td>
<td>/string/</td>
</tr>
<tr>
<td>RIGHT</td>
<td>number</td>
</tr>
<tr>
<td>RVAR</td>
<td></td>
</tr>
<tr>
<td>STRIP</td>
<td></td>
</tr>
<tr>
<td>STRIPL</td>
<td></td>
</tr>
<tr>
<td>STRIPR</td>
<td></td>
</tr>
<tr>
<td>SUBSTR</td>
<td>position.length</td>
</tr>
<tr>
<td>UPCASE</td>
<td></td>
</tr>
<tr>
<td>V2C</td>
<td></td>
</tr>
<tr>
<td>X2C</td>
<td></td>
</tr>
<tr>
<td>YESNO</td>
<td></td>
</tr>
<tr>
<td>ZDT</td>
<td></td>
</tr>
</tbody>
</table>

### Output Order:

<table>
<thead>
<tr>
<th>Order</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PIPE EDIT</td>
</tr>
<tr>
<td>2</td>
<td>70</td>
</tr>
<tr>
<td>3</td>
<td>Pr</td>
</tr>
<tr>
<td>4</td>
<td>Program</td>
</tr>
<tr>
<td>5</td>
<td>ming: Pipes</td>
</tr>
</tbody>
</table>
## Synonyms

<table>
<thead>
<tr>
<th>Stage Operands</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOTOKEN</td>
<td>AUTOTOKE, IFRAUTOK</td>
</tr>
<tr>
<td>C2GV</td>
<td>C2VG</td>
</tr>
<tr>
<td>COLOR</td>
<td>COLOUR, LINEATTR, LINEATRS</td>
</tr>
<tr>
<td>COPYREV</td>
<td>COPYVERSE, REVERSECOPY, REVCOPY</td>
</tr>
<tr>
<td>DESC</td>
<td>IFRAUWDS</td>
</tr>
<tr>
<td>FINDLINE</td>
<td>LINE</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>MCSFLAGS</td>
<td>MCSFLAG</td>
</tr>
<tr>
<td>MSGORIGIN</td>
<td>MSGORIGN</td>
</tr>
<tr>
<td>MSGSENDR</td>
<td>MSGSENDER</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>TOPLINE</td>
<td>TL</td>
</tr>
<tr>
<td>UCHARS</td>
<td>IFRAUSRC</td>
</tr>
</tbody>
</table>
Stage Operands | Synonym
---|---
UFLAGS | IFRAUSRB
WORD | TOKEN, W1, W2, ... W9
WRITELINE | WL

**Command Description**

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

- Message or command data
- Line attributes
- Message attributes
- Literal data

With EDIT, messages or commands can be created or reformatted in any fashion that you want. 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 or commands from the input stream to the edit phrase.
- Writing messages or commands from the edit phrase to the output stream.

**Notes:**

1. 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.
2. There are many environments in which edit orders can be run:
   - PIPE EDIT
   - WHEN statements
   - REVISE statements
   - Acquire
   - AutoEdit
   - Common Base Events
3. All orders that are valid for REVISE actions are also valid for WHEN actions, except that when used for output only, the NEXT, NEXTWORD, and numeric char position can be used.

Input Orders
These orders define the source of data that is 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 can 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, see sample CNME2011 (SESMGET). To begin with simple examples, see “Example: Selecting a Word” on page 113 and “Example: Creating a Command” on page 113.

Table 3. Global Order Summary. For more information, see “Global Orders” on page 85.

<table>
<thead>
<tr>
<th>Global Order</th>
<th>Task Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBETEMP</td>
<td>Used to read in a Common Base Event template.</td>
</tr>
<tr>
<td>CONSZERO</td>
<td>Sets the 4-byte console ID to zero.</td>
</tr>
<tr>
<td>COPY</td>
<td>Copies one or more unread lines in a multiline message from input to output. The order of output is in ascending order.</td>
</tr>
<tr>
<td>COPYREST</td>
<td>Copies all unread lines in a multiline message from input to output.</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>
</tr>
<tr>
<td>FINDLINE n</td>
<td>Changes the current line to the absolute line number indicated by the argument.</td>
</tr>
<tr>
<td>FINDLINE /string/</td>
<td>Advances the current line to the line that contains the specified target string.</td>
</tr>
<tr>
<td>FWDLINE n</td>
<td>Moves the current line forward by the number specified.</td>
</tr>
<tr>
<td>LASTLINE</td>
<td>Resets the input to the last line of a multiline message.</td>
</tr>
<tr>
<td>LOGTO</td>
<td>Sets an option to log the messages that are displayed using the CONSOLE stage or ROUTE stage.</td>
</tr>
<tr>
<td>NAMEBIND</td>
<td>Creates a name/value pair that is recognized by the alert and message adapters.</td>
</tr>
<tr>
<td>NEXTLINE</td>
<td>Specifies both the READLINE and WRITELINE keywords, and advances both the input message line (source) and the output message line (destination).</td>
</tr>
<tr>
<td>NOEXPOSE</td>
<td>Sets the IFRAUNEX automation flag.</td>
</tr>
</tbody>
</table>
Table 3. Global Order Summary (continued). For more information, see “Global Orders” on page 85.

<table>
<thead>
<tr>
<th>Global Order</th>
<th>Task Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOLOGTO</td>
<td>Sets an option to not log the messages that are displayed using the CONSOLE stage or ROUTE stage.</td>
</tr>
<tr>
<td>NOTECHO</td>
<td>Turns off the echo flag (WQEMCSM) in the message that is logged or automated. NOTECHO does not affect the display.</td>
</tr>
<tr>
<td>ONTO</td>
<td>Redefines the logical end of the input line.</td>
</tr>
<tr>
<td>PAD</td>
<td>Defines the padding character to be used by all other orders.</td>
</tr>
<tr>
<td>PARSE</td>
<td>Defines how the WORD input order counts words.</td>
</tr>
<tr>
<td>READCBE</td>
<td>Used to switch the primary input to a multiline template message initialized by the CBETEMP order.</td>
</tr>
<tr>
<td>READLINE</td>
<td>Provides the next line of a multiline message to the input orders.</td>
</tr>
<tr>
<td>READSRC</td>
<td>Switches the primary input to the original message or MSU on the input stream of the EDIT stage.</td>
</tr>
<tr>
<td>RESET</td>
<td>Cancels all existing SKIPTO and UPTO orders.</td>
</tr>
<tr>
<td>RESETAUTO</td>
<td>Sets the IFRAUHND, IFRAUMTB, and IFRAUNEX automation flags to 0.</td>
</tr>
<tr>
<td>ROUTEZERO</td>
<td>Overrides route codes for a specific message.</td>
</tr>
<tr>
<td>SETACTION</td>
<td>Sets the IFRAUACN and IFRAUNVD automation flags.</td>
</tr>
<tr>
<td>SETAUTO</td>
<td>Sets the IFRAUMTB automation flag to 1.</td>
</tr>
<tr>
<td>SETBEEP</td>
<td>Sets an audible alarm for message output.</td>
</tr>
<tr>
<td>SETCLEAR</td>
<td>Sets the IFRAUCLR automation flag.</td>
</tr>
<tr>
<td>SKIPTO</td>
<td>Redefines the logical start of the input line.</td>
</tr>
<tr>
<td>TOPLINE</td>
<td>Resets the input to the first line of a multiline message.</td>
</tr>
<tr>
<td>UPTO</td>
<td>Redefines the logical end of the input line.</td>
</tr>
<tr>
<td>WRITELINE</td>
<td>Writes all text built by the output orders to the output message.</td>
</tr>
</tbody>
</table>

Table 4. Input Order Summary. For more information, see “Input Orders” on page 93.

<table>
<thead>
<tr>
<th>Input Order</th>
<th>Task Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIFR</td>
<td>Specifies that the input is the 256-byte AIFR body.</td>
</tr>
<tr>
<td>ALL</td>
<td>Inputs the text in the current line. Same as 1.*.</td>
</tr>
<tr>
<td>ASTYPE</td>
<td>Indicates how the address space was started (job type).</td>
</tr>
<tr>
<td>AUTHGRP</td>
<td>Specifies the ACEE group ID (ACEEGRPN), if available. If it is not available, returns <em>UNKNWN</em>.</td>
</tr>
</tbody>
</table>

The values that are supplied for the AUTHUSER and AUTHGROUP input orders are derived from the ACEE that is active at the time the message is routed through the z/OS subsystem interface. Certain authorized programs issue messages in an environment that causes them to be routed asynchronously through the CONSOLE address space. These messages show "*CONSOLF*" as the AUTHUSER value and a single asterisk (*) as the AUTHGROUP value. Messages that are issued from the master scheduler show "*MASTER*" for the AUTHUSER value, instead. Only authorized programs can issue messages in this way.
Table 4. Input Order Summary (continued). For more information, see “Input Orders” on page 93.

<table>
<thead>
<tr>
<th>Input Order</th>
<th>Task Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTHUSER</td>
<td>Specifies the z/OS ACEE user ID (ACEEUSRI), if available. If it is not available, returns <em>UNKNWN</em>. The values that are supplied for AUTHUSER and AUTHGROUP input orders are derived from the ACEE that is active at the time the message is routed through the z/OS subsystem interface. Certain authorized programs issue messages in an environment that causes them to be routed asynchronously through the CONSOLE address space. These messages show &quot;+CONSOLE&quot; as the AUTHUSER value and a single asterisk (*) as the AUTHGROUP value. Messages that are issued from the master scheduler show &quot;+MASTER+&quot; for the AUTHUSER value, instead. Only authorized programs can issue messages in this way.</td>
</tr>
<tr>
<td>CHKEY</td>
<td>Obtains the CHKEY as defined by system macro IEECHAIN. This is the step-name of a task or the job-name of a job.</td>
</tr>
<tr>
<td>CMDX</td>
<td>Inputs the first 88 (X'58') bytes of the IEZVX101 control block.</td>
</tr>
<tr>
<td>COLOR</td>
<td>Inputs text describing the line attributes.</td>
</tr>
<tr>
<td>CONSAUTH</td>
<td>Indicates the authority of the console issuing the command.</td>
</tr>
<tr>
<td>CONSNAME</td>
<td>Obtains the console name, or indicates the name of the console issuing the command.</td>
</tr>
<tr>
<td>CURGGMT</td>
<td>Obtains an 8-byte store clock value generated at the time the order is executed.</td>
</tr>
<tr>
<td>CZID</td>
<td>Obtains the Canzlog ID of a message or command or DOM that has been logged.</td>
</tr>
<tr>
<td>D4NV</td>
<td>Indicates whether the console name to which a message is delivered is owned by a NetView task.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Determines whether a message has been deleted.</td>
</tr>
<tr>
<td>disposition</td>
<td>Provide information about the disposition of the message.</td>
</tr>
<tr>
<td>flag_bytes</td>
<td>Produces a string that corresponds to the bit in the referenced flag byte.</td>
</tr>
<tr>
<td>FLGDSCDn</td>
<td>Indicates that the specified descriptor code byte is to be read.</td>
</tr>
<tr>
<td>FLGRTCDn</td>
<td>Indicates that the specified route code byte is to be read.</td>
</tr>
<tr>
<td>HDRMTYPE</td>
<td>Indicates that the one-byte message type indicator is to be read.</td>
</tr>
<tr>
<td>hexstring</td>
<td>Specifies a hexadecimal string.</td>
</tr>
<tr>
<td>IFRAUHND</td>
<td>Use as input the automation action flag from the message.</td>
</tr>
<tr>
<td>IFRAUIN3</td>
<td>Indicates that the 32 bits that are defined as IFRAUIN3 are used as input.</td>
</tr>
<tr>
<td>IFRAUMTB</td>
<td>Use as input the automation submission flag from the message.</td>
</tr>
<tr>
<td>IFRAUNEX</td>
<td>Use as input the forbid exposure flag from the message.</td>
</tr>
<tr>
<td>JOBNAME</td>
<td>Indicates the job name.</td>
</tr>
<tr>
<td>LEVEL</td>
<td>Specifies the data set concatenation level of the current line.</td>
</tr>
<tr>
<td>lineattr</td>
<td>Specifies that the input is one of the line attributes of the current line.</td>
</tr>
<tr>
<td>LINESENDER</td>
<td>Specifies the name of the sender.</td>
</tr>
<tr>
<td>MCSFLAGS</td>
<td>Provides a 16-bit output suitable as input to a C2B conversion.</td>
</tr>
</tbody>
</table>
Table 4. Input Order Summary (continued). For more information, see “Input Orders” on page 93.

<table>
<thead>
<tr>
<th>Input Order</th>
<th>Task Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRT</td>
<td>Indicates that the message has been exposed to the Message Revision Table.</td>
</tr>
<tr>
<td>msgattr</td>
<td>Specifies that the input is one of the message attributes of the current message.</td>
</tr>
<tr>
<td>MSGID</td>
<td>The message identifier of the received message.</td>
</tr>
<tr>
<td>MSGSENDTR</td>
<td>Indicates the name of the task that created or most recently sent the message.</td>
</tr>
<tr>
<td>MSUSEG</td>
<td>Indicates the contents of one segment of an MSU.</td>
</tr>
<tr>
<td>NVABLE</td>
<td>Returns &quot;Yes&quot; if a NETVONLY action can succeed, otherwise returns &quot;No&quot;. A NETVONLY action cannot succeed if the NetView procedural space is down, the subsystem router is down, or if the task defined by the ?MVSCmdRevision CNMSTYLE statement is inactive.</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>
</tr>
<tr>
<td>REPLYL</td>
<td>Returns the length of the reply ID in decimal format, with a leading plus sign (+).</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>
</tr>
<tr>
<td>/string/</td>
<td>Specifies a delimited character string.</td>
</tr>
<tr>
<td>UCHARS</td>
<td>Obtains the 16-byte &quot;user char&quot; area. In the MRT, this field is available only if previously set.</td>
</tr>
<tr>
<td>UFLAGS</td>
<td>Obtains the 2-byte &quot;user flags&quot; area. In the MRT, this field is available only if previously set.</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>
</tr>
<tr>
<td>WQE</td>
<td>Enables the WQE for conversion.</td>
</tr>
<tr>
<td>WTOKEY</td>
<td>Obtains the key field associated with the WTO system macro, which is the WQEKEY in system macro IHAWQE.</td>
</tr>
</tbody>
</table>

Table 5. Conversion Order Summary. For more information, see “Conversion Orders” on page 102.

<table>
<thead>
<tr>
<th>Conversion Order</th>
<th>Task Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASTYPE</td>
<td>From a 2-byte binary ASID, a one-character value is returned. <strong>Note:</strong> The message revision table (MRT) supports ASTYPE only as an input order.</td>
</tr>
<tr>
<td>B2C</td>
<td>Converts string of Boolean values to a character string.</td>
</tr>
<tr>
<td>CHKEY</td>
<td>Obtains the CHKEY as defined by system macro IEECHAIN. This is the step-name of a task or the job-name of a job.</td>
</tr>
<tr>
<td>C2B</td>
<td>Converts input to a string of Boolean values.</td>
</tr>
<tr>
<td>C2D</td>
<td>Converts input to a string representing a decimal number.</td>
</tr>
<tr>
<td>Conversion Order</td>
<td>Task Performed</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>C2F</td>
<td>Converts input to a string representing a signed floating point number.</td>
</tr>
<tr>
<td>C2G</td>
<td>Converts fixed-length string to double-byte (DBCS) string.</td>
</tr>
<tr>
<td>C2GV or C2VG</td>
<td>Converts varying length string to double-byte (DBCS) string.</td>
</tr>
<tr>
<td>C2P</td>
<td>Converts a packed-decimal number into a signed decimal.</td>
</tr>
<tr>
<td>C2S</td>
<td>Converts internal, floating point data into a 14-byte output string.</td>
</tr>
<tr>
<td>C2V</td>
<td>Converts a varying length string to a character string.</td>
</tr>
<tr>
<td>C2X</td>
<td>Converts input to a string representing its hexadecimal notation.</td>
</tr>
<tr>
<td>CHKEY</td>
<td>Obtains the CHKEY as defined by system macro IEECHAIN. This is the step-name of a task or the job-name of a job.</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>
</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>
</tr>
<tr>
<td>D2C</td>
<td>Converts a signed integer number into a fullword.</td>
</tr>
<tr>
<td>D2X</td>
<td>Converts a decimal number to hexadecimal representation.</td>
</tr>
<tr>
<td>DT</td>
<td>Assumes that 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. <strong>Note:</strong> The current GMT offset is used in interpreting the local date and time, whether a different offset was in effect at the given date and time. For example, if the given value was before the latest daylight saving time adjustment, the result can be off one hour from another interpretation of the same date and time of an application.</td>
</tr>
<tr>
<td>DTS</td>
<td>Assumes that 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. <strong>Note:</strong> The current GMT offset is used in interpreting the local date and time, whether a different offset was in effect at the given date and time. For example, if the given value was before the latest daylight saving time adjustment, the result can be off one hour from another interpretation of the same date and time of an application.</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>
</tr>
<tr>
<td>FOUND</td>
<td>Normally used after a SKIPTO or FINDLINE operation, FOUND translates a null string into No and any other string into Yes.</td>
</tr>
<tr>
<td>F2C</td>
<td>Converts a signed floating point number into a doubleword.</td>
</tr>
<tr>
<td>G2C</td>
<td>Converts double-byte (DBCS) data to single-byte (SBCS) data.</td>
</tr>
<tr>
<td>GV2C or VG2C</td>
<td>Converts double-byte (DBCS) data into a varying length single-byte (SBCS) string.</td>
</tr>
<tr>
<td>JOBNAME</td>
<td>From a 2-byte binary ASID, the corresponding job name is returned.</td>
</tr>
</tbody>
</table>
Table 5. Conversion Order Summary (continued). For more information, see "Conversion Orders" on page 102.

<table>
<thead>
<tr>
<th>Conversion Order</th>
<th>Task Performed</th>
</tr>
</thead>
<tbody>
<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>
</tr>
<tr>
<td>ODDBYTES</td>
<td>Alternately, keeps and discards the input data.</td>
</tr>
<tr>
<td>OPDT</td>
<td>Assumes that 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>
</tr>
<tr>
<td>P2C</td>
<td>Converts a signed decimal number into packed-decimal.</td>
</tr>
<tr>
<td>PREFIX</td>
<td>Adds a constant to the beginning of a string.</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>
</tr>
<tr>
<td>RVAR</td>
<td>From an input revision variable name, returns the current value or a null string.</td>
</tr>
<tr>
<td>STRIP</td>
<td>Removes all padding characters from the beginning and end of the input.</td>
</tr>
<tr>
<td>STRIPL</td>
<td>Removes all padding characters from the beginning of the input.</td>
</tr>
<tr>
<td>STRIPR</td>
<td>Removes all padding characters from the end of the input.</td>
</tr>
<tr>
<td>SUBSTR</td>
<td>Selects a subset of the input data.</td>
</tr>
<tr>
<td>UPCASE</td>
<td>Translates the standard 26-character Latin letters (as defined in code page 037) to uppercase. The asterisk argument is required.</td>
</tr>
<tr>
<td>V2C</td>
<td>Converts input to a varying length string.</td>
</tr>
<tr>
<td>X2C</td>
<td>Converts character data to internal hexadecimal format.</td>
</tr>
<tr>
<td>YESNO</td>
<td>Converts a 1-byte field to the character string Yes or No.</td>
</tr>
<tr>
<td>ZDT</td>
<td>Assumes that the input text is a store clock (STCK) and converts the value to a readable character string for Greenwich Mean Time in the format MM/DD/YY HH:MM:SS.</td>
</tr>
</tbody>
</table>

Table 6. Output Order Summary. For more information, see "Output Orders" on page 109.

<table>
<thead>
<tr>
<th>Output Order</th>
<th>Task Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOTOKEN</td>
<td>Sets the 8-character automation token in the message (IFRAUTOK).</td>
</tr>
<tr>
<td>COLOR</td>
<td>Sets presentation attributes for the output line.</td>
</tr>
<tr>
<td>CONSNAME</td>
<td>Sets the console name.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Deletes a message or command, or undeletes a previously deleted message or command.</td>
</tr>
<tr>
<td>disposition</td>
<td>Control or change the disposition, subject to certain system constraints.</td>
</tr>
<tr>
<td>FINDLINE</td>
<td>Functions the same as global order FINDLINE $n$ except that the target to be found is derived from the previous input order or conversion order. If the number is negative, EDIT counts from the end of the message.</td>
</tr>
<tr>
<td>flag_bytes</td>
<td>Accepts a string that corresponds to the bit in the referenced flag byte.</td>
</tr>
<tr>
<td>FLGDSCD$n$</td>
<td>Indicates that the specified descriptor code byte is to be written.</td>
</tr>
<tr>
<td>FLGRTCD$n$</td>
<td>Indicates that the specified route code byte is to be written.</td>
</tr>
</tbody>
</table>
Table 6. Output Order Summary (continued). For more information, see “Output Orders” on page 109.

<table>
<thead>
<tr>
<th>Output Order</th>
<th>Task Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDRMTYPE</td>
<td>Indicates that the one-byte message type indicator is to be written.</td>
</tr>
<tr>
<td>LINETYPE</td>
<td>Defines the line type attribute of the output line.</td>
</tr>
<tr>
<td>MRT</td>
<td>Indicates that the message has been exposed to the Message Revision Table.</td>
</tr>
<tr>
<td>NEXT</td>
<td>Specifies that the input is to be placed into the output without an intervening blank.</td>
</tr>
<tr>
<td>NEXTWORD</td>
<td>Specifies that the input is to be placed into the output with an intervening blank.</td>
</tr>
<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>
</tr>
<tr>
<td>SETGMT</td>
<td>Sets the IFRAUGMT value of the output message and zeros out the CANZLOG reference in the output message. The order is carried out only if the input available is exactly eight bytes.</td>
</tr>
<tr>
<td>UCHARS</td>
<td>Sets a 16-byte “user char” area. When used as an output order, you can specify data with a length other than 16. If the length is shorter than 16, it uses that; if the length is greater than 16, then it truncates the length to 16.</td>
</tr>
<tr>
<td>UFLAGS</td>
<td>Sets a 2-byte “user flags” area. UFLAGS (much like UCHARS) is supported as an input or output order in both the MRT and PIPE EDIT. For the MRT, this field accepts a string of up to 16 characters consisting of 0s, 1s, and Xs. These correspond to the requirement to clear, set, or leave intact the corresponding bit in the byte being referenced. For PIPE EDIT, you can specify data with a length other than 2. If the length is shorter than 2, it uses that; if the length is greater than 2, then it truncates the length to 2.</td>
</tr>
<tr>
<td>WTOKEY</td>
<td>For the Message Revision Table (MRT), WTOKEY sets the key field associated with the WTO system macro, which is the WQEKEY in system macro IHAWQE.</td>
</tr>
</tbody>
</table>

Table 7. Supported environments for EDIT orders

<table>
<thead>
<tr>
<th>EDIT Order</th>
<th>PIPE</th>
<th>MRT REVISE and WHEN</th>
<th>CRT REVISE and WHEN</th>
<th>Auto Edit and Acquire</th>
<th>CBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIFR</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ALL</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AMRF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASID</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASTYPE</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AUTHGRP</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AUTHUSER</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUTOMATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUTOMATED</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AUTOTOKEN</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>B2C</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 7. Supported environments for EDIT orders (continued)

<table>
<thead>
<tr>
<th>EDIT Order</th>
<th>PIPE</th>
<th>MRT REVISE and WHEN</th>
<th>CRT REVISE and WHEN</th>
<th>Auto Edit and Acquire</th>
<th>CBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROADCAST</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C2B</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C2D</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C2F</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C2G</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C2GV</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C2P</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C2S</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C2V</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C2VG</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C2X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CBETEMP</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHKEY</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMDX</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNVDT</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CNVDT0</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>COLOR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CONSAUTH</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONSNAME</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONSZERO</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPY</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>COPYREST</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>COPYREV</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CURRGMT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CZID</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>D2C</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>D2X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>D4NV</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESC</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DISPLAY</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DTS</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ETIME</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>F2C</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FINDLINE</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>flag_bytes</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLGDSCDn</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 7. Supported environments for EDIT orders (continued)

<table>
<thead>
<tr>
<th>EDIT Order</th>
<th>PIPE</th>
<th>MRT REVISE and WHEN</th>
<th>CRT REVISE and WHEN</th>
<th>Auto Edit and Acquire</th>
<th>CBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLGRTCDT</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOUND</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FWDLINE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G2C</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GV2C</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HDRM_TYPE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUCON</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUCPY</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUGMT</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUHND</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUIN3</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUMTB</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUNEX</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUPPT</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUPRI</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUSD</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUSECF</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUOK</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUWAS</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUWDS</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUWJA</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUWJU</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFRAUWR</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOBNAME</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>JOBNUM</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LASTLINE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEFT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LEVEL</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEATTR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINECOUNT</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEORIGIN</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINESENDER</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINETYPE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGTO</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MCS_FLAGS</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRT</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSGCOUNT</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 7. Supported environments for EDIT orders (continued)

<table>
<thead>
<tr>
<th>EDIT Order</th>
<th>PIPE</th>
<th>MRT REVISE and WHEN</th>
<th>CRT REVISE and WHEN</th>
<th>Auto Edit and Acquire</th>
<th>CBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSGID</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MSGORIGIN</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSGSEND</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MSUSEG</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>NAMEBIND</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>NEXT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>NEXTPAG</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEXTWORD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>NOEXPOSE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOLOGTO</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTECHO</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVABLE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ODDBYTES</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ONTO</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OPDT</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>P2C</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PARSE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>position</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PREFIX</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROG</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READCBE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READLINE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READSRC</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESET</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>RESETAUTO</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RIGHT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ROUTECODES</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ROUTEZRO</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RVAR</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SESSID</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SETACTION</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SETAUTO</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETBEEP</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SETCLEAR</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETGMT</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SKIPTO</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>STRIP</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Table 7. Supported environments for EDIT orders (continued)

<table>
<thead>
<tr>
<th>EDIT Order</th>
<th>PIPE</th>
<th>MRT REVISE and WHEN</th>
<th>CRT REVISE and WHEN</th>
<th>Auto Edit and Acquire</th>
<th>CBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRIPL</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>STRIPR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SUBSTR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SYSCONID</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SYSLOG</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SYSNAME</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOKEN</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>TOPLINE</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UCHARS</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UFLAGS</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UPCASE</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UPTO</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>V2C</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>VG2C</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>W1, W2, ... W9</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>WQE</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>WRITELINE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>WTOKEY</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>X2C</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>YESNO</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ZDT</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</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

Usage Notes

- Code one edit phrase or one global order on each source line because edit scripts consisting of many edit phrases can be difficult to read. Together with appropriate commentary, your edit script is easy to understand. See sample CNME2011 (SESMGET) for an example of this type of coding.
**PIPE EDIT**

- 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 statements convert 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 statements convert the time portion of the GMT:

```
PIPE ...|EDIT IFRAUGMT ZDT CNVDT
         ('HH:MM:SS' TIME) NEXT|...
```

**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 in the range of 0 - 9 or a character in the range of 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 defaults *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:

```
PIPED CAN BE FUN!
```

**This... Results in...**

```
1.* PIPES CAN BE FUN!
+7.6 CAN BE
```
Global Orders

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

**CBETEMP (global order)**

CBETEMP is used to read in a Common Base Event template. The CBETEMP order reads in the template as a secondary chain of NetView buffers. The READCBE and READSRC global orders can be used to switch the primary input source for the EDIT phrase. This order is primarily intended for use by the EDIT automation table action in order to build a Common Base Event from a message or from an MSU.

The following parameter is required:

*template_name*

The name of the Common Base Event template, corresponding to the `name` attribute on a `<cbedata>` tag in the template file.

A simple phrase to create a CBE from a message is:

```
CBETEMP /msgdefault/ COPY *
```

If the template name cannot be found, message BNH883I is returned.

**CONSZERO (global order)**

Sets the 4-byte console ID to zero.

**COPY (global order)**

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 command stream:
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 for more information.

COPYREST (global order)
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 process for all input messages minus one:
READLINE
1.* 1
WRITELINE
COPYREST
/* copy lines 2 through end to output */
WRITELINE
/* write first line */

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 process:
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 WRITELINE for more information.

COPYREV (global order)
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.

FINDLINE n (global order)
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/ (global order)
Advances the current line forward to a line containing the specified target string. If no line after the current line contains the target, a null line is used for subsequent input orders.

FWDLINE n (global order)
Moves the current line ahead by the number specified.

LASTLINE (global order)
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.

**LOGTO (global order)**
Sets an option such that if the message is presented for display using either the CONSOLE stage or the ROUTE stage, the message is logged. One of these arguments is required:
- N: This action is for the net log.
- S: This action is for the system log.
- H: This action is for the hardcopy log.
- *: This action is for all three log methods: net log, system log, and hardcopy log.

**Usage:**
1. If the message is automated, then automation can override the logging options set by EDIT. If the message is delivered to a distributed autotask, then the log options are honored on the local domain, but is reset to default values by RMTCMD message forwarding. Also, if a message is routed directly to a remote domain, the options are reset to default values.
2. Messages that are exposed for logging are always written to the CanzLog file. You can control visibility of your messages in the CanzLog file by using the TAG keyword of the LOGTO stage.

**NAMEBIND (global order)**
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 halfword length value. The message ID (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 113, CNMEALUS, and CNMEMSUS. For more information about PPI, see "PIPE PPI" on page 168.

To create a valid adapter name/value pair binding, do one of these items:

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

**Note:** This is required for alert automation. Alerts cannot 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.

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

**NEXTLINE** *(global order)*
This specifies both the READLINE and WRITELINE keywords, and advances both the input message line (source) and the output message line (destination).

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

**NOLOGTO** *(global order)*
Sets an option such that if the message is presented for display using either the CONSOLE stage or the ROUTE stage, the message is not logged. One of these arguments is required:
- **N** This action is for the net log.
- **S** This action is for the system log.
- **H** This action is for the hardcopy log.
- ***** This action is for all three log methods: net log, system log, and hardcopy log.

**Usage:**
1. If the message is automated, then automation can override the logging options set by EDIT. If the message is delivered to a distributed autotask, then the log options are honored on the local domain, but is reset to default values by RMTCMD message forwarding. Also, if a message is routed directly to a remote domain, the options are reset to default values.
2. Messages that are exposed for logging are always written to the Canzlog file. You can control visibility of your messages in the Canzlog file by using the TAG keyword of the LOGTO stage.

**NOTECHO** *(global order)*
Turns off the echo flag (WQEMCSM) in the message that is logged or automated. NOTECHO does not affect the display.

**ONTO** *(global order)*
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.

**PAD** *(global order)*
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** *(global order)*
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 quotation marks are counted as words regardless of embedded blanks. The single quotation marks are
removed from the parsed words. If the input data contains unbalanced quotation marks, only the data to the point where the error was discovered is returned.

/string/

Specifies how subsequent work/token orders counts 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:

'PIPES ARE' 'REALLY' 'FUN'

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

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

For the command revision table environment, the default PARSE value is /,=() where the string consists of the following characters:

For all other environments, the default PARSE value is C.

READCBE (global order)

READCBE is used to switch the primary input to a multiline template message initialized by the CBETEMP order. If the CBETEMP order has not been used, no input is available for future edit orders to process. This order is used with READSRC to switch input sources. Using the two orders, EDIT phrases can be constructed to output part of a template, switch to the input message to fill in data such as message text, and then switched back to the CBETEMP input to construct the rest of the output.

READLINE (global order)

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 is similar to the following output:

```
IEE114I 11.44.14 96.141 ACTIVITY 444
JOBS M/S TS USER SYSAS INIT/SA MAX VNAM OAS
00000 00006 00001 00016 00002 00001/00300 00000
```

The following edit script builds 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.

READSRC (global order)
READSRC is used to switch the primary input to the original message or msu on the EDIT stage’s input stream. This order is used in conjunction with READCBE to switch input sources. Using the two orders, EDIT phrases can be constructed to output part of a template, switch to the input message to fill in data such as message text, and then switched back to the CBETEMP input to construct the rest of the output.

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

RESETAUTO (global order)
Sets the IFRAUHND, IFRAUMTB, and IFRAUNEX automation flags to zero (‘0’B). Take care not to create automation loops when using this function.

ROUTEZERO (global order)
Overrides any previous specifications of route code for the message being revised.

SETACTION (global order)
Sets the IFRAUACN and IFRAUNVD automation flags. This causes the message to be held on an operator’s screen unless a setting by the DEFAULTS or OVERRIDE command prevents it. See also the BULLETIN option of the PIPE ROUTE stage command in the NetView online help or in IBM Tivoli NetView for z/OS Programming: Pipes.

SETAUTO (global order)
Sets the IFRAUMTB automation flag to one (‘1’B). Subsequent routing of the message within a domain does not result in a submission to the automation table.

SETBEEP (global order)
Sets an audible alarm for message output. When the message output is displayed, the audible alarm is issued.

SETCLEAR (global order)
Sets the IFRAUCLR automation flag. This causes the NetView command facility screen to be erased just before presentation of the message containing this flag. Use with care to avoid losing important operator messages.

SKIPTO (global order)
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:

PIPS ARE FUN. PIPES USING EDIT ARE EVEN BETTER!

And, the edit script is:
The line is processed as follows:

And, the output is:

And, the output is:

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:

And, the edit script is:

The line is processed as follows:

And, the output is:

SING

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 is 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 and RESET.

TOPLINE (global order)

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.
**PIPE EDIT**

**UPTO (global order)**

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:

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

And, the edit script is:

```
UPTO /FUN/
WORD -1
NEXT
```

The line is processed as follows:

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

```
UPTO /FUN/  
WORD -1  
```

And, the output is:

```
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:

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

And, the edit script is:

```
UPTO 21
WORD -1
NEXT
```

The line is processed as follows:

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

```
UPTO 21  
WORD -1  
```

And, the output is:

```
PIPS
```

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 is 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 and RESET.

**WRITELINE (global order)**

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 is 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 (input order)

Specifies that the input is the 256-byte AIFR body. For additional information about the AIFR fields, see IBM Tivoli NetView for z/OS Programming: 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 IBM Tivoli NetView for z/OS Programming: Assembler plus one (1).

ALL

Inputs the text in the current line. Same as 1.*.

ASTYPE (input order)

Indicates how the address space was started (job type):

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>USS persistent procedure. The address space has a name for initiated programs, appropriate for a JOB. However, the existence of an OpenMVS address space block indicates a special purpose USS persistent procedure.</td>
</tr>
<tr>
<td>E</td>
<td>The address space is a system address space that is started during operating system initialization (NIP) processing, and the message was issued before NetView subsystem initialization.</td>
</tr>
<tr>
<td>J</td>
<td>The address space is a JOB.</td>
</tr>
<tr>
<td>N</td>
<td>The address space is a system address space started during operating system initialization (NIP) processing.</td>
</tr>
<tr>
<td>S</td>
<td>The address space is a Started Task (STC).</td>
</tr>
</tbody>
</table>

Note: Because of the manner in which TN3270 is started, it may appear as type S rather than type D, as might be expected.

T     | The address space is a Time-Sharing User (TSO). |
U     | The address space is a USS forked or spawned procedure. |
*     | Error: the address space where the command originated has closed or else the message is not from the local LPAR. |
?     | Error: inconsistent data (might be a transient condition). |
!     | Error: inconsistent data. |
Error: the supplied ASID is larger than the allowed ASID limit for the system

**CHKEY (input order)**
Obtains the CHKEY as defined by system macro IEECHAIN. This is the step-name of a task or the job-name of a job.

**CMDX (input order)**
Inputs the first 88 (X'58') bytes of the IEZVX101 control block.

**COLOR (input order)**
Inputs the characters describing the attributes of the current line, including color, highlighting, line type, and intensity. See "Output Orders" on page 109 for the description of text values. This is an example:

```
CY HU IN TD
```

**CONSAUTH (input order)**
Indicates the authority of the console issuing the command:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>Master</td>
</tr>
<tr>
<td>I</td>
<td>I/O</td>
</tr>
<tr>
<td>S</td>
<td>System</td>
</tr>
<tr>
<td>C</td>
<td>Console</td>
</tr>
<tr>
<td>X</td>
<td>Information is not available, or conflicting data is found.</td>
</tr>
</tbody>
</table>

**CONSNAME (input order)**
Obtains the console name:
- For the CRT, returns the name of the console issuing the command.
- For the MRT, returns the destination console name.

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

**CZID (input order)**
Obtains the Canzlog ID of a message or command echo or DOM that has been logged. It returns 0 if the message has not been logged.

**D4NV (input order)**
D4NV (destined for the NetView program) is used with a WHEN or REVISE statement. This input-only edit order indicates whether the console name to which a message is to be delivered is owned by a NetView task. You can reenable system logging for a particular message, if desired, by using the SYSLOG order.

**DELETE (input order)**
Determines whether a message has been marked for deletion. If the message has been marked for deletion, the DELETE input order supplies a non-null output; if the message has not been marked for deletion, the output is a null character. This output can be used as input to other decision-making orders such as the YESNO order.

The DELETE input order is only available for message revision.

**disposition (input order)**
When used as input orders, these orders provide information about the disposition of the message. These edit orders produce a binary representation of the value (X'00' or X'80'), which can be converted to a Yes or No value using the YESNO conversion. The disposition orders are as follows:
AMRF
Returns the following status: message is to be retained in AMRF

AUTOMATE
Indicates that the message is to be automated.

BROADCAST
Indicates that the message is to be sent to all active consoles

DISPLAY
Indicates that the message is to be displayed at the console.

PROG
Displays programming information

SYSLOG
Writes to the system log

flag_bytes (input order)
Used with routing and descriptor codes and represents an 8-bit section of the field. When used as an input order, it produces a string of 8 characters, consisting of 0s and 1s. These correspond to the bit values in the byte that is being referenced.

FLGDSCon (input order)
Indicates that the specified descriptor code byte is to be read, where n is 1, 2, 3, or 4. The value of the byte is supplied as a string of eight EBCDIC values (either 0 or 1).

FLGRTCDn (input order)
Indicates that the specified route code byte is to be read, where n is 1 to 16. The value of the byte is supplied as a string of eight EBCDIC values (either 0 or 1).

HDRMTYPE (input order)
Indicates that the one-byte message type indicator is to be read in the message line that is being processed.

IFRAUHND (input order)
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.

IFRAUIN3 (input order)
Indicates that the 32 bits that are defined as IFRAUIN3 are used as input. For these 32 flags, see the DSIIFR assembler macro.

IFRAUMTB (input order)
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 (input order)
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.

LEVEL (input order)
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 (+).
PIPE EDIT

For example, if the following data sets are concatenated in this order under the DSIPARM DDNAME:

USER.INIT
USER2.INIT
NMPTLS.INIT
BNVE330E.PROCEED.DSIPARM
NETV.E120E.PROCEED.DSIPARM
NETV.E120E.PROCEED.CNMSAMP

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

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

`lineattr (input order)`
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 `lineattr` specifies attributes of the current line being processed by the edit phrase.

The `lineattr` attribute can be one of the following values:

**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 230, and "PIPE STEM and PIPE $STEM" on page 202 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 value of the HDRDOMID field for the message buffer being examined. The HDRDOMID field usually contains the domain ID where the line originated, but it might contain some other value. For example, when the From Disk (<) stage command is used, the HDRDOMID field contains the member name read from disk. The value can differ for different lines in the same message. For example, if the COLLECT stage 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 The current line is a control line.
TL The current line is a label line.
TD The current line is a data line.
TE The 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.

For example, the output from the following specification:

```
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.

**MCSFLAGS** *(input order)*

The 16-bit MVS multiple console support flag.

Check these bits:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>The message is to be queued to the console if it is active</td>
</tr>
<tr>
<td>3</td>
<td>The message is a command response WTO</td>
</tr>
<tr>
<td>5</td>
<td>The message is a reply to a WTO</td>
</tr>
<tr>
<td>6</td>
<td>The message is to be broadcast to all active consoles</td>
</tr>
<tr>
<td>7</td>
<td>The message is to be queued to hardcopy only</td>
</tr>
<tr>
<td>8</td>
<td>The message is to be queued unconditionally to the console</td>
</tr>
<tr>
<td>9</td>
<td>The message is not to be time-stamped</td>
</tr>
<tr>
<td>14</td>
<td>The message is not to be queued to hardcopy</td>
</tr>
</tbody>
</table>

**Length:** 16 bits

**Type:** Message

**MRT** *(input order)*

Indicates that the message has been exposed to the Message Revision Table. If this flag is on when a message is output with a WTO command, then the message is not subject to further processing by the MRT unless overridden. See the NetView online help or the *IBM Tivoli NetView for z/OS Command Reference Volume 2 (O-Z)* for information about the WTO command.

**msgattr** *(input order)*

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", see the assembler mapping of DSIIFR and the *IBM Tivoli NetView for z/OS Automation Guide*.

**msgattr** can be one of the following specifications:

**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.
AUTHGRP
Specifies the ACEE group ID (ACEEGRPN), if available. If it is not available, returns *UNKNWN*.

The values that are supplied for AUTHUSER and AUTHGROUP input orders are derived from the ACEE that is active at the time the message is routed through the z/OS subsystem interface. Certain authorized programs issue messages in an environment that causes them to be routed asynchronously through the CONSOLE address space. These messages show "+CONSOLE" as the AUTHUSER value and a single asterisk (*) as the AUTHGROUP value. Messages that are issued from the master scheduler show "+MASTER+" for the AUTHUSER value, instead. Only authorized programs can issue messages in this way.

AUTHUSER
Specifies the z/OS ACEE user ID (ACEEUSRI), if available. If it is not available, returns *UNKNWN*.

The values that are supplied for AUTHUSER and AUTHGROUP input orders are derived from the ACEE that is active at the time the message is routed through the z/OS subsystem interface. Certain authorized programs issue messages in an environment that causes them to be routed asynchronously through the CONSOLE address space. These messages show "+CONSOLE" as the AUTHUSER value and a single asterisk (*) as the AUTHGROUP value. Messages that are issued from the master scheduler show "+MASTER+" for the AUTHUSER value, instead. Only authorized programs can issue messages in this way.

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 the NetView program.

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.

IFRAUIN3
Indicates that the 32 bits that are defined as IFRAUIN3 are used as input. For these 32 flags, see the DSIIFR assembler macro.
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.

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 is 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 is 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. Only use MSGCOUNT if a stage preceding EDIT is COUNT EACHMSG. Any other source for MSGCOUNT yields unpredictable results. See “PIPE COUNT” on page 54 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.

MSGID
The message identifier of the received message. MSGID is a character ID of up to 12 characters. The message identifier is usually the first token of the message. If a REPLYID is sent with the message, the REPLYID is not used as the first token.

MSGORIGIN
Use as input the 8-character domain ID where the message originated.
MSGSEND
Indicates that the edit phrase input is the 8-character name (OPID) of the task that created the message or, if the message was routed between tasks, that most recently sent the message.

MSUSEG
Indicates the contents of one segment of an MSU. The compare items can be a bit string or a parse template.

Use any of these choices to specify the location of the data to be compared:

H For an MDS-MU, indicates that the first key is to be obtained at the MDS-MU level, rather than the major-vector level. If you use this parameter and the MSU being processed is not an MDS-MU, MSUSEG returns a value of null.

key The 2- or 4-character representation of the 1- or 2-byte hexadecimal ID of the generalized data stream (GDS) variable or key of the major vector, subvector, subfield, or sub-subfield.

You can specify multiple key values, separating them with periods. Each additional key specifies a lower-level structure within the structure identified by the preceding key.

occurnum The occurrence number, counting from 1, of the generalized data stream (GDS) variable, major vector, subvector, subfield, or sub-subfield. An asterisk (*) indicates that you want any occurrence. For example, used at the subvector level, an occurnum value of 2 means that you want the second instance of the key subvector. An occurnum value of * means that you want the first subvector with a key of key, if any, that results in equality with the compare item that you specified. The maximum occurnum value is 32,767. The default value is 1.

MSGSEND (input order)
Indicates that the edit phrase input is the 8-character name (OPID) of the task that created the message or, if the message was routed between tasks, that most recently sent the message.

NVABLE (input order)
Returns "Yes" if a NETVONLY action can succeed, otherwise returns "No".

A NETVONLY action cannot succeed if the NetView procedural space is down, the subsystem router is down, or if the task defined by the ?MVSCmdRevision CNMSTYLE statement is inactive.

REPLYL (input order)
Returns the length of the reply ID in decimal format, with a leading plus sign (+).

ROUTECODES (input order)
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 IFRAUWR.

SESSID (input order)
Specifies that the edit phrase input is the TAF session ID or, following a PPI pipe receive stage, is the SAF ID of the PPI sender.
**SYSCONID** *(input order)*
Use as input the 8-character MVS System Console name. If the message was not received from MVS, blanks are returned.

SYSCONID is synonymous with IFRAUCON.

**SYSNAME** *(input order)*
Use as input the 8-character name of the system from which the message originated. If the message was issued locally, the name of the local system is returned. If the message is a remote message (one which originated on another system in a sysplex), the name returned is the remote system name, which is different from the local system name. You can compare the value returned with the &SYSNAME. system symbolic to determine whether the message is local or remote.

**UCHARS** *(input order)*
Obtains the 16-byte "user char" area. In the MRT, this field is available only if previously set. In PIPE EDIT, this field is equivalent to IFRAUSRC.

**UFLAGS** *(input order)*
Obtains the 2-byte "user flags" area. In the MRT, this field is available only if previously set.

**WORD** *(input order)*
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 ... Results in ...

**WORD 1.*

PIPES CAN BE FUN!**

**WORD 2.2**

CAN BE

**WORD -2.*

BE FUN!**
PIPE EDIT

WORD 2.3
CAN BE FUN!

WORD 2
CAN

WORD -25.5
a null string is returned

WORD -6.3
PIVES

WQE (input order)
Enables the WQE for conversion. Always use the SUBSTR order following
WQE and determine the positions needed by consulting mapping IHAWQE.
The SUBSTR order uses a position.length (starting with one) and not an offset
(starting with zero). The maximum allowable character string length for
revision orders is 127.

For example, the WQEMCSC command response flag is not typically accessible
with an edit order. From a listing, we can determine that WQEMCSC is at
offset 'XAC' (decimal 172) in the WQE and it is the third bit in that byte.
Therefore, the edit phrase WQE SUBSTR 173.1 c2b substr 3.1 yields either a
character 1 or a character 0, according to the value of the WQEMCSC flag.

WTOKEY (input order)
Obtains the key field associated with the WTO system macro, which is the
WQEKEY in system macro IHAWQE.

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.

ASTYPE (conversion order)
Specifies that the input contains a two-byte binary ASID value. A one-character
value is returned as indicated in Table 8.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>USS persistent procedure. The address space has a name for initiated programs, appropriate for a JOB. However, the existence of an OpenMVS address space block indicates a special purpose USS persistent procedure.</td>
</tr>
<tr>
<td>E</td>
<td>The address space is a system address space that is started during operating system initialization (NIP) processing, and the message was issued before NetView subsystem initialization.</td>
</tr>
<tr>
<td>J</td>
<td>The address space is a JOB.</td>
</tr>
<tr>
<td>N</td>
<td>The address space is a system address space started during operating system initialization (NIP) processing.</td>
</tr>
<tr>
<td>S</td>
<td>The address space is a Started Task (STC).</td>
</tr>
</tbody>
</table>

Table 8. Returned Value from ASTYPE
Table 8. Returned Value from ASTYPE (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>The address space is a Time-Sharing User (TSO).</td>
</tr>
<tr>
<td>U</td>
<td>The address space is a USS forked or spawned procedure.</td>
</tr>
<tr>
<td>&gt;</td>
<td>Error: the supplied ASID is larger than the allowed ASID limit for the system</td>
</tr>
<tr>
<td>*</td>
<td>Error: the supplied ASID is not currently assigned (no such address space).</td>
</tr>
<tr>
<td>?</td>
<td>Error: inconsistent data (might be a transient condition).</td>
</tr>
<tr>
<td>!</td>
<td>Error: inconsistent data.</td>
</tr>
</tbody>
</table>

**B2C (conversion order)**

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 is 11000011000010 B2C returns 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 (conversion order)**

Specifies that the input data is 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 is AB, C2B returns 11000011000010.

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. For more information, see the conversion order for **SUBSTR**.

C2B is the inverse of B2C.

**C2D (conversion order)**

Specifies that the input data is treated as a 2's complement binary number. This input data is then converted into a positive or a negative decimal number. For example, if the input is 1, C2D returns a result of -15. If the input is AB, C2D returns 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:

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

Use C2D 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 (conversion order)**

Specifies that the input data is converted to a displayable floating point
notation. The input can be a 2- to 8-byte floating point number. The converted value is a 22-byte, right-aligned, 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 conversion order for \(\text{F2C}\) and the conversion order for \(\text{C2F}\).

**C2G** *(conversion order)*

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.

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

**C2GV or C2VG** *(conversion order)*

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 because 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.

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

**C2P** *(conversion order)*

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

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

C2P is the inverse of P2C.

**C2S** *(conversion order)*

Specifies that the input data is converted to a displayable floating point notation. The input can be a 2- to 8-byte floating point number. The converted value is a 14-byte, right-aligned, output string in the form \(-n.mmmmmE-dd\) 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 converted to 2.

See also the conversion order for \(\text{F2C}\) and the conversion order for \(\text{C2F}\).
**C2V** *(conversion order)*

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.

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

**C2X** *(conversion order)*

Specifies that the input data is converted to hexadecimal notation. For example, if the input is AB, C2X returns the EBCDIC hexadecimal value ‘X’C1C2’. C2X is particularly useful when you must display input data containing nondisplayable characters.

C2X is the inverse of X2C.

**CHKEY**

Obtains the CHKEY as defined by system macro IEECHAIN. This is the step-name of a task or the job-name of a job.

**CNVDT** *(conversion order)*

Specifies that the input data is a date or time value and is 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** *(conversion order)*

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

**D2C** *(conversion order)*

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. For more information, see the conversion order for SUBSTR.

D2C is the inverse of C2D.

**D2X (conversion order)**

Specifies that a decimal number is to be converted to hexadecimal representation.

**DT (conversion order)**

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.

**Note:** The current GMT offset is used in interpreting the local date and time, whether a different offset was in effect at the given date and time. For example, if the given value was before the latest daylight saving time adjustment, the result can be off one hour from another interpretation of the same date and time of an application.

**DTS (conversion order)**

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.

**Note:** The current GMT offset is used in interpreting the local date and time, whether a different offset was in effect at the given date and time. For example, if the given value was before the latest daylight saving time adjustment, the result can be off one hour from another interpretation of the same date and time of an application.

**ETIME (conversion order)**

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 can be longer than 10 digits. The result can also be a negative number indicating that the message originated before NetView startup.

**FOUND (conversion order)**

FOUND is used after a SKIPTO or FINDLINE operation to translate a null string into No and any other string into Yes. The case of the character string is exactly as displayed, No or Yes.

**F2C (conversion order)**

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 (conversion order)**

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.

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

**GV2C or VG2C (conversion order)**
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.

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

**JOBNAME (conversion order)**
Specifies that the input contains a 2-byte binary ASID value. The corresponding job name is returned. If an error occurs, one of the four error tokens (>,*?, or !) in Table 8 on page 102 might be returned.

**LEFT (conversion order)**
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 or order is used.

**ODDBYTES (conversion order)**
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 (conversion order)**
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 DEFU LTS or OVERRIDE date and time formats for the system and operator where the conversion is done.

For example, after the command OVERRIDE LONGDATE=YY.MM.DD, the value X'CA0A68B6F34C3820' is converted to 13.01.31 10:28:11.

A typical use is 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 (conversion order)**
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 has `scale` number digits included in the packed decimal. For example, \( \text{C2P 2} \) converts the input 123.456 to \( \text{X'12345C'} \) which is the packed decimal representing 12345.

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

\( \text{C2P} \) is the inverse of \( \text{P2C} \).

**PREFIX** *(conversion order)*

Adds a constant to the beginning of a string.

**RIGHT** *(conversion order)*

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

**RVAR** *(conversion order)*

From input revision variable name, returns the current value.

**STRIP** *(conversion order)*

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 \( \text{PAD} \) global order specification within the edit phrase.

**STRIPL** *(conversion order)*

Specifies that padding characters at the beginning of the data are to be removed. The padding character is defined by the most current \( \text{PAD} \) global order specification within the edit phrase.

**STRIPR** *(conversion order)*

Specifies that any padding characters at the end of the data are to be removed. The padding character is defined by the most current \( \text{PAD} \) global order specification within the edit phrase.

**SUBSTR** *(conversion order)*

Specifies that a subset of the input data is to be selected. \( \text{Position.length} \) indicates the starting position and length of data to be selected. For information about defining \( \text{position.length} \), see \( \text{position.length} \).

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

**UPCASE** *(conversion order)*

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

**V2C** *(conversion order)*

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.

\( \text{V2C} \) is the inverse of \( \text{C2V} \).

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

**X2C** *(conversion order)*

Specifies that the input data is converted from displayable hexadecimal notation to internal binary representation. For example, if the input is \( \text{X'C1C2'} \), \( \text{C2X} \) returns the hexadecimal values \( \text{X'AB'} \). The resulting hexadecimal value is half the length of the original.
X2C is the inverse of C2X.

**YESNO (conversion order)**

Specifies that the first byte of the input data is 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. The case of the character string is exactly as displayed, Yes or No.

This conversion order is particularly useful when using the IFRAUCPY, IFRAUPPT, IFRAUPRI, and IFRAUSEC msgattr input orders.

**ZDT (conversion order)**

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.

**AUTOTOKEN (output order)**

Sets the 8-character automation token in the message.

AUTOTOKEN is synonymous with IFRAUTOK.

**COLOR (output order)**

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**

<table>
<thead>
<tr>
<th>Code</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>CB</td>
<td>Blue</td>
</tr>
<tr>
<td>CR</td>
<td>Red</td>
</tr>
<tr>
<td>CP</td>
<td>Pink</td>
</tr>
<tr>
<td>CY</td>
<td>Yellow</td>
</tr>
<tr>
<td>CG</td>
<td>Green</td>
</tr>
<tr>
<td>CW</td>
<td>White</td>
</tr>
<tr>
<td>CT</td>
<td>Turquoise</td>
</tr>
<tr>
<td>CD</td>
<td>Default</td>
</tr>
</tbody>
</table>

**Intensity**

<table>
<thead>
<tr>
<th>Code</th>
<th>Intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Not intensified</td>
</tr>
<tr>
<td>IH</td>
<td>Intensified</td>
</tr>
<tr>
<td>ID</td>
<td>Output line is dark. Although the output line exists, it is not displayed.</td>
</tr>
</tbody>
</table>

**Highlighting**

<table>
<thead>
<tr>
<th>Code</th>
<th>Highlighting</th>
</tr>
</thead>
<tbody>
<tr>
<td>HR</td>
<td>Reverse video</td>
</tr>
<tr>
<td>HU</td>
<td>Underlined</td>
</tr>
<tr>
<td>HB</td>
<td>Blinking</td>
</tr>
</tbody>
</table>
For example, if you want to create a blue, flashing data line of normal intensity, specify:

/\CB IN HB TD/ COLOR

COLOUR, LINEATTR, and LINEATRTRS are synonyms for COLOR.

**CONSNAME** *(output order)*
Sets the console name.

**DELETE** *(output order)*
Deletes a message or command, or undeletes a previously deleted message or command, depending on the input:

- If the input is null, the letter N, or the number 0, the DELETE output order undoes a previous deletion.
- If the input is a non-null value other than the letter N or the number 0, the DELETE output order deletes the message or command. Marking a message for deletion does not prevent the message from being automated.

The DELETE output order is available only when performing a REVISE action.

**disposition** *(output order)*
When used as output orders, these orders control or change the disposition, subject to certain system constraints. These edit orders write null strings and the 0, n, and N strings to the WQE as a 0. Any other string writes a 1 to the WQE indicating the function is enabled. Following are the disposition orders:

- **AMRF**
  Retains the message in AMRF

- **AUTOMATE**
  Allows automation. Note that setting AUTOMATE to an affirmative value (such as Y) causes a foreign unsolicited message to be automated.

- **BROADCAST**
  Sets an indicator that causes the operating system to send messages to all active consoles.

- **DISPLAY**
  Sets an indicator that the message is displayable at the console.

- **PROG**
  Displays programming information

- **SYSLOG**
  Writes to the system log

**FINDLINE** *(output order)*
Functions the same as global order FINDLINE n except that the target to be found is derived from the previous input order or conversion order. If the number is negative, EDIT counts from the end of the message.

**flag_bytes** *(output order)*
Used with routing and descriptor codes and represents an 8-bit section of the
field. When used as an output order, it requires a string of 8 characters, consisting of 0s, 1s, and Xs. These correspond to the requirement to clear, set, or leave as-is the corresponding bit in the byte that is being referenced.

The following example indicates that the message is to be sent to route codes 1 and 2 but not to route code 5, and that the other route codes are to be left as is:

REVISE('11xx0xxx' FLGRTCD1) ! Send to rt cd 1 and 2, not 5, leave others as is.

**FLGSDCd** *(output order)*

Indicates that the specified descriptor code byte is to be written, where $n$ is 1, 2, 3, or 4. The value of the byte is changed according to the input mask. The input mask must be a string of eight EBCDIC values (0, 1, or x). Each descriptor code flag bit that corresponds to a 0 is turned off; each descriptor code flag bit that corresponds to a 1 is turned on; and the other bits are left as is.

**FLGRTCDn** *(output order)*

Indicates that the specified route code byte is to be written, where $n$ is 1 to 16. The value of the byte is changed according to the input mask. The input mask must be a string of eight EBCDIC values (0, 1, or x). Each route code flag bit that corresponds to a 0 is turned off; each route code flag bit that corresponds to a 1 is turned on; and the other bits are left as is.

**HDRMTYPE** *(output order)*

Indicates that the one-byte message type indicator is to be written in the message line that is being processed.

**LINETYPE** *(output order)*

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 values:

- **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.
- **TE** 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.

**MRT** *(output order)*

Indicates that the message has been exposed to the Message Revision Table.

**NEXT** *(output order)*

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** *(output order)*

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** *(output order)*

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 text on the output stream:

MANIPULATING MESSAGES CAN BE FUN WITH EDIT!

**SETGMT** (output order)

SETGMT sets the IFRAUGMT value of the output message. The order is carried out only if the input available is exactly eight bytes. You can use this order with the CURRGMT order if you want to reinsert a message into the Canzlog log.

**UCHARS** (output order)

Sets a 16-byte "user char" area. In PIPE EDIT, UCHARS is equivalent to the previously-existing IFRAUSRC, except that UCHARS accepts a value shorter than 16 characters (no padding occurs) or truncates a value longer than 16 characters.

**UFLAGS** (output order)

Sets a 2-byte "user flags" area. In the MRT, this field accepts a string of up to 16 characters consisting of 0s, 1s, and Xs. These correspond to the requirement to clear, set, or leave as-is the corresponding bit in the byte being referenced. In PIPE EDIT, UFLAGS is equivalent to the previously-existing IFRAUSRB, except that UFLAGS accepts a value shorter than 2 characters (no padding occurs) or truncates a value longer than 2 characters.
WTOKEY

For the Message Revision Table (MRT), WTOKEY sets the key field associated with the WTO system macro, which is the WQEKEY in system macro IHAWQE.

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 DSIO01I 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=/
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 is:

```
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 descriptions of WORD and SKIPTO for more information.

Example: Sending an Alert to the NetView Alert Adapter

The following example 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.
/* construct a value from MYVAR and a unique identifier */
/* (GMT value set when alert received). Note: convert GMT to */
/* displayable chars */
'PIPE (NAME TECBIND)',
' | SAFE *',  /* copy complete automation alert into pipeline */
' | EDIT',  /* begin edit */
' | COPY *',  /* copy complete automation alert to EDIT output*/
' | /myvar/*',  /* start value one: variable value */
' | IFRAUGMT C2X NEXTWORD',  /* add EBCDIC hex value */
' | NAMEBIND /EVENTID/',  /*create output line for TEC slot */
' | /MSUSEG('0000.31.30',3)' /',  /*start 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 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. */

---

**PIPE ENVDATA**

**Syntax**

```
ENVDATA:
```

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENVDATA</td>
<td>ENV</td>
</tr>
</tbody>
</table>

**Command Description**

The ENVDATA stage outputs environment data, which consists of a multiline message in the following format:

```
SCREEN DEPTH mnn
SCREEN WIDTH mnn
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 items:

- 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 can 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.
- The numbers in output do not include leading zeros.

Example: Capturing Environment Data

The following example captures environment data in variables inside a REXX procedure.

Coded as a REXX example:

```rexx
'PIPE ENVDATA',
' NOT CHOP 13',
' VAR ROWS COLS'
SAY ROWS
SAY COLS
```

```plaintext
---> 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. PLCCMDMOD was the invoker of PIPE ENVDATA.

PIPE EXPOSE

Syntax

EXPOSE:
PIPE EXPOSE

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 is 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 are 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 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 is not exposed to the same interfaces again. The default value is RESPECT.
TOTRAP
Specifies that messages are only exposed to TRAP processing. TOTRAP implies NOLOG.

Usage Notes
- EXPOSE cannot be a first stage.
- 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 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.
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 result in TOMMY being 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 are input to FANIN and subsequently placed in the NAMES stem.

Finally, all other names that are not SMITH or BAKER are processed by FANIN and placed in the NAMES stem.

Unlike the FANINANY example, under "Example: Process a List of Names" on page 119, this example uses FANIN.

Note: The double connector C:B: is used to connect the secondary output of LOCATE /BAKER/ to the tertiary input of FANIN.

PIPE FANIN

Syntax

FANINANY:

|——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>
</tbody>
</table>
### Termination Conditions

FANINANY terminates when all input streams disconnect or the output stream disconnects.

### Operand Descriptions

This stage 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 are 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.

### 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 result in TOMMY being 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 118, in this FANINANY example, the names in the NAMES. stem are in the same order as the records read from the data set.

```plaintext
PIPE (END ~) < /NAMES.
|A: LOCATE /SMITH/ |
|CHANGE /TOMMY/TOM/ |
|B: FANINANY |
|STEM NAMES. |
~A: |
|C: LOCATE /BAKER/ |
|CHOP 22 |
|B: |
~C: |
|B: |

**Note:** The double connector C:B: is used to connect the secondary output of LOCATE /BAKER/ to the tertiary input of FANINANY.
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, see IBM Tivoli NetView for z/OS Programming: 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 ends when the input stream disconnects or all output streams disconnect.

Operand Descriptions

This stage has no operands.

Usage Notes

- The messages are not written to the output streams in any predetermined order. Streams waiting on input from FANOUT can 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. Pass the primary output stream from the second FANOUT stage to a HOLE stage because the primary output from the second FANOUT stage does not have a copy attribute.

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, see sample CNMS4275 and IBM Tivoli NetView for z/OS Programming: Assembler.

Additional commands can be driven off the same message by adding commands in stages following additional \( \text{SEQLOG} \): connectors.

\[
\text{PIPE (NAME COPYNLOG END ~)}
\begin{align*}
\text{SAFE MYSAFE} \\
\text{SEQLOG: FANOUT} \\
\text{NETV MSGROUTE SAM HOLD(Y)} \\
\text{\( \sim \) SEQLOG:} \\
\text{EDIT /ASEQLOG/ 1 1.* NEXTWORD} \\
\text{NETVIEW}
\end{align*}
\]

### PIPE FMTPACKT

**Syntax**

**FMTPACKT:**

```
FMTPACKT:
  SUMMARY
  FULL
  SHORT
  TALLY
  LOCAL
  GMT
  PORTSEL
  ASCII
  BOTH
  EBCDIC
  HEX
```

```
FMTPACKT:
  SUMMARY
  LOCAL
  PORTSEL
  ASCII
  BOTH
  EBCDIC
  HEX
```

```
FMTPACKT:
  SUMMARY
  LOCAL
  PORTSEL
  ASCII
  BOTH
  EBCDIC
  HEX
```

```
FMTPACKT:
  DUMP
  FORMAT
  SUMMARY
```

```
FMTPACKT:
  DUMP
  FORMAT
  SUMMARY
```
Synonyms

<table>
<thead>
<tr>
<th>Stage Operand</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORMAT</td>
<td>FMT</td>
</tr>
</tbody>
</table>

Command Description

The FMTPACKT stage takes raw TCPIP packet data, converts it into readable form, and generates reports that are passed to the primary output stream. The input stream is discarded unless there is a conversion error and there is a secondary output stream.
If a secondary output stream is connected, input stream data that cannot be converted is sent to the secondary output stream as the second message in an MLWTO message. The first message is DWO050E, which contains the return code and the reason code from the EZBCTAPI macro. Contact IBM Software Support with this information. If there is no secondary output stream, only the DWO050E message is written to the log.

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

FMTPACKT ends when the primary input stream or the primary output stream disconnects.

Operand Descriptions

**ASCII**

Dumped packet trace data is shown in hexadecimal and interpreted in ASCII translation only.

**BASIC**

Specifies the formatting option for packet trace data.

**DETAIL**

For specific packet types, format each element of the packet data. This applies to DNS, RIP, and SNMP packet data. This is the default value.

**SUMMARY**

For specific packet types, provide summary data for the packets. This applies to DNS, RIP, and SNMP packet data.

**BOTH**

Dumped packet trace data is shown in hexadecimal format and interpreted with both ASCII and EBCDIC translations.

**CLEANUP=nnnnnnn**

Defines a record interval. After the specified interval has elapsed, saved packet information in storage is released. The minimum value is 100 records; the maximum value is 1,000,000 records, and the default is 500 records. If the record interval is set to 0, cleanup does not occur.

**DUMP=nnnnnn**

Dump the selected packet in hexadecimal format with EBCDIC and ASCII translations, if these were selected or defaulted (the default value of PORTSEL produces both translations). The IP and protocol headers are dumped separately from the packet data. The value nnnnn represents the maximum amount of packet data that is to be dumped from each packet. The default value is 65,535 bytes. The minimum value is 0. The maximum value is 65,535. The IP and protocol headers are not subject to this maximum.

The PORTSEL, BOTH, ASCII, EBCDIC, and HEX keywords describe how the dumped packets are translated. The default value is PORTSEL. The display can be changed using these keywords. The default ASCII translation table is used. This table cannot match the table being used by the application.
If the STREAMS report is chosen, then the dump of the packets is deferred until the stream of data has been collected.

**EBCDIC**

Dumped packet trace data is shown in hexadecimal format and interpreted in EBCDIC translation only.

**FORMAT**

Specifies the format option.

- **DETAIL**
  Formats the IP header, protocol header, and the protocol data. This is the default value.

- **SUMMARY**
  Formats the IP header and the protocol header.

**FULL**

Equivalent to DUMP and FORMAT. SUMMARY is the default value.

**GMT**

The time stamps are converted to GMT time. LOCAL is the default value.

**HEX**

Dumped packet trace data is shown in hexadecimal format only with no translation.

**LINESIZE**

Specifies the line width at which the generated reports and data lines are wrapped.

If the output is directed to the operator screen, and a value for LINESIZE is specified that is greater than the width of the NetView operator screen, the displayed lines appear truncated. This appearance of truncation can be avoided by issuing the command using the WINDOW command which allows scrolling left and right.

* An asterisk indicates that the NetView operator screen width is used if running under an OST with a real or a virtual screen. Otherwise, the LINESIZE default value of 80 is used. A null value for LINESIZE is the same as LINESIZE=*.

    \[ nnn \]

    A value 60 - 250. The default value is 80.

**LOCAL**

The time stamps are converted to local time. This is the default value.

**NOREASSM**

Do not reassemble fragmented IP packets into a complete packet. REASSEM is the default value.

**NOSEGMENT**

Packet trace records that span multiple NetView IP trace records are not recombined. Only the first segment of a packed is used. The rest of the segment records are discarded. SEGMENT is the default value.

**PORTSEL**

For some well known ports, dumped packet trace data is shown in hexadecimal format and interpreted with either ASCII or EBCDIC translations, depending on how the port is defined. If a dump format selection cannot be made, both ASCII and EBCDIC translations are provided. This is the default value.
REASSEMB
Reassembles IP fragments into a complete packet.

(nn,DETAIL)
DETAIL generates the reassembly statistics for each packet when a packet completes reassembly.

nn specifies the maximum size allowed for a reassembled packet. This value can be 576 - 65535 bytes. The default value is 65535 bytes.

(nn,SUMMARY)
SUMMARY generates the reassembly statistics and information for packets that did not complete reassembly. This is the default.

nn specifies the maximum size allowed for a reassembled packet. This value can be 576 - 65535 bytes. The default value is 65535 bytes.

SEGMENT
Packet trace records that span multiple NetView IP trace records are recombined. Data from segmented records is saved until all the NetView IP trace records have been read to recreate the original packet. This is the default.

If the packet trace records as received from the PKTS QUERY command were truncated, the NOSEGMENT option is automatically used.

SESSION
List TCP and UDP session information.

DETAIL
List each of the packets for TCP and UDP sessions, as well as the summary statistics. This is the default value.

STATE
List the beginning and ending state for each TCP and UDP session.

SUMMARY
Show only the summary statistics for each TCP and UDP session.

SHORT
Equivalent to FORMAT=SUMMARY. SUMMARY is the default value.

SPEED
The link speed, in megabits per second, for the local (ls) and remote (rs) link. These values are used in throughput calculations in the TCP session report. Valid values are in the range 0 - 17171. The default value is 10. Specify the slowest speed of the link in the route.

STATS
After all of the records have been processed, generates statistical reports.

DETAIL
Lists the number of records selected by record type, device type, job name, link name, protocol number, IP address, and port numbers.

SUMMARY
Lists the IP address and port number pairs with the number of records, the first and last record numbers, and the first and last record times. This is the default value.

STREAMS
Collects the packet data for dumping or formatting after all the trace data has been processed.
PIPE FMTPACKT

(\texttt{nnn,DETAIL})
nnn represents the maximum amount of storage used to capture each stream. This value is specified in 1024 (1K) units. The range is 16 - 512 KB. The default value is 128 KB.

DETAIL generates messages about the status of each stream.

\textbf{Note:} The DUMP option is required to dump the packet data.

(\texttt{nnn,SUMMARY})
nnn represents the maximum amount of storage used to capture each stream. This value is specified in 1024 (1K) units. The range is 16 - 512 KB. The default value is 128 KB.

SUMMARY generates messages about each packet in the streams. This is the default value.

\textbf{Note:} The DUMP option is required to dump the packet data.

\textbf{SUMMARY}
Format a single line for each trace record. This is the default value.

\textbf{TALLY}
Equivalent to the STATS=DETAIL option. SUMMARY is the default value.

\textbf{Usage Notes}

FMTPACKT cannot be the first stage.

If input stream packets are truncated, the packet formatter might generate error messages.

PIPE HELDMSG

\textbf{Syntax}

\texttt{HELDMSG:}

\begin{verbatim}
HELDMSG
\end{verbatim}

\textbf{Synonyms}

\begin{array}{|c|c|}
\hline
\textbf{Stage Name} & \textbf{Synonym} \\
\hline
HELDMSG & HELED \\
\hline
\end{array}

\textbf{Command Description}

The HELDMSG stage 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 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

<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

HELDMSG terminates when it finishes processing its output or when the output stream disconnects.

Usage Notes

- HELDMSG must be the first stage.
- 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.

Example: Deleting Held Messages

To delete all the IEE123 messages on an operator's screen, enter:

```
PIPE HELDMSG
    LOCATE 1.6 /IEE123/
    CONSOLE DELETE
```

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
```

Command Description

The HOLE stage 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 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 NETVIEW 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 152 for a list of some commands for which command and response correlation is supported.

**Example: Waiting 5 Seconds**

The following command waits 5 seconds.

```
PIPE CORRWAIT 5 | HOLE
```

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:

\[
\text{INSTORE} - \text{ddname.membername} \quad \text{COMMON} \quad \text{LOCAL} \quad \text{NOREPLACE} \quad \text{REPLACE} \quad \text{CRYPTO} \quad \text{NOCRYPTO}
\]

Synonyms

Stage Operands          Synonym
-------------------------------------------
REPLACE                 REPL
NOREPLACE               NOREPL

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. See 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.

+0000000000  member loaded or unloaded successfully
+0000000008  member already loaded and REPLACE not specified
+0000000012  INSTORE terminated by secondary input request
+0000000016  user not authorized
+0000000020  insufficient storage or related internal problem
+0000000024  a record length above 255 was encountered
+0000000028  unload requested for a member that was not loaded

Termination Conditions

INSTORE terminates when its function completes.

Operand Descriptions

\textit{ddname}

Specifies the name of a standard NetView DDNAME, such as DSIPARM or DSICLTD. See the BROWSE command help for a list of valid DDNAMES.

\textit{membername}

Specifies the name of the member being loaded or unloaded under the specified DDNAME.

\textbf{COMMON}

Specifies that the common in-storage member is loaded or unloaded.

\textbf{LOCAL}

Specifies that the loaded or unloaded member is local to this procedure family (LRCE group).

\textbf{NOREPLACE}

Specifies that the member is loaded only if it is not already in storage.

\textbf{REPLACE}

Specifies that the member, if already in storage, is to be replaced.

\textbf{NOCRYPTO}

Specifies that the member, when loaded into storage, is not encrypted.

\textbf{CRYPTO}

Specifies that the member, when loaded into storage, is to be encrypted such that its contents are not visible in a storage memory dump. Determine the need for this against the processor 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 data sets 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:
  
  \textbf{PROTECT  }\texttt{ *.DSIPIINS.*.DSIPARM/M1 }

- To permit loading member M2 in DSICLTD in the local procedure family but prohibit loading it in COMMON storage, code the following PROTECT statement:
  
  \textbf{PROTECT  }\texttt{ *.DSIPIINS.COMMON.DSICLTD/M2 }
Note: Do not prohibit LOCAL access to the DSIOPEN and CNMPNL1 DDs.

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 might not indicate a comment in every member and stripping these lines can 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

See 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 to INSTORE*/
  '  B: INSTORE DSIPARM.TEST7 COMMON REPLACE',/*Load COMMON */
  '  HOLE', /*End pipe 1       */
  '  ; A:', /*END comes here... */
  '  B:', /*back to INSTORE secondary-input */
  '  CONS' /*Output INSTORE's return code */

PIPE INTERPRT

Syntax

INTERPRT:

\[
\text{INTERPRT} \quad \text{[*]}
\]

Synonyms

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERPRT</td>
<td>INT</td>
</tr>
</tbody>
</table>

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 is too long for the
processing environment. For example, the NetView Command List Language restricts commands to 240 characters. If your pipeline specification is 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 specifications coded must be considered to be complete stage specifications. The INTERPRRT 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 is appropriate if a complete PIPE command is coded on the NETVIEW stage.

Often, the stages are read into the pipeline by way of the STEM stage. They are processed by the INTERPRRT 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

INTERPRRT modifies the stages in the current message. Because it is a pipeline builder stage, INTERPRRT does not have its own termination conditions. See the information on the stages INTERPRRT is modifying for termination conditions.

### Operand Descriptions

* Specifies that the current message is to be processed by INTERPRRT. Asterisk (*) is the default.

### Usage Notes

- INTERPRRT can be used anywhere in the pipeline specification.
- Do not include an INTERPRRT stage among the stages being inserted by the INTERPRRT stage.
- All stages are assigned numbers based on their position in the pipeline specification. These numbers are shown in some error messages. Stage names inserted by the INTERPRRT stage are assigned numbers according to the following formula:

\[
(pn) + ((10000)(sn))
\]

Where:

- \(pn\) Specifies the positions of the stages in the pipeline specification.
- \(sn\) Specifies the stage number of the INTERPRRT stage in the pipeline specification.

### 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 INTERPRRT stage provides for the possibility that OPER_CMD can cause the pipeline specification to exceed the 240-character limit.

```rexx
/* 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 */
/* (X is the name of the variables), and drive */
/* the INTERPRRT stage. */
/* */
'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 are formed into a multiline message. The NETVIEW stage uses the pipeline message it receives to drive the INTERPRRT stage. The INTERPRRT stage interprets each of the data records in the message as a complete stage specification, then builds stages from the input data for the command. All such stages are substituted into the pipeline specification in place of the INTERPRRT stage.

An equivalent pipeline created without using the INTERPRRT stage is shown below. This pipeline is simpler but fails if the size of OPER_CMD forces the entire specification to become too large.

```rexx
/* SAMPLE REXX COMMAND LIST */
'PIPE NETVIEW OPER_CMD',
  'SEPARATE',
  'TAKE 10',
  'COLLECT',
  'CONSOLE'
EXIT
```

**Example: 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 number is:

```
(3) + ((10000)(1)) = 10003
```

because TAKE is the third stage inserted by the INTERPRRT stage in the pipeline specification and INTERPRRT is the first stage 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 DSICLCLD. There must be one stage specification per record. To run the pipeline, enter:
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 can replace PIPE (NAME INPIPE) INTERPRT * with this:

```
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.

```
PIPE (NAME OUTPIPE) < DSICLD.LGPIPE
  COLLECT
  NETVIEW PIPE (NAME INPIPE) INTERPRT *
  LOGTO HCYLOG
```

---

**PIPE IPLOG**

**Syntax**

```
IPLOG:

|— IPLOG— —host— —port— —facility— —priority—|
```

**Command Description**

The IPLOG stage 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 might require the SEPARATE stage. See “PIPE SEPARATE” on page 188.

**Operand Descriptions**

**host**

Specifies the name of the remote host. It can be specified as a host name or an IP address.
port
   Specifies the server port to use. In most cases, it is 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.

Example: Sending a System Log Message
   PIPE LIT /My Message/
   | IPLOG NMP119 512 USER ALERT

PIPE JOINCONT

Syntax
JOINCONT:

```
JOINCONT  _NOT_ _LEADING_  _TRAILING_ _/string_/ |
```

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 can include blanks. For instance, an 80-byte record that is read from a file into the pipeline can 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 is appended to the previous message.

NOT
   Specifies that the absence of the specified strings can cause lines to be joined.

/strint/
   Specifies the character string as one of the following strings:
PIPE JOINCONT

- 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 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/ up 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.
- 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, yields unpredictable results.

**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 'PIPES 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
    LOCAL/keep_id/
    GLOBAL/keep_id/
    (note)
    timeout
    APPEND
    SEIZE
    (note)
    exitOption
```

exitOption:

```
NOSPILL
SPILL
ENDCMD /cmd string/
```

Command Description

The KEEP stage enables you to define a place to store messages. You can read from, or write to, storage defined by the keepname or keep_id. The name and message 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 and into the output stream.
The PIPE KEEP stage is similar to the PIPE SAFE stage. PIPE KEEP enables you to define a task or place to store messages; PIPE SAFE is a place to store one or more messages associated with a command procedure.

To display information about KEEPs that are currently defined, use the QRYKEEP command.

**Operand Descriptions**

**keepname**

A 1- to 8-character name of the KEEP. This name is case-sensitive.

Note that if you use the same name as the *keep_id* that is specified with the LOCAL or GLOBAL specification, each KEEP that is defined is separate and distinct.

**LOCAL**

Specifies that the KEEP can be used only from the task where it is created. The KEEP is identified by the specified *keep_id*.

**GLOBAL**

Specifies that the KEEP can be used from any NetView task. The KEEP is identified by the specified *keep_id*.

**keep_id**

Specifies a delimited string that identifies the KEEP. Note that if you use the same string value with both the LOCAL and GLOBAL specifications, the two KEEPs that are defined are separate and distinct. The keep-id can be 1 - 255 characters in length. These can be any characters except for delimiters.

**timeout**

Specifies a timeout period. If no message is added for the specified period, 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 are to 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.

**Usage:**

1. When no exit option is selected and the KEEP already exists, the previous option remains in force. When the KEEP does not exist, the default action is NOSPILL.
2. A task termination causes a timeout for a local keep. A global KEEP
does not expire when a task ends, but only when the NetView program
terminates.

NOSPILL
Used when KEEP is not a first stage. When the KEEP expires, NOSPILL
indicates to discard messages in the KEEP.

ENDCMD
Used when KEEP is not a first stage. If ENDCMD is specified, the
delimited string /cmd string/ (described below) is required. ENDCMD
specifies what is to happen when the keep expires. A local keep expires
when either a timeout occurs or when the task is terminated. A global keep
expires when the NetView program terminates.

When the keep expires, the specified command is invoked once for each
message in the keep. For a local keep, the commands run at the task where
the keep exists. For a global keep, the commands run at the task specified
in the CNMSTYLE statement endcmd.autotask. For each invocation, the
message being disposed is made to be the current message. Command
invocation is at low priority, meaning non-preemptive. However, if CMD
HIGH is specified as the first eight characters of the command string, then
it is queued at high priority and preempts other procedures.

When the local keep expires because a task ends, the command is invoked
only if it is EXCMD or DOM (neither EXCMD nor DOM synonyms is
honored, and no command is invoked if the NetView program is closing).
Note that a global KEEP expires when the NetView program ends
normally. A time period given by the CNMSTYLE statement
endcmd.close.leeway. If the NetView program ends because of an abend,
CLOSE ABEND, or system cancel command (abend), endcmd processing is
terminated.

/cmd string/
A delimited string used only with the ENDCMD option. This string defines
the command to be invoked with each message in the keep at termination.
The command is invoked with the same authority as the pipe command
that specified the command string. Be careful when composing the
command because neither authority nor syntax is checked until the time of
invocation.

Example: Accumulating Data

In the following examples, data is accumulated about a resource prior to an action:

```plaintext
IF MSGID='IST950I' & 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:

```plaintext
/* CHKNCP REXX */
parse arg keepName /* Keep names are case sensitive */
'PIPE (NAME CHKNCP1),'
'| KEEP keepName 'SEIZE', /* empty, not copy, the Keep */
```
Example: Operator Logoff Actions

In a user-written REXX procedure, you can use

'PIPE SAFE * | EDIT MSGSENDER | VAR targetOP'

to determine the source of the current message.

/* Use AUTO1 to take action related to other tasks logging off */
'PIPE (NAME SETLOGOF)'
'
' LITERAL /any text here/ ,
' KEEP LOGOFF * ENDCMD /EXCMD AUTO1, myRexx/

Example: Using Global Keep

To use a global keep, you can use

IF MSGID='IST950I' & TEXT = 'NCP44'. THEN
  EXEC(CMD('PIPE SAFE * | KEEP GLOBAL /NCP44/ 600 APPEND'))
  ROUTE(ONE NETOP1);

Note that references to KEEP GLOBAL /NCP44/ do not need to be invoked on the same opid.

IF "MSGID'='IST6778A & TEXT='NCP44' THEN
  EXEC(CMD('CHKNCP NCP44'));

Where CHKNCP begins as follows:

/* CHKNCP REXX */
parse arg keepName /* Keep names are case sensitive */
'PIPE (NAME CHKNCP1)',
'
' KEEP' GLOBAL 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 Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>LITERAL</td>
<td>LIT</td>
</tr>
</tbody>
</table>

Command Description

The LITERAL stage 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 ends when the input stream or output stream is disconnected.

Operand Descriptions

/\text{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, messages remain in the pipeline and text added by this stage is inserted in front of the existing messages.

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 /\text{string}/ BLANK NULL
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE</td>
<td>LOC</td>
</tr>
</tbody>
</table>

Command Description

The LOCATE stage 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, 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 ends when the input stream or both output streams are disconnected.

**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. You can specify the letter $ for the length if the specification is followed by a delimited string. The LOCATE stage replaces the letter with the length of that delimited string.

**/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.

**BLANK**

Specifies that the character string for which to search contains only blanks. The search occurs in the range specified by the position.length parameter, but if the data contains only blanks, a match is recognized regardless of the length specified.

**NULL**

Specifies that the stage is to search for no data whatsoever, that is, null data (not even blanks). This means that a match is recognized only when the data is shorter than the number specified for position in the position.length parameter.
**Usage Notes**

- LOCATE cannot be the first stage.
- You can specify the `position.length` and `/string/` pair up to 40 times.

**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:

```
LOGTO

+---+---+---+---+
|   | ALL| CANZLOG| HCYLOG| NETLOG| SYSLOG |
+---+---+-------+-------+-------+-------+
```

**CANZLOG:**

```
CANZLOG

+---+---+
|   | NVMSG| TRACE |
+---+---+---+
```

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGTO</td>
<td>LOG</td>
</tr>
</tbody>
</table>

**Command Description**

The LOGTO stage 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**

The LOGTO stage ends 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.

**CANZLOG**
Sends the message to the CanzLog log regardless of DEFAULTS or OVERRIDE command settings.

**NVMSG**
Logs the message as a Canzlog NetView message. No other Canzlog tags are turned on. This is the default if no Canzlog tags are specified.

**TRACE**
Sends the message as a Canzlog trace (only) message. No other Canzlog tags are turned on.

**HCYLOG**
Sends the message to the hardcopy log regardless of DEFAULTS or OVERRIDE command settings.

**NETLOG**
Sends the message to the network log and to the Canzlog log regardless of DEFAULTS or OVERRIDE command settings.

**SYSLOG**
Sends the message to the system log regardless of DEFAULTS or OVERRIDE command settings.

Usage Notes

- The LOGTO stage cannot be the first stage.
- In the LOGTO stage, do not specify * or ALL operands with any other keywords. You can specify any or all of the other logs in any combination.
- If you use the CANZLOG, NETLOG, SYSLOG, HCYLOG, or ALL operands, the settings for all logs in the DEFAULTS and OVERRIDE commands are ignored, not just the logs specified on the LOGTO stage.
- Messages written to the Canzlog file by the LOGTO stage are considered new messages and might result in duplication in the Canzlog file.

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:**

```
LOOKUP 1.*
```

```
detail_position.length reference_position.length
```
**Command Description**

The LOOKUP stage compares two input streams and indicates on its output streams whether 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 is 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 is to compare a new data stream to an old data stream to determine added, deleted, or unchanged lines.

**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 ends 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 is a multiline message consisting of the detail record and the reference record. If the detail record is a multiline message, the multiline message is 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
is compared to the reference records.

Detail_position can be any positive number.

Length can be any positive number or an *. An asterisk (*) indicates that all
characters in the message beginning at the position indicated by detail_position
are 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 *. An asterisk (*) indicates that all
characters in the message beginning at the position indicated by
reference_position are 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 is
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 are 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 are 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 are
considered a match.

The third character, z, indicates the exception to the x value. Any single
character in the character position indicated by x is 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 items are considered a match:
• Detail record fields at least 4 characters long beginning with an A, and
• Having any character other than an asterisk (*) as the second character, and
• Having the third and fourth characters CD.

Note: Z can be used to check a detail record that represents a wildcard
pattern. The intended use is to enable security checking of wildcard patterns.
If a blank is specified for any of the three characters, x, y, or z, indicates to not
use that option to select matching records.
Usage Notes

- LOOKUP requires two input streams and supports up to three output streams. If an output stream is not connected, the data to be 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 can clog.

For information about clogged pipelines, see "Clogged Pipelines" on page 31.
- 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 are 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.
- Avoid duplicate reference records. LOOKUP processing cannot guarantee which of the duplicate records is to be examined. Duplicate reference records must be particularly avoided if the APPEND keyword is specified, because the record selected by LOOKUP to be appended is unknown.
- Reference records passed to the third output stream might not be in the same order as received on the secondary input stream.

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 %)'
PIPE LOOKUP

Note:
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, see 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

See 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

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMLIST</td>
<td>MEML</td>
</tr>
</tbody>
</table>

IDLEOFF Example

See 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.
**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, see 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.

**(DSN)**

Specifies that the specification is for a data set name.

**dsnnd**

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- to 8-character value without period delimiters or quotation marks, it is considered a DD. Otherwise, it is considered a DSN. The *dsnnd* value can 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 produces the following text: '+0000000032 XX'.

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
</table>
The user is not authorized to the data set.
The data set is unavailable. It might be in use by another user or task.
The data set does not exist.
Input or operand is not valid.
An RDJFCB error occurred. It might be a nonpartitioned DSORG.
An unspecified OPEN error occurred.
An unspecified READ error occurred.
A system problem occurred.

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
```

**Command Description**

The MVS stage 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 152 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, \textit{cmdtext} is required.

If MVS 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.

**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 might 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 IBM Software 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 can result in the return codes issued by the MVS command or various return codes indicating storage allocation failures. Message DSI124I is also issued.

For information about the NetView MVS command, see 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.

**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 TAG |

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>NETVIEW</td>
<td>NETV</td>
</tr>
</tbody>
</table>

**Command Description**

The NETVIEW stage specifies to run a NetView, MVS, or 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 are 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 is disconnected, NETVIEW continues 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

cmdtext

Specifies the command.

If the NETVIEW stage is the first stage, *cmdtext* is required.

If the NETVIEW stage 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.

If *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 *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 are 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.

CGI

Use the CGI option for a command that can 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 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 might have returned. If you do not specify MOE (message on error), return codes from commands are ignored.

NOPANEL

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

NOPANEL has restrictions when used with the ATTACH command. See the IBM Tivoli NetView for z/OS Command Reference Volume 1 (A-N) or the NetView online help for information about ATTACH.

TAG

The value for TAG is a 1 - 16 character string, which is placed in the
IFRAUSRC field of each correlated response. This value overlays any previous value for the length given. If the value contains any blanks, commas, or equal signs, you must enclose the value within quotation marks.

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 DS1039I 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 DS1001I 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 and PERSIST stages; see "PIPE CORRWAIT" on page 49 and "PIPE PERSIST" on page 161.

• The following commands are among those NetView commands supported:
  – User-written commands that use the CORR keyword on DSIPUSH, DSIFIND, or DSIMQS assembler macros. See the IBM Tivoli NetView for z/OS Programming: Assembler for more information about these macros.
  – MVS system commands issued from the NetView program.

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. See IBM Tivoli NetView for z/OS Programming: 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 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 the long running command support provided by the NetView program 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 storage 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. The NetView program
  ignores the blank, but passes it to another application where it is processed as
  part of the stage data.

- If you are using the NETVIEW stage to invoke a command procedure that uses
  the TRAP command, make sure you allow the trapping to occur. You can use the
  EXPOSE stage after the NETVIEW stage to allow the trap exit to detect messages
  for trapping.

**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 the specific return code to the IBM Software 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>

Other return codes indicate either a storage failure or the return code from the
command. If there is a DSI124I message at the system console for this same time
frame, you can assume the storage failure caused the non-zero return code (the
code depends on the processing stage where the storage failure occurred).
Otherwise, the return code was returned by the command.

**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:

\[\text{NLOCATE} \rightarrow \text{position.length} / \text{string} / \text{BLANK} \rightarrow \text{NULL} \rightarrow \]

Synonyms

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLOCATE</td>
<td>NLOC</td>
</tr>
</tbody>
</table>

Command Description

The NLOCATE stage 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, matches 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 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. You can specify the letter S for the length if the specification is followed by a delimited string. The NLOCATE stage replaces the letter with the length of that delimited string.

/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.

BLANK
Specifies that the character string for which to search contains only blanks. The search occurs in the range specified by the position.length parameter, but if the data contains only blanks, a match is recognized regardless of the length specified.

NULL
Specifies that the stage is to search for no data whatsoever, that is, null data (not even blanks). This means that a match is recognized only when the data is shorter than the number specified for position in the position.length parameter.

Usage Notes
- NLOCATE cannot be the first stage.
- You can specify the position.length and /string/ pair up to 40 times.

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:

```
NLS:
```

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.

See IBM Tivoli NetView for z/OS Command Reference Volume 1 (A-N) 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 are not translated.

Usage Notes

- After being translated by NLS, and as long as the message attributes remain unchanged, messages are not translated again by a subsequent NLS stage or by Presentation Services.
- Translated messages must not be stored in a variable and reissued as new messages. The resulting messages are unusable.
- Remote domain translation requirements are ignored if a message translated by NLS is routed cross domain. Do not route translated messages cross domain.

Example: Display a Translated Message

The following example displays the translated version of BNH054 on the view panel:

```plaintext
/* Write BNH054 to the IMMED message area */

"PIPE (NAME PlsWait)",
" | NETVIEW MESSAGE BNH054",
" | NLS",
" | EDIT /MESSAGE IMMED DSI009 '/ 1",
" | 1.* NEXT",
" | '/ / NEXT",
" | NETVIEW"
```

Note:

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

NOT:

| NOT -- stage_specification |

Command Description

The NOT stage 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 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

stage_specification

The stage specification being modified, including its operands.

Usage Notes

- The NOT stage does not invert the function of the specified stage. NOT SEPARATE does not become COLLECT. NOT SEPARATE discards all input, because SEPARATE keeps all input.
- The STRIP stage is not considered a selection stage. NOT STRIP discards all input.
- The stage NOT CHOP chop-operands keeps the part of each line that CHOP usually passes to the secondary output stream, if connected.
- 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.
- NOT is only supported for stages with two output streams.

Example: Passing Messages After One Containing a Certain String

The following example displays all data that follows the separator line, where TOSTRING normally stops.
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 NPDAEVD

**Syntax**

NPDAEVD:

```
NPDAEVD:
```

**Command Description**

The NPDAEVD stage command outputs NPDA (hardware monitor) event detail text messages and recommended actions. This stage is designed for use in a command list that automates NPDA events, and requires as input the type of message or messages that \texttt{SAFE *} produces in that environment. It outputs information that is difficult to derive directly from the input management services unit (MSU) with the MSUSEG function. If input that is not valid is provided, the NPDAEVD stage outputs message CNM464I. If valid input is provided, the NPDAEVD stage outputs up to five multiline messages described as follows:

Recommended actions:
- \texttt{LINES WHICH CLOSELY APPROXIMATE THE NPDA RECOMMENDED ACTION DISPLAY}

Event description:
- NPDA "LONG" EVENT DESCRIPTION (USUALLY ONE LINE)

Probable cause:
- NPDA "LONG" PROBABLE CAUSE (USUALLY ONE LINE)

Qualifiers:
- 1. QUALIFIER 1 TEXT
- 2. QUALIFIER 2 TEXT

Other:
- OTHER NPDA NON-GENERIC-ALERT TEXTS

**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

NPDAEVD terminates when its input stream disconnects.

Usage Notes

- NPDAEVD cannot be a first stage command.
- NPDAEVD supports only SNA major vectors (typically a "0000" generic or non-generic alert) within an NMVT or newer envelope. Older architectures, such as RECMS or RECFMS, are not supported.

Example: Creating NPDA Event Detail Text

The following example is in an MSU-automation REXX procedure. This example creates NPDA event detail texts. In this example, the NPDA event had no qualifiers or "other" data.

`'PIPE SAFE * | NPDAEVD | CONS'`

You receive a response similar to the output (three multiline messages) in the following example:

Recommended actions:
- USER CAUSED - NONE
- INSTALL CAUSED - NONE
- FAILURE CAUSED - TOKEN-RING INTERFACE COUPLER (TIC)
  ACTIONS - D123 - REACTIVATE THE PHYSICAL LINK
  IF PROBLEM PERSISTS THEN...

Event description:
- HARDWARE ERROR DETECTED AT INITIALIZATION

Probable cause:
- TOKEN-RING INTERFACE COUPLER

PIPE PERSIST

Syntax

**PERSIST:**

```
PERSIST minutes MINUTES DISPLAY ROUTE label: COMMAND cmd TRAP tText
```

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 active following the completion of the pipeline in which it is found. The range is 0 - 100,000.

- **MINUTES**
  Optional parameter, used only for readability.
PIPE PERSIST

DISPLAY
Indicates that the correlated output is to be displayed on the console of the task where the pipeline ran. The output messages can be automated.

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.

TRAP
Specifies that messages are to be added to the message queue of the REXX procedure which called the pipeline containing the PERSIST TRAP stage. Such messages satisfy a REXX WAIT FOR MESSAGES command and are accessible to a MSGREAD command.

See the sample usage of PERSIST TRAP under option 13 in sample CNMS1101.

tText
Specifies optional text to be written to the message queue as a last message at the time the persist completes. Note that MVS commands and the VTAM VARY command do not generally give an indication of completion. For these commands, the tText parameter is only written to the queue when the specified timeout occurs. After receiving the message containing this text, the procedure does not issue another WAIT FOR MESSAGES request or use the VIEW option EXTEND with respect to the completed persist.

Usage Notes

• Issue NCCF LIST PERSIST to get information about outstanding PERSIST elements.

• 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 statements is true:
  – 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 PERSIST stage can be used after a NETV stage without any WAIT stage. In this case, the PERSIST stage is always enabled and absorbs all messages from the preceding command, even synchronous messages.

• The command, routing, or display specified on your PERSIST stages is activated by the arrival of a message that was to be 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 takes 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, is subject to exposure in the same way 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.
- When PERSIST TRAP is active, the invoking procedure can issue a WAIT FOR MESSAGES command or use the EXTEND option on a VIEW command to wait for additional data. When the persist completes, the REXX function EVENT() returns a value of X.
- Do not use the TRAP option when including the PERSIST stage in sample DSICCDEF.

**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:

```plaintext
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.

**Example: Continuing Work During an Asynchronous Request**

To continue with other work while an asynchronous request completes, enter:

```plaintext
'PIPE CPDOMAIN USIBMNT.RMTCP',
'| PERSIST 1 TRAP'
/* other work here, including other pipe commands
with their own waits or persists */
/* we've been "waiting" some already... see if any
more waiting is needed... */
'WAIT 20 SECONDS FOR MESSAGES'
IF EVENT() = 'M' THEN
do;
'MSGREAD'
/* act on msg from CPDOMAIN */
end;
```
**Syntax**

PICK:

```
PICK position.length ¬= position.length

¬< ≤> ≥

PAD '00'X

PAD hexstring /string/
```

**Command Description**

The PICK stage 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.
- `<`  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:

'nnX'

`Xnn`

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.
- PICK examines only the first line of the input message. Use SEPARATE to test each line or EDIT to rearrange lines if other tests are required.

**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...
WINDOW  63  1446  71512 08/08/10 13:38:19 R
```

`/* sample for REXX */`

`'PIPE (NAME LOWISEREX)'`

`NETVIEW MAPCL', /* obtain display of REXX in storage */`

`SEPARATE', /* handle lines individually */`

`DROP 3', /* header lines */`

`DROP LAST 2', /* trailer line and totals */`

`PICK 14.5 < 6/', /* compare 5 chars from data with "6"*/`

`CONSOLE' /* display result */`
Note:
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 can be restored to the output using FANINANY and the secondary outputs from the two DROP stages. However, the totals on the trailer lines no longer accurately reflect the data lines above them.

Additional Examples

Additional examples can be found in CNMS1101.

PIPE PIPEND

Syntax

PIPEND:

```
PIPEND: *

number
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIPEND</td>
<td>PIPEEND</td>
</tr>
</tbody>
</table>

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 ends 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 is used to determine the return code for the pipeline. If this data is a number, it is returned as the return code from the PIPE command. If it is not a number, return code 100 is returned from the PIPE command.

[number]

Specifies the return code to be returned from the PIPE command.
Number can be any number up to $2^{31} - 1$.

Usage Notes

Return codes returned by the PIPE command are not to be used because of their predefined meaning to the NetView program. These return codes are shown in “PIPE (NCCF)” on page 19.

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 must not be used with PIPEND.

Example: Ending a Pipeline with a Return Code

The pipe in the following example ends with two different return codes depending on whether the WAIT times out.

If you pass the command LIST STATUS=DSILOG as the first parameter to this REXX example and the log is not busy, the LIST request easily completes within 2 seconds and the rc returned from the PIPE command is 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, 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 secs*/
  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 is passed as an argument, the pipeline always timeout and rc always sets to 8. Although this is a simple command which completes in 2 seconds or less, MVS and VTAM commands do not inform a pipeline when they 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 is zero.
**Syntax**

**PPI Sender:**

```
PPI (DATAONLY) receiver_name
(TERCRTE) (TECRTCFM) (TRAPROUT) (MLWT0) (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, follow the PPI with CORRWAIT * so data can be received continuously without deactivating the receiver. If the receiver is deactivated, even for a short time, senders might 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 19 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, follow the PPI with CORRWAIT with a sufficient wait time for the response to be returned. PPI automatically ends the wait when one message is received. This message can 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 the NetView program:

- 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 line attributes are provided.
- Line type must be one of the following types:
  
<table>
<thead>
<tr>
<th>Line Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Control line</td>
</tr>
<tr>
<td>L</td>
<td>Label line</td>
</tr>
<tr>
<td>D</td>
<td>Data line</td>
</tr>
<tr>
<td>E</td>
<td>End line (can contain data)</td>
</tr>
</tbody>
</table>

**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 the NetView program.

When PPI is a receiver, *receiver_name* is the name that is used by other programs to communicate with the PPI stage.
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 (IFRAUSD). 

**APONLY**

Specifies that messages are accepted only from an APF authorized program.

**DATAONLY**

Specifies that only the data portion of each line is sent. NetView buffer headers and structures are not sent. DATAONLY is only valid when PPI is used as a sender.

**MLWTTO**

Specifies that the message receiver can receive messages formatted to NetView multiline message standards. Messages are sent as a NetView multiline message unit. MLWTTO 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 is converted by the Event/Automation Service into an Event Integration Facility (EIF) event and sent to the server.

**TECRTCFM**

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 or alert is converted by the Event/Automation Service into an an Event Integration Facility (EIF) event and sent to a server. The Event/Automation Service expects the server to send a reply confirming or rejecting the EIF event.

**TRAPROUT**

Specifies that the alert is to be formatted and transferred to the Event/Automation Service associated with the named PPI receiver. The alert is 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/\n
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, to send the entire original alert. Additions to the alert can be made using the NAMEBIND EDIT order. Deletions or other changes to the alert can cause the message to be unrecognized as an alert by the alert adapters.

- When sending messages to another NetView system, the NV option preserves 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 canceled 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. See the *IBM 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 is 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 PPIPREFX, see *IBM Tivoli NetView for z/OS Command Reference Volume 1 (A-N).*

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 incorrect 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 DSIPIPPI 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:

```
PIPE PPI GEORGE | WAIT ...
```

Code the following PROTECT statement:

```
PROTECT *.*.DSIPIPPI.RECEIVE.GEORGE
```

To prohibit using:

```
PIPE LITERAL /STUFF/ | PPI SAM ...
```

Code the following PROTECT statement:

```
PROTECT *.*.DSIPIPPI.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 might 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. See the <em>IBM 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>Did 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 because of the receipt of an unrelated message from the same sender.</td>
</tr>
<tr>
<td>1004</td>
<td>An illegal alert forwarding loop was detected. The NetView program attempts to avoid loops by not forwarding alerts back to their source. Specifically, PPI (TECROUTE) or PPI (TECRTCFS) is not valid for a...</td>
</tr>
</tbody>
</table>
generic alert whose subvector X'92', flag bit 7, is on ('1' B). PPI (TRAPROUT) is not valid for a generic alert whose subvector X'92' flag bit 4 is on. For more information, refer to the SNA library.

1005 The specified target type does not support the data. For example, (TRAPROUT) was specified for a message which is not an alert.

1012 The user is not authorized.

Other Any valid return code returned by a PPI request type 4 (init), 14 (send), or 22 (receive). See the IBM Tivoli NetView for z/OS Application Programmer’s Guide for information about these return codes.

Example: Generating an Alert from Hex Data

The following REXX example produces an alert similar to the following example 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 might not be recognized.

```
/*** Make an alert *******
altxt = '4103B00000000000000000000700000'X
altxt = altxt || '0892000001100012345678'X
altxt = altxt || '1010000D110E0A0040F2F3F4F5F6F7F8'X
altxt = altxt || '069304032012'X
altxt = altxt || '0E950601150213E1068101011504'X
altxt = altxt || '1103030109C7C5D5C1D3F34040C3D6D4C3'X
altxt = altxt || '04931001'X
altxt = altxt || '30310602046E01F40512'X
altxt = altxt || '032111'X
altxt = altxt || '2030'X
altxt = altxt || 'Here is my subvector 31 stuff.'
'pipe (end =) var altxt',
| 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 receive the response ERROR 1.

Note: The receiver is not shut down when the pipeline ends so the response can be generated.

```
/*** responding to a request *******
PIPE (NAME CNMCOUNT END -)',
| PPI CNMCOUNT', /* receive for receiver name "CNMCOUNT" */
| WAIT *, /* CORRWAIT (until 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 */
| /* ---------------------------- end of pipeline 1 ---------------- */
| ~ X: ', /* invalid requests come here. */
| EDIT /ERROR 1/ 1', /* error message... */
| PPI *', /* sending error message to requestor */
```

Note:
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 can be written with the WRITELINE order to create a multiline message. Because this example does not assume that the requestor is another NetView system, it cannot assume that the requestor can handle a multiline response.

Adding the NV option to the PPI send stage, stage 6, adds 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 system. Because it is another NetView system, a multiline response is possible.

```plaintext
/***
*** issuing a request ********/
address NETVASIS,
'PIPE (NAME PPIOPS)',
| PPI OEXXX /egrep "NetView" set.log/; /*sendrequest 'egrep'*/
| WAIT 180', /*wait to 3 min for resp */
| STEM NVnSET.' /*store response */
```

Note:

1. The wait automatically ends when one message is received. This message can be a multiline message.

2. Multiline identifiers and line descriptors, if any, are removed when the multiline response is built by PPI processing.

Additional Examples

For additional PPI examples, see the following specifications:

- “PIPE EDIT” on page 67
- Example CNMEALUS in DSICLD
- Example CNMEMSUS in DSICLD
- Sample CNMS1101

### PIPE PRESATTR

**Syntax**

```
PRESATTR:
```

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESATTR</td>
<td>COLOR, COLOUR</td>
</tr>
</tbody>
</table>
### Stage Operands

<table>
<thead>
<tr>
<th>Stage Operands</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<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>UNDERSCORE</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 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 colors:
  - **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 can be overridden by message automation. To ensure that you get the settings you have specified, use CONSOLE ONLY or WINDOW to display your messages.

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
PIPE QSAM

Syntax

QSAM:

```
QSAM
  (DD)
  (DSN)
  data_set_name
  data_definition_name
  APPEND
```

Synonyms

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>QSAM (read)</td>
<td>&lt;</td>
</tr>
<tr>
<td>QSAM (write)</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

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 quotation marks. The < synonym has additional functionality as the < (from disk) stage. See “PIPE < (From Disk)” on page 245 and “PIPE > (To Disk)” on page 248 for more information.

Command Description

The QSAM stage 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 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 quotation marks. The quotation marks are ignored.

When specified as a first stage to read a file, QSAM reads 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 is disconnected.
A signed 10-digit decimal return code is written to the secondary output stream if the secondary stream is connected.

**Operand Descriptions**

`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- to 8-character value without period delimiters or quotation marks, then it is considered a *data definition name*. Otherwise, the argument is considered a *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 can include a member name as part of the specification. A data definition name that is specified to QSAM with a member name is 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.

`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- to 8-character value without period delimiters or quotation marks, it is considered a *data definition name*. Otherwise, the argument is considered a *data set name*.

`(DD)`

Specifies that the specification is for a data definition.

`(DSN)`

Specifies that the specification is for a data set name.

**APPEND**

Specifies that the data is to be appended onto the end of the records that are already in the file (otherwise writing generally occurs from the start). APPEND causes the DCB to be opened internally with the EXTEND option. Another method for appending onto the end is to allocate the file using the MOD keyword, in which case the QSAM APPEND option is redundant. See the ALLOCATE command in the NetView online help or *IBM Tivoli NetView for z/OS Command Reference Volume 1 (A-N)*.

Do not use the APPEND option when writing to a PDS member.

**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. See the *IBM 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.
- The QSAM stage allocates the data set with DISP=SHR specified and then opens the data set requesting OUTPUT access. It is possible for another application or task to be writing or reading the data set at the same time as the PIPE QSAM.
stage is reading or writing the data set. You might encounter this restriction when two different QSAM stages are attempting to write to different members of the same PDS. When this situation occurs, the QSAM stage issues message DSI084I and a return code of 16. In this case, the system issues the message "IEC143I 213-30".

- If you omit the FILE operand, a unique $ddname$ with a name of $SYSnnnn$ is assigned by dynamic allocation and then returned in the CNM272I message. Do not specify the FILE operand unless a specific $ddname$ must be allocated. This prevents allocations failing because of $ddname$ conflicts. It also prevents problems caused by deallocating a data set that is shared by multiple NetView tasks. Each NetView task must allocate the file with a unique $ddname$. If one task deallocates its $ddname$, the other tasks do not lose their access to the file.

- If you allocate a partitioned data set as an input data set and specify a member name that does not exist, the ALLOCATE command completes successfully with a return code of 0. However, you receive an OPEN error when you attempt to open the data set for input.

- Allocate the files with the FREE operand whenever possible. The files are then deallocated automatically when they are closed. This reduces virtual storage use. There is also an MVS limit of 1635 concurrent allocations. When this limit is reached, deallocate resources to do additional allocations. Allocating files with the FREE operand helps to keep the allocations below the limit. This procedure also minimizes the time that critical data sets, volumes, and units are tied up. System output data sets also are spooled immediately when the files are closed, instead of when the NetView program ends.

- If you specify the same operand more than once on the ALLOCATE command, the last one specified is used and the previous operands are ignored.

- The NetView LISTA command displays the $ddnames$ and $dsnames$ of currently allocated files.

- For disk files, the following operands are ignored by dynamic allocation:

<table>
<thead>
<tr>
<th>OPERANDS</th>
<th>ALLOWED VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPIES</td>
<td>HOLD</td>
</tr>
<tr>
<td>DEN</td>
<td>OUTLIM</td>
</tr>
<tr>
<td>DEST</td>
<td>POSITION</td>
</tr>
<tr>
<td>FORMS</td>
<td>WRITER</td>
</tr>
</tbody>
</table>

- For tape files, the following operands are ignored by dynamic allocation:

<table>
<thead>
<tr>
<th>OPERANDS</th>
<th>ALLOWED VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK</td>
<td>TRACK</td>
</tr>
<tr>
<td>CONTIG</td>
<td>MXIG</td>
</tr>
<tr>
<td>COPIES</td>
<td>MXIG</td>
</tr>
<tr>
<td>DEST</td>
<td>RELEASE</td>
</tr>
<tr>
<td>DIR</td>
<td>ROUND</td>
</tr>
<tr>
<td>FORMS</td>
<td>SPACE</td>
</tr>
<tr>
<td>HOLD</td>
<td>NOHOLD</td>
</tr>
</tbody>
</table>

- The QSAM stage uses one or more QSAM read or write operations. The NetView program uses the QSAM GET macro to perform read operations and the QSAM PUT macro to perform write operations. See the appropriate QSAM documentation for more information about the QSAM GET and the QSAM PUT. If the NetView program is running under z/OS Version 1.10 or earlier, and if a QSAM read operation is performed on a newly allocated data set that is not managed by SMS before any write, the read operation might return residual data
from the previously deleted data set. If this previously deleted data set had a
different record size, the QSAM read operation fails with message
DWO970I QSAM : GET FAILED WITH RETURN CODE 1006

Message DWO050E is also logged.
To avoid these problems, you can perform one of the following steps:
– Write a blank line to the data set before doing a read operation
– Manage the data set with SMS
– Start the NetView program under z/OS Version 1.11 or higher

**Return Codes**

The following return codes are returned by the QSAM stage, on the secondary
output stream, as a signed, 10-digit decimal numbers:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>QSAM completed successfully.</td>
</tr>
<tr>
<td>4</td>
<td>APPEND failed on a PDS member.</td>
</tr>
<tr>
<td>8</td>
<td>Error detected when attempting to access the data set. For example, a partitioned data set with no associated member name was requested.</td>
</tr>
<tr>
<td>12</td>
<td>The user is not authorized to the data set.</td>
</tr>
<tr>
<td>16</td>
<td>An open error occurred. Look for message IEC143I on your system console for more information.</td>
</tr>
<tr>
<td>20</td>
<td>An ABENDx13 occurred while trying to open the data set. See the system console for messages issued relating to this ABEND.</td>
</tr>
<tr>
<td>28</td>
<td>The data set is unavailable. It might 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>36</td>
<td>The NetView program does not support writing to this data set. For example, the data set might have been defined with RECFM(U).</td>
</tr>
<tr>
<td>40</td>
<td>A record with an incorrect length was encountered while reading the contents of the data set.</td>
</tr>
<tr>
<td>69</td>
<td>A syntax error was detected.</td>
</tr>
<tr>
<td>100</td>
<td>An internal failure or abend occurred.</td>
</tr>
</tbody>
</table>

**Example: Reading from a Data Definition**

The following reads data from a data set specified by a data definition:

```
PIPE QSAM (DD) allocddd
    ...
```

where allocddd is the 'FILE' value from the ALLOCATE command.

**Example: Reading from a Data Set Name**

The following reads 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)'.

Chapter 2. Pipeline Stages and Syntax  179
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 causes 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 writes data from the input stream to a data set specified by a data definition:

```
PIPE ...
       QSAM (DD) allocddd
       ...
```

where allocddd is the 'FILE' value from the ALLOCATE command.

Example: Writing to an Existing Data Set Name

The following writes 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:
```

Synonyms

```
Stage Name       Synonym
REVERSE          REV
```

```
Stage Operands       Synonym
LINE                L
MESSAGE              M
STREAM              S
```
Command Description

The REVERSE stage can be used to reverse message text, multiple line write-to-operator (MLWT0) 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 19601ISD. 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.

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 DSVSMX 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:NT7E002 STATUS:NOT ACTIVE
--> TYPE: OST TASKID: TOM RESOURCE:NT7E001 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: DSILCPR RESOURCE:DSILCPR STATUS:ACTIVE
--> END OF STATUS DISPLAY
--> SWITCH DSILOG,$

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 REVISRPT

Syntax

REVISRPT:

Command Description

The REVISRPT stage command creates a report of the action of the active revision table for a simulated message.

The input message is submitted to the active revision for simulated revision. The simulation is not perfect because of the following reasons:

- Some WQE fields from the original message are not recorded in the NetView message AIFR.
- Other message alterations from MPF and other subsystem operations are not reproduced. The simulation also ignores the NETONLY action of the revision table.

The revised message is written to the primary output stream. If connected, the secondary output stream receives a BNH informational message, reporting the conditions satisfied by the simulated message.

Termination Conditions

The REVISRPT stage command ends when either its primary input or primary output stage disconnects; a secondary output stream is optional.

Usage Notes

1. The REVISRPT stage cannot be a first stage.
2. Processing of the REVISRPT stage occurs asynchronously. Therefore, use the CORRWAIT stage to allow the results to be processed by subsequent pipe stages.

PIPE ROUTE

Syntax

ROUTE:

ROUTE

ROUTE: label: AUTHRCVR BULLETIN
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, see the RMTSYN statement in the CNMSTYLE member.

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:) or two-part syntax (rmtalias/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.
PIPE ROUTE

• A percent sign (%) can be used in place of the oper_id to indicate to send the message to the authorized receiver at the target domain, as indicated by the netid and luname.

• An exclamation (X'5A') can be used in place of the oper_id to indicate that the message becomes a bulletin at the target domain, as indicated by the netid and luname.

AUTHRCVR
Indicates the messages are sent to the authorized receiver in the local domain.

BULLETIN
Indicates a copy of the message is sent to every operator and autotask currently logged on. A copy of the message is also issued to every operator who subsequently logs on.

You can cancel a bulletin message by issuing a DOM command against it. For example, from an operator station that still has a copy of the message (ABC123E), issue the following command:

```
PIPE HELDM | LOCATE /ABC123E/ | NETV DOM CURMSG
```

Usage Notes
• To send messages to a remote domain over IP, the IP address of the domain must be defined using the RMTSYN statement in the CNMSTYLE member.
• The authorized receiver is determined by the individual message. For more information, see the online help for the ASSIGN command (PRI and SEC keywords).

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 */
| ' COLLECT', /* MLWTO ->"route one message to all"*/
| ' DUP *', /* make copies until next stg disc */
| ' A:' /* feed msgs with labels up to ROUTE */
```

Example: Sending a Message to the Authorized Receiver

To send a message to the authorized receiver at CNM02, enter:

```
PIPE LITERAL /ABC123I more and more/ | ROUTE CNM02/%:
```
PIPE SAFE

Syntax
SAFE

Command Description

A SAFE is a place to store one or more messages that are associated with a command procedure. You can use the SAFE stage 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 is red. Moreover, a DOM that matched the original message matches the retrieved copy.

The PIPE SAFE stage is similar to the PIPE KEEP stage; 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 KEEP stage, see "PIPE KEEP" on page 137.

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 can 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.
PIPEC SAFE

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 is added after data that 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 used.

name

Specifies the 1- to 8-character name of a named SAFE. The value for the name is case sensitive and has the following restrictions:

• It cannot start with any of the following characters:
  - comma (,)
  - parenthesis (( or ))
  - equal sign (=)
• It cannot start with a digit (0 - 9).
• It cannot end with a comma (,).
• It cannot contain blanks.
• The stage separator, END, and ESCAPE characters can be used, if escaped.

All other characters, including uppercase and lowercase letters, are valid.

When the command procedure group ends, all named SAFEs that are created by the group are removed 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.
• Because 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.

Example: Determining Whether a Named SAFE Exists

Use the following test to determine if a given named SAFE (MYSAFE) exists.

```rexx
/* 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 example shows how to create a named SAFE which contains a NULL message.

```rexx
/* 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.

```rexx
PIPE LITERAL /Message created by outer PIPE/
    NETVIEW PIPE (STAGESEP %) SAFE *
        % LITERAL /Message created by inner pipe/
        % COLLECT
        % CONSOLE
    % COLLECT
    % CONSOLE
```

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.

```rexx
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:
PIPE SAFE

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.

PIPE SEPARATE

Syntax

SEPARATE:

SEPARATE SEQUENCE DATA

Synonyms

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEPARATE</td>
<td>SEP</td>
</tr>
</tbody>
</table>

Command Description

The SEPARATE stage 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 is 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.
• 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.

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:

```bash
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.

```
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.

```
PIPE (END %)
  | ...
  | /* stages creating input stream */
  | A: SEPARATE DATA
  | ...
  | /* stages processing data lines */
  | %A:
  | ...
  | /* stages processing control/label lines */
```

### PIPE SORT

#### Syntax

```
| SORT
| PAD '00'X | A |
| PAD 'nn'X | D |
| TIME | A |
| D |
```

#### Command Description

The SORT stage 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 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 are sorted in ascending order. That is, messages where the sort fields are lower EBCDIC values are output before those with higher EBCDIC values.

D  Specifies that the messages are sorted in descending order. That is, messages where the sort fields are higher EBCDIC values are 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 are:
oneA
numb

And, the messages are 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'nn'

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 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:

PIPES CAN BE FUN!

This ... Results in ...

7.6   CAN BE
9.20  N BE FUN!
8     A
PIPE SORT

Up to eight sort fields can 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.

**TIME**

Specifies that messages are to be sorted by time, based on the message IFRAUGMT value.

**Usage Notes**

- SORT cannot be the first stage.
- 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.

**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
Ano h es

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. Because they are different (ge and me), the two records can be sorted appropriately. The messages are passed to the output stream as follows:

Another message
One message
One more message

---

PIPE SPLIT

**Syntax**

**SPLIT:**

```
| AT | / | /
| AT | /
| / | / | /|char|
| charcnt | / | /|WITHPFIX—|/string/|
| 0 | AT | / | / |
| charcnt | AT | / | / | / | / | /|ANYOF |
| string | /
| AT | /
| AFTER |
| BEFORE |
```
Synonyms

<table>
<thead>
<tr>
<th>Stage Name</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 are to be split, use SEPARATE 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 is 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 is a match.

  See also the STRING keyword.

AT
- The line is split where the specified string or character is found. The matching string or character is deleted from the output.

BEFORE
- The line is 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.

char
- A delimited character. SPLIT searches for /char/ and splits the input line at each occurrence. The default value for /char/ is a single blank.

  A single character or blank must be enclosed within the delimiters. /char/ cannot be a null string (//).

  See also the AT keyword.

charcnt
- For the AFTER and BEFORE options, 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 can be a positive number, negative number, or zero (0). The default value is zero (0). Valid values for charcnt are in the range -10000000 to +10000000. For the WITHPFIX option, indicates the split point for a line that is longer than charcnt characters. The line is split every charcnt characters, into two or more lines. For a line that is equal to or shorter than charcnt characters, no action is taken. When charcnt is specified alone with no other parameters or when it is specified with the the WITHPFIX option, the value of charcnt must be a positive number. Valid values are +1 to +10000000.

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.

For the AFTER and BEFORE options, SPLIT searches for /string/ as indicated by the ANYOF or STRING keyword and splits the input line at each occurrence. The default value for /string/ is a single blank.

For the WITHPFIX option, /string/ is required. After the input line is split into two or more lines, the delimited string is appended to the front of each line except the first line.

One or more characters must be enclosed within the delimiters. /String/ cannot be a null string (/ /).

WITHPFIX
An input line that is longer than charcnt characters is split every charcnt characters into two or more lines. After the first line of output, each line starts with the string that is specified in the required /string/ argument, so that you can easily rejoin the data by using the JOINCONT LEADING stage. Every line before the last line contains exactly charcnt characters from the input data. The output for each single input line is formatted as a multi-line message.

Tip: If you have to edit the lines to pad the output data to a fixed length, use the PIPE SEPARATE stage. To be properly rejoined with the JOINCONT stage, the lines must also be separated.

Example: Splitting at Blanks

The following splits the literal string /HERE IS SOMETHING TO SPLIT/ at each blank:

```plaintext
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

Syntax

SQL

SelectStatement:
EXECUTE DESCRIBE SELECT string

InsertStatement:
EXECUTE INSERT INTO word

ReleaseStatement:
EXECUTE RELEASE locationname CURRENT SQL ALL PRIVATE
PIPE SQL

SetConnection:

| EXECUTE | SET | CONNECTION | locationname |

SetCurrent:

| EXECUTE | SET | CURRENT | PACKAGESET | USER packagesetname |
| DEGREE | 1 | ANY |
| RULES | DB2 | STD |
| SQLID | USER | value |

Command Description

The SQL stage queries DB2 tables, inserts rows into DB2 tables, and issues DB2 commands.

For additional information about interacting with SQL databases, see Chapter 6, “Using Tivoli NetView for z/OS SQL Stages for Access to DB2,” on page 299.

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 halfwords 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 halfwords 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**

The *word* specifies the plan to use.

The CNMSJSQL sample contains the plan name for your level of the NetView program. 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 provide 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 IBM Software Support.

**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 can 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 a NetView system, 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 Notes

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 cannot 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 quotation mark (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 items:

- 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 quotation mark (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. See 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

Definitions of parameters match the SQL language. Because these statements cannot be executed using dynamic SQL, the NetView program provides explicit code support for them. These statements can be used in a "PIPE SQL EXECUTE" on the primary input stream as is done for other functions.

Using SQL LISTREGS

This function, which is provided with the NetView program, lists the values in the DB2 special registers. It provides the following output:

-----------------------------------------------------------------------
* NT98 PIPE SQL SSID DB2 LISTREGS | CONS
* NT98
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; see 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.

```sql
/* 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. Use the NOCOMMIT option 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 a NetView system, 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 interfaces 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.

The plan closes when none of the SQL stages within the pipe specify NOCLOSE. For example, PIPE SQL COMMIT WORK | CONSOLE commits 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 | CONSOLE at the start of SQL processing to ensure that any previous plan is closed. Alternatively, use PIPE SQL NOCLOSE CONNECT RESET | CONS to ensure that 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 might 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 terminates when the primary output stream is disconnected. It also terminates 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 67 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.

```haskell
/* 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. CNMSJSQ is the input to the preparation process; it is shipped with the NetView JCL samples. Your database administrator gives privileges to the NetView system using the GRANT statement.

- Refer to your DB2 documentation for more information about preparing the NetView SQL stages plan (DSISQLnn in sample CNMSJSQ).

- 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.

---

**PIE SQLCODES**

**Syntax**

```
SQLCODES:
```

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Name</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.
PIPED SQLCODES

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.

PIPED STEM and PIPE $STEM

Syntax

STEM

```
STEM
  (0)
  (number)
  (COMMON)
  (TASK)

FROM 1
  FROM
  APPEND
```

Command Description

The STEM stage 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, VARNAMEn). 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 are created containing messages 1, 2, and 3 respectively, VARNAME0 contains the number 3.

The use of the STEM stage 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 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 is appended as additional STEM variables following STEM data that already exists as determined by the count in element zero. APPEND can be used only on a stage that is not first.

The APPEND option is not enabled if STEM is the first stage. 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. End it 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
PIPE STEM and PIPE $STEM

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 STEM stage. However, you can assign these values to or from other named variables, which you can use in the STEM stage.

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 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 of a pipeline specification, the following conditions apply:
  - The maximum size of a message buffer output from the STEM stage is 32,000 bytes. Message buffers exceeding 32,000 bytes are truncated to 32,000 bytes.
  - If the variable specified on the STEM stage 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 10,000,000.

• When STEM is not the first stage 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 must be specified on the STEM stage. If STEM is not a first stage, a COLLECT stage must precede the STEM stage.

For example, if one task is updating the common global stem \( X \). and your task is reading it, the following example might get some of the updated \( X \). values and some of the older values:

```rexx
PIPE STEM (COMMON) X. | CONSOLE
```

However, the following command gets all of the older values or all of the updated values.

```rexx
PIPE STEM (COMMON) X. COLLECT | CONSOLE
```

- STEM ignores the structure of the messages it receives. Thus, 10 one-line messages set 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 117 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 is null. However, a null value is handled the same way as if the count variable had been set to zero (0).

### 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:

```rexx
PIPE SAFE *
    | NETVIEW SECND
    | STEM SSLT.
```

### Example: Saving the Count of Records Processed

In this example, the CNMCMDF 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 CNMCMDF.

```rexx
/* REXX COMMAND LIST */
'PIPE < CNMCMDF INCL',
    '  | STEM A'
    "  SAY 'THERE ARE ' A0 ' LINES IN CNMCMDF.'"
```

For examples of using $STEM, see the WINDOW command list (CNME1505).
PIPE STRIP

Syntax

STRIP:

- BOTH
- LEADING
- TRAILING
- BLANK
- TO
- /charset/
- limit

Command Description

The PIPE STRIP stage removes blanks or other specified characters from the beginning or end of message data. Alternately, STRIP removes 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.
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.
- 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 is 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.

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
PPLE

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.
PIPE SUBSYM

Syntax

SUBSYM:

---

Synonyms

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSYM</td>
<td>SUBS</td>
</tr>
</tbody>
</table>

Command Description

The SUBSYM stage takes messages in the pipeline and substitutes any MVS or user-defined system symbolics, including the &DOMAIN symbolic supplied with the NetView product, 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.
- Substitution is performed on the &DOMAIN symbolic, unless substitution was disabled when the NetView program 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 the NetView program was started
  - If you have not defined an MVS system symbolic on your MVS system
PIPE TAKE

Syntax

TAKE:

Command Description

Use the TAKE stage 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 - 10,000,000. 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.
- 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.
**PIPE TAKE**

- 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.

**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:

```
PIPE VTAM DISPLAY NET,APPLS
  CORRWAIT 10
  TAKE 2
  TAKE LAST 1
  CONSOLE
```

**PIPE TOSTRING**

**Syntax**

```
TOSTRING:

- ALL
- FIRST
- LAST
- INCL
- NOINCL
- position.length
- /string/
- BLANK
- NULL
```

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Name</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 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. You can specify the letter $ for the length if the specification is followed by a delimited string. The TOSTRING stage replaces the letter with the length of that delimited string.

/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.

BLANK
Specifies that the character string for which to search contains only blanks. The search occurs in the range specified by the position.length parameter, but if the data contains only blanks, a match is recognized regardless of the length specified.

NULL
Specifies that the stage is to search for no data whatsoever, that is, null data (not even blanks). This means that a match is recognized only when the data is shorter than the number specified for position in the position.length parameter.

Usage Notes

• TOSTRING cannot be the first stage.
• If the delimited string is longer than the length specified on the TOSTRING stage, no matches occur and all messages are discarded from the pipeline.
• TOSTRING is a terminating stage, meaning that when a match is found, processing for this stage ends along with any outstanding CORRWAIT.
• You can specify the position.length and /string/ pair up to 40 times.

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
terminates after it has processed the message containing the string /IST314I/. This termination disconnects the output stream of CORRWAIT and causes the wait to end.

`PIPE (NAME ENDSOON)', /* Issue VTAM display command */
`VTAM D NET,LINES', /* Wait up to 50 seconds for response*/
'CORRWAIT /IST314I/', /* Last line expected: IST314I END */
'TOSTRING /IST314I/', /* (for example) */
'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.

**PIPE TSO**

**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 is executed and the results returned. Specify CORRWAIT to 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 is 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 is passed to the TSO server for execution. The message data is contained in a sequential file `userid.NVCMDIN` that is allocated as DD NVCMIDIN. NVCMIDIN 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 can 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 is 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 14 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 are disconnected. If followed by CORRWAIT, CORRWAIT automatically ends when the TSO command is completed and the response has been received.
Operands 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 is invoked once if the TSO stage has an input stream. Otherwise, the command is 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 delays 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, code the following statements:

```
PROTECT *.*.DSIPITSO
PERMIT BOSS *.*.DSIPITSO
```

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, code the following statements:

```
PROTECT *.*.DSIPITSO.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, code the following statements:

```
PROTECT *.*.DSIPITSO.TSOSERV.USER1/SERVJOB1
```

See 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 +000000100 PPI.

The keyword can be one of the following keywords:

AUTHCHK
An error occurred during authorization checking. AUTHCHK is followed by a DSIKVS return code indicating the error. See IBM Tivoli NetView for z/OS Programming: 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 canceled.
+0000000104
   Indicates that a user abend occurred in PPI processing.

See the IBM 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 incorrect 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 cannot be established with the PPI.
In the following example, the TSO stacks serving a TCP user are listed:

```plaintext
/* 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 TCP stack name or other mnemonic used */
/* as a member name (copy of CMMSJTSO) for */
/* 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 NETVASS
arg theUser
IF words(theUser) <> 1 THEN
   DO;
      say 'User ID required'
      EXIT 12
   END;
ELSE theUser = left(theUser,8)

'PIPE (NAME SRVRLIST)',
'NETV LIST STATUS=TSOSERV', /* Obtain list of all servers. */
'SEPARE DATA', /* sep & discard label lines. */
'SORT 21.8', /* sort on member (= stack name). */
   'LOCATE? See NOTE 1, below */
'DELDUPES KEEPFIRST 21.8', /* keep one of each. */
   'EDIT WORD 2 NEXT', /* build record: tso userid and */
   'WORD 3 NEXTWORD', /* member name from each line. */
'STACKS.' /* save these to feed TSO stage */
   'on its secondary input, below.*'

'PIPE (NAME MULTSTAT END %)',
'LITERAL /NETSTAT/ ', /* creating a command */
   'DUP *', /* make copies indefinitely: NOTE2*/
   'A: TSO', /* read destinations from A below*/
   'WAIT 90', /* wait plenty */
   'SEPARE* ', /* can't use SEP DATA */
   'LOCATE 1.10 /EZA0185I', /* get just the data lines */
   'LOCATE 10.10 /'theUser/', /* and lines with our user's name*/
   '/* Now that we have data about our user*/
   '/* where did it come from? The answer */
   '/* in the attributes: JOBID */
   'EDIT JOBID 1 ', /* Build msg with member name, */
```

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 TCP stack name or other mnemonic used */
/* as a member name (copy of CMMSJTSO) for */
/* 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 NETVASS
arg theUser
IF words(theUser) <> 1 THEN
   DO;
      say 'User ID required'
      EXIT 12
   END;
ELSE theUser = left(theUser,8)

'PIPE (NAME SRVRLIST)',
'NETV LIST STATUS=TSOSERV', /* Obtain list of all servers. */
'SEPARE DATA', /* sep & discard label lines. */
'SORT 21.8', /* sort on member (= stack name). */
   'LOCATE? See NOTE 1, below */
'DELDUPES KEEPFIRST 21.8', /* keep one of each. */
   'EDIT WORD 2 NEXT', /* build record: tso userid and */
   'WORD 3 NEXTWORD', /* member name from each line. */
'STACKS.' /* save these to feed TSO stage */
   'on its secondary input, below.*'

'PIPE (NAME MULTSTAT END %)',
'LITERAL /NETSTAT/ ', /* creating a command */
   'DUP *', /* make copies indefinitely: NOTE2*/
   'A: TSO', /* read destinations from A below*/
   'WAIT 90', /* wait plenty */
   'SEPARE* ', /* can't use SEP DATA */
   'LOCATE 1.10 /EZA0185I', /* get just the data lines */
   'LOCATE 10.10 /'theUser/', /* and lines with our user's name*/
   '/* Now that we have data about our user*/
   '/* where did it come from? The answer */
   '/* in the attributes: JOBID */
   'EDIT JOBID 1 ', /* Build msg with member name, */
```
'word 2 NW ', /* user's name, and */
'word 6 NW ', /* current state. */
' CON ', /* */
' TAKE LAST 1', /* IS there a last one? */
' PIPEND 2', /* IF so, make rc = 2 */
'/* ------ end of main pipeline ------ */
'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 can 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 secondary input stream (servers to use), it */
/* will accept as many commands as there are servers feed */
/* to it. After the secondary input stream disconnects, */
/* TSO stage disconnects and the copying ends. */
/* ------------------------------------------*/

---

**PIPE TROUTE**

**Syntax**

**TSROUTE:**

```
|---TSROUTE---|
```

**Synonyms**

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>TROUTE</td>
<td>TSR</td>
</tr>
</tbody>
</table>

**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.
- 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.

Example: Automation Table Sample

See sample DSIAPML for an example of TSROUTE.

PIPE UNIX

Syntax

UNIX:

```
|UNIX| (|ECHO|) |unix_command|
```

Command Description

The PIPE UNIX stage transfers a command to a NetView UNIX server where the command is executed and the results returned. Specify CORRWAIT to 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 are disconnected. If UNIX is followed by CORRWAIT, CORRWAIT automatically ends 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 is invoked once if the UNIX stage has an input stream. Otherwise, the command is 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, code the following:

```plaintext
PROTECT *.*.DSIPIUNX
PERMIT BOSS *.*.DSIPIUNX
```

UNIX commands are submitted under the UNIX user name equivalent to the NetView operator ID. The command is 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 is 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 canceled.
Indicates that a user abend occurred in PPI processing.

For information about other codes, see the IBM Tivoli NetView for z/OS Application Programmer’s Guide.

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.

-0000000005
Indicates that an internal error during open pipe (open) processing caused UNIX command processing to terminate.

-0000000006
Indicates that an internal error during spawn (spawn) processing caused UNIX command processing to terminate.

-0000000007
Indicates that an internal error during write to pipe (write) processing caused UNIX command processing to terminate.

-0000000008
Indicates that an internal error during read from pipe (read) processing caused UNIX command processing to terminate.

-0000000009
Indicates that an incorrect UNIX command was sent.

-0000000011
Indicates that an internal error during get user information (getuid) processing caused UNIX command processing to terminate.

-0000000012
Indicates that an internal error during set user identity (setuid) processing caused UNIX command processing to terminate.

-0000000013
Indicates that an internal error during set user’s group (setgid) processing caused UNIX command processing to terminate.

-0000000014
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.

Indicates that an error return code was returned as a result of invoking the SIGACTION system service.

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
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
   | WAIT 99
   | COLOR GREEN
   | CONSOLE ONLY+
   | A: COLOR YELLOW
   | CONSOLE ONLY
   ```

3. Run the HelloWorld executable:

   ```
   NetVAsIs PIPE (END +)
   A: UNIX java HelloWorld
   | WAIT 99
   | COLOR GREEN
   | CONSOLE ONLY+
   | A: COLOR YELLOW
   | CONSOLE ONLY
   ```

PIPE VAR and PIPE $VAR

**Syntax**

```
VAR and $VAR:

VAR $VAR  
   (O)  
   (COMMON)  
   (TASK)  
   (number)  

name
```
Command Description

The VAR stage can be used anywhere in the pipeline specification.

When VAR is the first stage, 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, lines are read from its input stream and are written to both the specified variables and to its output stream. The VAR stage ignores the multiline nature of the input. VAR stores data from the first line input into the first variable named, the second line into the second variable, and so on. All messages are passed unaltered to the secondary output, if connected.

When all specified variables have been assigned, VAR writes subsequent messages directly to its output stream.

The use of the VAR stage 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 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.
### PIPE VAR and PIPE $VAR

#### 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. However, you can assign these values to, or from, other named variables, that you can use in the VAR stage.

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 can 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 conditions apply 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 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 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 has not been initialized, the output value is a null message (an automation internal function request with a single, zero-length message buffer).

- When VAR is not the first stage of a pipeline specification, the following conditions apply:
  - The variable specified on the VAR stage 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 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 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.
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'
```

Example: Using the STAGESEP Stage for DBCS Problems

You can use the following command for character problems encountered in DBCS:

```rexx
DATA1 = sql statements including x'4Fxx'
'PIPE (END %) LITERAL /' DATA1/' | ------ ' ====> results: BAD
'PIPE (END %) VAR DATA1 | EXCUTE ------ ' ====> results: GOOD
```

PIPE VARLOAD

Syntax

```plaintext
VARLOAD:
```

<table>
<thead>
<tr>
<th>VARLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
</tr>
<tr>
<td>(COMMON)</td>
</tr>
<tr>
<td>(TASK)</td>
</tr>
<tr>
<td>(number)</td>
</tr>
</tbody>
</table>

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 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.

Note:
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 items are written to the secondary output stream:
- All input messages with errors in the variable name
- If data is in the form *variable1=variable2*, all messages where *variable1* does not equal *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 Description

**COMMON**

Specifies *variable1* is accessed from the common global variable dictionary instead of the local variable dictionary.

**TASK**

Specifies *variable1* is accessed from the task global variable dictionary instead the local variable dictionary.

**(number)**

Specifies the number invocations (generations) to refer back to when accessing *variable1*. The number of generations refers to the current nesting level within the REXX, PL/I, or C calling sequence. ENVDATA can be used to determine the number of generations.

*(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.

The default for *(number)* is zero (0).

### Usage Notes

- VARLOAD does not delay the stream.
• Variable1 and 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 are 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 STRIP with the TRAILING keyword prior to VARLOAD.
• All data after the second delimiter is assigned to the variable regardless of intervening blanks or subsequent delimiters.

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 COPYGLB)',
  '  | CORRCMD /opid*: QRYGLOBL TASK VARS=TCP*, /* opid*/
  '  | SEPARATE',
  '  | LOCATE 1.7 /BHN039I/', /* just the DATA please */
  '  | EDIT "/* NEXT', /* change format to suit*/
  '  | "/* NEXT',
```
Example: Update Current Group Members

The following process updates 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 of VARLOAD*/
UpDate.2 = '/currGrp=currGrp/' || currGrp
UpDate.0 = 2

/* Update.2 contains multiple references to currGrp. */
/* CurrGrp is being used in three ways: the first references */
/* the common global dictionary (see options 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. */
/* */
/* Updates of variables from a multi-line message are made */
/* together. All comparisons for the lines of the message */
/* are done before any updates. If any comparison fails, */
/* no updates are done from that MLWTO. Therefore we collect*/
/* related updates into one multi-line message. */
/* */

'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 VERIFY**

**Syntax**

```
VERIFY: VERIFY | position.length | NOT | /string/
```

**Command Description**

The VERIFY stage examines the first line of each message in the range specified. If all the characters examined are also in the string specified, then the message is written to the primary output stream. Otherwise, the message is written to the secondary output stream, if connected.

When NOT is specified, VERIFY verifies that the characters examined are not in the argument string.

**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**

VERIFY terminates when the input stream or both output streams are disconnected.
Operand Descriptions

position.length
Specifies the character position where examination is to occur within the line. Only the first line of each message is examined. If you specify a length of *, the remainder of the message is searched. If you do not specify a position.length, the entire line is examined.

NOT
Specifies the characters that are considered as matching those not in the /string/ specification.

/string/
Specifies the characters that are considered as matching during the examination of the input line. When NOT is also specified, VERIFY verifies that the characters examined are not in the argument string.

Duplicate characters can be specified, but are not significant.

The first nonblank character encountered after the stage name, position.length, or NOT 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

VERIFY cannot be the first stage.

Example: Separating Domain Names from IP Addresses

In the following example, it is assumed that, because IP addresses consist of the digits 0 - 9 and periods, any value having other characters must be a domain name:

```
PIPE (END %) STEM allValues.
 XX: VERIFY 1.15 /0123456789/
 STEM ipAddrs.
 % XX:
 STEM domNames.
```

### PIPE VET

**Syntax**

**VET (first stage):**

```
<table>
<thead>
<tr>
<th>VET</th>
<th>NEXT</th>
<th>ROWS</th>
<th>--NAME-- only one</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CURRENT</td>
<td>FIELDS</td>
<td>--NAME-- attach name</td>
</tr>
</tbody>
</table>
```

**VET (stage other than first):**

```
<table>
<thead>
<tr>
<th>VET</th>
<th>cursor position</th>
<th>ENTER</th>
<th>--NAME-- only one</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW.COL</td>
<td>action key</td>
<td>--NAME-- attach name</td>
<td></td>
</tr>
</tbody>
</table>
```
VET (command):

VET [cursor position] ROW.COL/STRING/ ENTER action key

NAME—only one
NAME—attach name

Synonyms

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>VET</td>
<td>VOSTIO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stage Operands</th>
<th>Synonym</th>
</tr>
</thead>
<tbody>
<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.

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>

Chapter 2. Pipeline Stages and Syntax 231
PIE VET

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:

<table>
<thead>
<tr>
<th>Key</th>
<th>Key</th>
<th>Key</th>
<th>Key</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF1</td>
<td>PF2</td>
<td>PF3</td>
<td>PF4</td>
<td>PF5</td>
</tr>
<tr>
<td>PF6</td>
<td>PF7</td>
<td>PF8</td>
<td>PF9</td>
<td>PF10</td>
</tr>
<tr>
<td>PF11</td>
<td>PF12</td>
<td>PF13</td>
<td>PF14</td>
<td>PF15</td>
</tr>
<tr>
<td>PF16</td>
<td>PF17</td>
<td>PF18</td>
<td>PF19</td>
<td>PF20</td>
</tr>
<tr>
<td>PF21</td>
<td>PF22</td>
<td>PF23</td>
<td>PF24</td>
<td></td>
</tr>
<tr>
<td>PA1</td>
<td>PA2</td>
<td>PA3</td>
<td>ENTER</td>
<td>CLEAR</td>
</tr>
</tbody>
</table>

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 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.

**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.

**Note:** For any field of data that has an intensity attribute of DARK (ID), the entire returned field information has presentation attribute of DARK. As a result, the data cannot be displayed. One way to see the data is to use a COLOR NORMAL stage subsequent to the VET stage.

**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, that is, a VOST created outside of the procedure family.

NAME must be specified if VET is to interact with an independent VOST, that is, 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. Specify CORRWAIT as 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 49.

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 can 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 contain X'FF' characters.

**Note:** For any row of data that contains any field with an attribute of DARK, the returned row has a presentation attribute of DARK. As a result, the data is not displayed. One way to see the data is to use a COLOR NORMAL stage subsequent to the VET stage.

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 can return 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 accepts 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 (/ /) does not need to 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 statement:
    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, the NetView program 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, the NetView program retains the last data transmitted by the application that was running on the VOST for up to 5 minutes. A VET NEXT 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.

**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.

```
input. = '
input.5 = 'X'
input.0 = 5

PIPE (NAME REQNCP)
STEM input.
VET 1.1
```

---

**PIPE VTAM**

**Syntax**

VTAM:

```
VTAM: (CGI, NOPANEL, MOE) cmdtext
```

**Command Description**

The VTAM stage runs the VTAM commands DISPLAY, VARY, and MODIFY in the local 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:

```
DSI071I 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 might 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.
When NOPANEL is specified, the command does not display a full-screen panel. If it attempts to do so, message BNH113W is inserted into the pipeline and the command receives 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 IBM Software 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.

**Example: Issuing a VTAM Command**

Suppose you want to analyze the status of applications at 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, issue the following command procedure:

```plaintext
PIPE VTAM D NET,APPLS
   CORRWAIT 60
   NLOCATE 1.7 /IST097I/
   TAKE 1
   SEPARATE
   LOCATE /CONCT/
   STEM appldata
```

**Note:** In a REXX command procedure, the last stage can be STEM appldata. with a period (.).
PIPE XCFMSG

Syntax

XCFMSG:

XCFMSG group_name .member_name

Command Description

The XCFMSG stage sends and receives messages using z/OS XCF signaling services. Sending one-way messages and receiving unsolicited messages is supported.

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 XCFMSG stage terminates when the primary output stream is disconnected.

Operand Descriptions

group_name  
 specifies the name of the group on which to listen, or the group that contains the member to whom a message is being sent. The value can be 1 – 8 characters in length.

member_name  
 specifies the name of the member to receive the message. For DSIPLXnn groups, the member name must be the NetView domain name.

Usage Notes

• The XCFMSG stage accepts one input stream. If it is connected, the stage sends out the input message. If it is the first stage in the pipeline, the pipe receives messages from the specified group. The member_name option is not valid for receiving messages.
• For sends, the primary output is the message sent. For receives, the primary input is the received message, which is the second line of the multiline BNH590I message.
• If a multiline message is passed to the stage, separate XCF sends are done for each line of the message. You can use the JOINTCONT stage command to consolidate a multiline message and then use the SPLIT stage command in the receiving pipeline to preserve multiline messages.
• XCF messages can only be sent to or received from XCF groups to which the NetView program belongs.
• Only one XCF receive PIPE stage can be outstanding for a group at any given time.
Example: Sending Messages to an XCF Group Member

The following example sends a message to the NETVA member in the NetView group:

```
PIPE lit /test message/ | XCFMSG DSIPLX01.NETVA
```

Example: Receiving Messages from a NetView Group

The following is an example of receiving messages from the main NetView group:

```
PIPE XCFMSG DSIPLX01
   | WAIT *
   | CONSOLE
```

---

PIPE XCFQUERY

Syntax

```
XCFQUERY:
```

![Diagram of XCFQUERY syntax]

Command Description

The XCFQUERY stage retrieves sysplex data using the z/OS XCFQUERY programming interface. The retrieved data, with one exception, is mapped by z/OS IXCQUAA structures.

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

The XCFQUERY stage terminates when either its input stream or output stream disconnects.

Operand Descriptions

- **CF** Returns a list of coupling facilities in the sysplex. The CF keyword has the following optional keyword:

  - **cf_name**
    
    Returns detailed information about the name coupling facility, including a list of systems connected to the coupling facility. The value for `cf_name` can be 1 – 8 characters in length.
**CFGINFO**

Returns basic XCF information about the sysplex. The following information is returned:

- The maximum number of systems that can be in the sysplex
- Whether the sysplex is configured as LOCAL or MONOPLEX
- The XCF system identifier
- The XCF token for the sysplex
- The node descriptor for the system
- The name of the sysplex
- The maximum coupling facility level supported

**GROUPS**

Returns a list of groups in the sysplex. The following optional keywords are valid:

- `group_name`
  Returns detailed information about the specified group and its members. The value for `group_name` can be 1 – 8 characters in length.

- `member_name`
  Returns detailed information about the specified member. The value for `group_name` can be 1 – 16 characters in length. This option is only valid if `group_name` is specified.

**SYSTEMS**

Returns a list of systems in the sysplex.

**Usage Notes**

- Query failures generally produce return codes from the IXCQUERY macro that are returned in the CNM273I message. Return codes are documented in the z/OS V1R9.0 MVS Programming: Sysplex Services Reference, SA22–7618.
- The XCFQUERY stage command returns the query results to the primary output stream.
- The output of the XCFQUERY stage is unformatted data (including binary data) that is returned by the z/OS IXCQUERY macro instruction. You can use the EDIT stage to reformat the data to character data. The CF, GROUPS, and SYSTEMS output is mapped by the QUASYS1, QUAGRP, QUAMEM1 and QUACF1 mappings in the IXCYQUAA z/OS system macro. The CFGINFO output consists of the data in Table 9 (in the order listed).

<table>
<thead>
<tr>
<th>Number of bytes</th>
<th>Data</th>
<th>IXCQUERY z/OS system macro parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>maximum number of systems possible in the sysplex</td>
<td>MAXSYS</td>
</tr>
<tr>
<td>4</td>
<td>current maximum number of systems in the sysplex</td>
<td>CURRMAXSYS</td>
</tr>
<tr>
<td>8</td>
<td>sysplex name</td>
<td>PLEXNAME</td>
</tr>
<tr>
<td>4</td>
<td>coupling facility level</td>
<td>CFLEVEL</td>
</tr>
<tr>
<td>32</td>
<td>node descriptor</td>
<td>ND</td>
</tr>
<tr>
<td>8</td>
<td>sysplex token</td>
<td>SYSPLEXID</td>
</tr>
<tr>
<td>8</td>
<td>sysplex token</td>
<td>SYSTEMID</td>
</tr>
<tr>
<td>1</td>
<td>XCF-local mode flag (X'00' if false, X'01' if true)</td>
<td>LOCAL</td>
</tr>
</tbody>
</table>
Table 9. CFGINFO output (continued)

<table>
<thead>
<tr>
<th>Number of bytes</th>
<th>Data</th>
<th>IXCQUERY z/OS system macro parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>monoplex mode flag (X'00' if false, X'01' if true)</td>
<td>MONOPLEX</td>
</tr>
</tbody>
</table>

**Example: Displaying all the XCF Groups in a Sysplex**

The following example displays all the XCF groups in a sysplex:

PIE XCFQUERY GROUPS | CONS

**Example: Displaying all the Members in a Group**

The following example displays all the members in the DSIPLX01 NetView group:

PIE XCFQUERY GROUPS DSIPLX01 | CONS

**Example: Displaying a Member of a Group**

The following example displays the NTVE4 member of the DSIPLX01 NetView group:

PIE XCFQUERY GROUPS DSIPLX01.NTVE4 | CONS

**PIPE XCFTABLE**

**Syntax**

**XCFTABLE:**

```
<table>
<thead>
<tr>
<th>XCFTABLE</th>
<th>MEMBER</th>
<th>STATFLD</th>
<th>USERDAT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>plex_name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>label_name.plex_name.</td>
</tr>
<tr>
<td></td>
<td>group_name.</td>
<td>member_name.</td>
<td></td>
</tr>
</tbody>
</table>
```

**Command Description**

The XCFTABLE stage retrieves and sets the following data:

- state field maintained by the z/OS XCF service for a group member
- user data field maintained by the NetView program for the group member
- table entry for the member

**Usage Note:** This stage is used internally by the NetView program; use this stage only to set data for user–defined groups with the USERDAT or STATFLD options.

**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 XCFTABLE stage terminates when either its input stream or output stream disconnects.

Operand Descriptions

MEMBER
Specifies the member for the actual table entry.

STATFLD
Specifies the state field.

USERDAT
Specifies the user data field.

label_name
For members not in the local NetView's sysplex (which is the case at an enterprise master), specify the label used on the ENTSYSTEMS statement that established contact with the remote member's XCF group. This can be specified as a wildcard of an asterisk (*), which matches any label. This operand is not needed for members within the local NetView's sysplex.

plex_name
For members not in the local NetView's sysplex (which is the case at an enterprise master), specify the name of the sysplex in which the remote system resides. This can be specified as a wildcard of an asterisk (*), which matches any sysplex name. This operand is not needed for members within the local NetView's sysplex.

group_name
Specifies the name of the group. The value can be 1-8 characters in length.

member_name
Specifies the name of the member whose table data is being retrieved or set. An error message is generated if the member name is for a member other than one created by a START XCF command from the local NetView system.

Usage Notes

- The XCFTABLE stage command accepts one input stream. If it is connected, the stage sets the specified field with the input message. For the state field, this can be 1 - 32 bytes in length. This message is passed to the output stream.
- If the XCFTABLE stage command is the first stage, the pipeline returns the data field on the output stream.
- If group_name or plex_name is not unique within the enterprise, use of the wildcard (*) for plex_name can result in errors.

Example: Retrieving the state field in a user-defined XCF group

To retrieve the state field for member CNM01 in the user-defined XCF group SAMPLE in the local sysplex, enter this command:

PIPE XCFTABLE STATFLD SAMPLE.CNM01 | CONSOLE

Example: Retrieving the state field in a NetView XCF group in a local sysplex

To retrieve the state field for member CNM01 in the NetView XCF group DSIPLX01 in the local sysplex, enter this command:
Example: Retrieving the state field using a wildcard character in an enterprise environment

Assume the local NetView system is functioning as an enterprise master, and is managing a remote XCF group DSIPLX02 in sysplex SAMPPLEX. The label on the ENT.SYSTEMS statement is REMPLX1. To retrieve the state field for member CNM02, enter this command:

```
PIPE XCFTABLE STATFLD REMPLX1.SAMPPLEX.DSIPLX02.CNM02 | CONSOLE
```

Note that a wildcard can be used for the label name or the plex name. Any of the following forms would also work:

- `PIPE XCFTABLE STATFLD *.SAMPPLEX.DSIPLX02.CNM02 | CONSOLE`
- `PIPE XCFTABLE STATFLD REMPLX1.*.DSIPLX02.CNM02 | CONSOLE`
- `PIPE XCFTABLE STATFLD *.DSIPLX02.CNM02 | CONSOLE`

---

### PIPE Xlate

**Syntax**

**XLATE:**

```
 XLATE [1.*] [position.length] [UPPER
    - A2E
    - COMMON
    - E2A
    - LOWER
```

**Command Description**

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. 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.

**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:

```
'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:

```
'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 ASCII 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:

'PIPE | VAR linedata | XLATE A2E | VAR linedata'

PIPE < (From Disk)

Syntax

(From Disk):  

```
< DSIPARM. member
  ddname. *
  dsname
```

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 HDRsDOMID then contains the name of the panel.

When `dsname` is specified as a fully qualified data set name within single quotation marks, this stage is processed as a QSAM (read) stage. See the help for the QSAM stage ("PIPE QSAM" on page 176) instead of the help text below for details on how QSAM (read) processing works.

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 the NetView system is to search all standard DDNAMEs for the specified `member` name. The following libraries are searched, if allocated, in the following order:
**PIPE < (From Disk)**

1. DSICLD
2. DSIPARM
3. DSIPRF
4. DSIVTAM
5. DSIMSG
6. CNMPNL1
7. BNJPNL1
8. BNJPNL2
9. DSILIST
10. DSIOpen
11. DSISRC
12. DSARIPT

**ddname**
Specifies the name of a standard NetView DDNAME, such as DSIPARM or DSICLD, from where to read the member. See the BROWSE command help for a list of valid DDNAMEs. When ddname is not specified, the default is DSIPARM. When specifying ddname, a period (.) is used to separate it from the member name. Do not use spaces before or after the period.

**member**
Specifies the 1- to 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 ddname being used.

**INCL**
Specifies that %INCLUDE statements are expanded when the member or file is read, and that "data REXX" statements, if present, are executed.

**Note:** Data REXX files are special REXX programs that send data to an environment external to the data REXX program. For more information on data REXX, see *IBM Tivoli NetView for z/OS Customization Guide* and *IBM Tivoli NetView for z/OS Programming: REXX and the NetView Command List Language*.

**DISKONLY**
Indicates that any member loaded by the INSTORE stage is ignored.

**dsname**
When specified as a fully qualified data set name within single quotes, this stage is processed as a QSAM (read) stage. See the help for the QSAM (read) stage ("PIPE QSAM" on page 176) instead of this "From Disk" help text for details on how QSAM (read) processing works.

**SEQCHOP**
Indicates that the first record is to be examined for sequence numbers in columns 73 - 80. If sequence numbers are found in those columns, all records are truncated after column 72. Do not use this option with the dsname option.

**Usage Notes**
- The < (From Disk) stage must be the first stage.
- Access security for the < (From Disk) stage is provided through the READSEC command. See the *IBM Tivoli NetView for z/OS Security Reference* for information about the READSEC command.
- The < (From Disk) stage uses one or more QSAM read operations. The NetView program uses the QSAM GET macro to perform these operations. See the appropriate QSAM documentation for more information.
- If the NetView program is running under z/OS 1.10 or earlier and if a QSAM read is performed on a newly allocated data set that is not managed by SMS
before any write, the read might return residual data from the previously deleted data set. If this previously deleted data set had a different record size, the QSAM read fails with message

```
DWO970I QSAM : GET FAILED WITH RETURN CODE 1006
```

Message DWO050E is also logged.

To avoid these problems, you can perform one of the following processes:
- Write a blank line to the data set first before doing any read
- Have the data set managed by SMS
- Start the NetView program under z/OS release 1.11

**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>Incorrect INCLUDE operation (%INCLUDE record normally returned unchanged to primary output stream).</td>
</tr>
</tbody>
</table>

**Other** See the return codes documented for the DSIDKS assembler macro. See *IBM Tivoli NetView for z/OS Programming: Assembler* for more information about DSIDKS return codes.

**Example: Reading the Contents of a File into a Pipeline**

To display the contents of the CNMCMD file as a multiline message:

```
PIPE < CNMCMD
   COLLECT
   CONSOLE
```

This example reads lines of the CNMCMD member into the pipeline. The COLLECT stage builds these lines into a multiline message. When all lines are read from CNMCMD 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.
PIPE > (To Disk)

Syntax
(To Disk):

\[
> \text{name}
\]

Command Description

The > (To Disk) stage functions as a QSAM stage, as long as \text{name} is enclosed in single quotation marks, and there is an input stream. If \text{name} is not enclosed in single quotation marks, or if there is no input stream, a syntax error is reported. For any other QSAM function, use the QSAM stage explicitly.

Data received on the input stream is written to the specified name.

See “PIPE QSAM” on page 176 for more information.

Streams

See “PIPE QSAM” on page 176 for the streams supported by the QSAM stage.

Termination Conditions

See “PIPE QSAM” on page 176 for termination conditions.

Operand Descriptions

\text{name}

The QSAM data set name to which the input stream data is written.

Usage Notes

See “PIPE QSAM” on page 176 for QSAM usage notes.
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, see IBM Tivoli NetView for z/OS Programming: PL/I and C.

Device drivers are stages 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:

PIPE NETVIEW LIST "" | CONSOLE ONLY

Output from the pipeline follows:
NetView Pipelines Device Drivers

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 HELDMMSG | CONSOLE DELETE

Output showing existing held messages looks like this:

```
NCCF                      NETVIEW              CNM01 OPER6  04/14/10 15:06:33
" CNM01
 IEE104I  15.05.33  99.104  ACTIVITY 973
 JOB5   M/S   TS USERS  SYSSAS  INITS  ACTIVE/MAX VTAM
 000000  000007  000001  00014  00002  00001/00300
 LLA  LLA  LLA  NSW  S  VLF  VLF  VLF  NSW  S
 JES2  JES2  IEFPROC  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  04/14/10 15:07:00
 CNM01
 104I  15.05.33  99.104  ACTIVITY 973
 05   M/S   TS USERS  SYSSAS  INITS  ACTIVE/MAX VTAM
 00  000007  000001  00014  00002  00001/00300
 A  LLA  LLA  NSW  S  VLF  VLF  VLF  NSW  S
 S2  JES2  IEFPROC  NSW  S  MYVTAM  MYVTAM  VTAM  NSW  S
 0  TSO  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 HELDMSG 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 disappears.

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. Study this example 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 !
   CONSOLE
```

Output from the pipeline looks like this:

```
+ CNM01 pipe literal ? this is the ccc message ?  
   console
   literal / this is the bbb message /
   console
   literal ! this is the aaa message !
   console
```

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:

```
NCCF NETVIEW CMN01 OPER6 02/01/10 09:19:28
* CMN01
  | 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 |

| CMN01 | THIS IS THE CCC MESSAGE | (written by console stage # 1) |
| CMN01 | THIS IS THE CCC MESSAGE | (written by console stage # 2) |
| CMN01 | 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 CMN19 ORIGOPER 05/17/10 10:18:43
```

Output from the pipeline on OPER3's screen looks like this:

```
NCCF NETVIEW CMN19 OPER3 05/17/10 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:

```
PIPE HELDMSG | CONSOLE DELETE
```

Output showing existing held messages follows:

```
NCCF NETVIEW CNM01 OPER6 03/20/10 15:06:33
* CNM01
IEE104I 15.05.33 99.079 ACTIVITY 973
JOBS M/S TS USERS SYSAS INITS ACTIVE/MAX VTA
00000 00007 00001 00014 00002 00001/00300
LLA LLA LLA NSW S VLF VLF VLF NSW
JES2 JES2 JESPROC NSW S MVTAM MVTAM VTAM NSW
TSO TSO TCAS OWT S MYESSI MYESSI NETVIEW NSW
MYENV MYENV NETVIEW NSW S
USER2 OWT
* CNM01 MVS D A,L
```

Output of the PIPE command follows:

```
NCCF NETVIEW CNM01 OPER6 02/01/10 15:07:00
* CNM01
104I 15.05.33 99.032 ACTIVITY 973
BS M/S TS USERS SYSAS INITS ACTIVE/MAX VTAM
00 00007 00001 00014 00002 00001/00300
A LLA VLF LLA NSW S VLF VLF NSW S
S2 JES2 JESPROC NSW S MVTAM MVTAM VTAM NSW S
O TSO TCAS OWT S MYESSI MYESSI NETVIEW NSW S
ENV MYENV NETVIEW NSW S
RZ OWT
* CNM01 PIPE HELDMSG | CONSOLE DELETE
```

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:

<table>
<thead>
<tr>
<th>NETVIEW</th>
<th>CNM01 OPER5</th>
<th>03/01/10 09:13:24</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNM01</td>
<td>PIPE LITERAL % JACK BE NIMBLE %</td>
<td>CONSOLE</td>
</tr>
<tr>
<td>CNM01</td>
<td>JACK BE NIMBLE</td>
<td></td>
</tr>
</tbody>
</table>

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
/* 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:

<table>
<thead>
<tr>
<th>NETVIEW</th>
<th>CNM01 OPER5</th>
<th>03/26/10 09:15:10</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNM01</td>
<td>DISPOPID</td>
<td></td>
</tr>
<tr>
<td>CNM01</td>
<td>MY OPID IS OPER5</td>
<td></td>
</tr>
</tbody>
</table>

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:

```rexx
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:

<table>
<thead>
<tr>
<th>NETVIEW</th>
<th>CNM01 OPER5</th>
<th>05/02/10 10:40:10</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNM01</td>
<td>PIPE LITERAL ? THIS IS THE CCC MESSAGE ?</td>
<td>LITERAL / THIS IS THE BBB MESSAGE /</td>
</tr>
<tr>
<td>CNM01</td>
<td>THIS IS THE AAA MESSAGE</td>
<td></td>
</tr>
<tr>
<td>CNM01</td>
<td>THIS IS THE BBB MESSAGE</td>
<td></td>
</tr>
<tr>
<td>CNM01</td>
<td>THIS IS THE CCC MESSAGE</td>
<td></td>
</tr>
</tbody>
</table>

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 CANZLOG, 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 are 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 04/14/10 (92300) COLS 037 114
HOST:HOST1 +1* +2* +3* +4* S:CSR
---4--------5--------6--------7--------8--------9--------10--------11----

PIPEC 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 ACTIVE/Max VTAM
090080 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 MYESSI NETVIEW NSW S
MYENV MYENV NETVIEW NSW S
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:

PIPE NETVIEW MVS D A,L | CORRWAIT 5 | TAKE 1 | CONSOLE
Output from the pipeline follows:

```
NCCF     NETVIEW CNM01 OPER6 05/17/10 17:21:19
* CNM01  PIPE NETVIEW MVS D A,L | CORRWAIT 5 | TAKE 1 | CONSOLE
* CNM01
IEE104I 17.21.12 99.137 ACTIVITY 620
JOBS M/S TS USERS SYSSAS INITS ACTIVE/MAX VTAM
00000 00007 00001 00014 00002 00001/00300
VLF VLF VLF NSW S LLA LLA LLA NSW S
JES2 JES2 IEFPROC NSW S MYVTAM MYVTAM VTAM NSW S
TSO TSO TCAS OWT S MYESSI MYESSI NETVIEW NSW S
MYENV MYENV NETVIEW NSW S
USER2 OWT
```

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:

```
NCCF     NETVIEW CNM01 OPER6 05/17/10 17:05:01
* CNM01  PIPE NETVIEW MOE NONESUCH | CONSOLE
  - CNM01  DSIO021 INVALID COMMAND: 'NONSEUCH'
  - CNM01  DW0369I NETVIEW STAGE (1) HAD RETURN CODE 4
```

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:
NetView Pipelines Device Drivers

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</th>
<th>CMN9 OPER3</th>
<th>02/01/10 09:07:40</th>
</tr>
</thead>
<tbody>
<tr>
<td>00C9</td>
<td>PIPE LITERAL ? LIST STATUS=TASKS ?</td>
<td>NETVIEW MOE</td>
<td>LOCATE /NOT ACTIVE/</td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OPT TASKID: TASKNAME: SQLGTSK STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OPT TASKID: TASKNAME: ALIASAPL STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OPT TASKID: TASKNAME: DSISVRNT STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OPT TASKID: TASKNAME: DSISRQVS STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OPT TASKID: TASKNAME: CNM29VMT STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OPT TASKID: TASKNAME: CNM9B18W STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OPT TASKID: TASKNAME: DS1KREM STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OPT TASKID: TASKNAME: VPOTASK STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OPT TASKID: TASKNAME: DS1QTSK STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OPT TASKID: TASKNAME: DS1RQJOB STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OST TASKID: RESOURCE: A01A443 STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OST TASKID: RESOURCE: A01A4443 STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OST TASKID: RESOURCE: A01A444 STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OST TASKID: RESOURCE: A01A445 STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: OST TASKID: RESOURCE: A01A446 STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: NNT TASKID: RESOURCE: STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: NNT TASKID: RESOURCE: STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: NNT TASKID: RESOURCE: STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>TYPE: NNT TASKID: RESOURCE: STATUS: NOT ACTIVE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 are multiple messages in the pipeline, the NETVIEW stage invokes MSGROUTE once for each message. No copy of the routed message is written to the pipeline output stream. If the stage has an error, the MOE option causes a DWO369I message, with nonzero return code, to be placed in the pipe.
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:

```
CNM01 PIPE VTAM D NET,STATIONS,SCOPE=ALL | CORRWAIT 5 | NLocATE /IST097I/
   TAKE 1 | CONSOLE

IST3501 DISPLAY TYPE = STATIONS
IST393I PU T4/5 MAJOR NODE ADIMPU , 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 system to allow enough time for a response to return.
3. The NLocATE stage discards the IST097I message and passes the next message, an MLWT0, 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:

```
CNM01 PIPE VTAM MOE D NETT,TERMS | CORRWAIT 5 | CONSOLE
- CNM01 DSO71I INVALID VTAM COMMAND
- CNM01 DWO369I VTAM STAGE (1) HAD RETURN CODE 8.
CNM01 IST191I DISPLAY SYNTAX ERROR
```

NetView Pipelines Device Drivers

Chapter 3. NetView Pipelines Device Drivers 259
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 system 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

| CORRWAIT 20 |
| CONSOLE |
```

Output from the pipeline follows:

```
NCCF NETVIEW CNM01 OPER6 04/29/10 14:35:00
* CNM19 PIPE NETVIEW RMTCMD LU=CNM01, PIPE (STAGESEP %) VTAM D NET,CDRMS
% CORRWAIT 10 % CONSOLE | CORRWAIT 20 | CONSOLE
CNM01 IST097I DISPLAY ACCEPTED
CNM01 IST350I DISPLAY TYPE = CDRMS
IST0091 C01CDRMS TYPE = CDRMS SEGMENT , ACTIV
IST4821 C01M ACTIV, SA 1, EL 1, NETID = NETC
IST4821 C02M NEVAC, SA 2, EL 1, NETID = NETC
IST4821 C11M NEVAC, SA 11, EL 1, NETID = NETC
IST4821 A09M NEVAC, SA N/A, EL N/A, NETID = NETA
IST4821 A19M ACTIV, SA 10, EL 3, NETID = NETA
IST4821 A29M NEVAC, SA N/A, EL N/A, NETID = NETA
IST4821 A69M NEVAC, SA N/A, EL N/A, NETID = NETA
IST4821 A99M NEVAC, SA N/A, EL N/A, NETID = NETA
IST4821 B18M NEVAC, SA N/A, EL N/A, NETID = NETB
IST4821 B20M NEVAC, SA N/A, EL N/A, NETID = NETB
IST4821 B24M NEVAC, SA N/A, EL N/A, NETID = NETB
IST4821 B52M NEVAC, SA N/A, EL N/A, NETID = NETB
IST4821 D09M NEVAC, SA N/A, EL N/A, NETID = NETB
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 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 can 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 DW0801 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 is one blank preceding the earlier "is". Remember, Net
| NTVD View is one word. The value of &DOMAIN. is "&DOMAIN." and &NV21
| NTVD is
| NTVD "&NV21.".
| NTVD DOMAIN = C&NV2I.01
| NTVD %INCLUDE CNMSTPWD
| NTVD SuppChar = ?
| NTVD %INCLUDE CNMSTNX
| NTVD TRACE.OPTION=DISP,PSS,QUE,STOR,UEXIT // list options
| NTVD TRACE.MODE = INT // INT, EXT, GTF, or
| NTVD "NONE"
```

Processing steps:
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 'CNMESTMN' member that is associated with ddname DSICLD.

```
PIPE < DSICLD.CNMESTYL | LOCATE /'CNMESTMN'/ | CONSOLE
```

Output from the pipeline looks like this:

```
NCCF       NETVIEW    CMN01 OPER6  04/14/13 15:26:40
  * CMN01  PIPE < DSICLD.CNMESTYL | LOCATE /'CNMESTMN'/ | CONSOLE
  | CMN01  'CNMESTMN' prepArg(arg(1))
```

Processing steps:

1. The < stage reads the records from the member that is named CNMESTYL and is 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 'CNMESTMN' 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 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
********************************************************************
* NETVIEW command list language **
********************************************************************
PIPE NETVIEW DISPFK + 
  SEPARATE + 
  LOCATE /PA3/ + 
  VAR CLVAR1 + 
*
*
PIPE VAR CLVAR1 + 
  LITERAL /PA3 IS:/ + 
  CONSOLE
```

---

262 Programming: Pipes
** NetView Pipelines Device Drivers

* 
***************************************************************************** 
* Write return code information to console and exit command list ** 
***************************************************************************** 
&WRITE RETURN CODE IS &RETCODE
&EXIT 

Output from PIPEVAR command list follows:

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW</th>
<th>CNM01 OPER2</th>
<th>02/01/10 10:01</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNM01</td>
<td>PIPEVAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNM01</td>
<td>PA3 IS:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNM01</td>
<td>DSI6081 PA3 IMMED,IGNORE RETRIEVE AND EXECUTE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C CNM01</td>
<td>RETURN CODE IS 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 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 an 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*/
  ' | TOSTRING LAST \\END', /* Process until 'end' msg */
  ' | STEM bfruse.', /* Put each line into a REXX array called bfruse */
  ' | 0th element has the number of lines in the pipeline */
  ' | 8th element var'bfruse.*' /* Display variables */

do i = 0 to bfruse.0
  say 'BFRUSE.'i 'is ' bfruse.i
end
exit
```

Chapter 3. NetView Pipelines Device Drivers 263
Partial (page 1) output from PIPSTEMA looks like this. Output continues until the contents of BFRUSE.54 are displayed.

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW</th>
<th>CNM01 OPER6</th>
<th>09/20/10 16:49:11</th>
</tr>
</thead>
<tbody>
<tr>
<td>C CNM01</td>
<td>16:49:11 PIPSTEMA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C CNM01</td>
<td>16:49:11 BFRUSE.0 IS IST097I DISPLAY ACCEPTED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C CNM01</td>
<td>16:49:11 BFRUSE.1 IS IST350I DISPLAY TYPE = BUFFER POOL DATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C CNM01</td>
<td>16:49:11 BFRUSE.3 IS IST920I 1000 BUFF SIZE 00567 EXP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW</th>
<th>CNM29 OPER6</th>
<th>03/01/10 19:16:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>* CNM29</td>
<td>PIPSTEMB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNM29</td>
<td>APPLES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNM29</td>
<td>PEARS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNM29</td>
<td>ORANGES</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
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, because 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 can 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. When the NetView command list family is exited, the storage occupied by the named SAFE is freed.

Messages in an unnamed SAFE cannot be processed by a different NetView command list. When 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 */
*************************************************************************/
/*
*************************************************************************/
/* 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 */
/* PIPSAFEA REXX COMMAND LIST */
/* THIS COMMAND LIST USES THE PIPE COMMAND AND THE SAFE STAGE */
*************************************************************************/
```
NetView Pipelines Device Drivers

/* 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 = ' RC
EXIT

Output from PIPSAFEA follows:

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW</th>
<th>CNM19 OPER6</th>
<th>02/01/10 18:00:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CNM19</td>
<td>PIPSAFEA</td>
<td></td>
</tr>
<tr>
<td>+ CMN19</td>
<td>DISPLAY</td>
<td>ACCEPTED</td>
<td></td>
</tr>
<tr>
<td>CMN19</td>
<td>DISPLAY TYPE = CDMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IST089I</td>
<td>A19CDRMS TYPE = CDRM SEGMENT , ACTIV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IST482I</td>
<td>A19M</td>
<td>ACTIV, SA</td>
<td>19, EL 1, NETID = NETA</td>
</tr>
<tr>
<td>IST482I</td>
<td>A09M</td>
<td>NEVAC, SA</td>
<td>9, EL 1, NETID = NETA</td>
</tr>
<tr>
<td>IST482I</td>
<td>A29M</td>
<td>NEVAC, SA</td>
<td>29, EL 1, NETID = NETA</td>
</tr>
<tr>
<td>IST482I</td>
<td>A69M</td>
<td>NEVAC, SA</td>
<td>69, EL 1, NETID = NETA</td>
</tr>
<tr>
<td>IST482I</td>
<td>A99M</td>
<td>NEVAC, SA</td>
<td>99, EL 1, NETID = NETA</td>
</tr>
<tr>
<td>IST482I</td>
<td>B18M</td>
<td>NEVAC, SA</td>
<td>N/A, EL N/A, NETID = NETB</td>
</tr>
<tr>
<td>IST482I</td>
<td>B20M</td>
<td>NEVAC, SA</td>
<td>N/A, EL N/A, NETID = NETB</td>
</tr>
<tr>
<td>IST482I</td>
<td>B24M</td>
<td>NEVAC, SA</td>
<td>N/A, EL N/A, NETID = NETB</td>
</tr>
<tr>
<td>IST482I</td>
<td>B52M</td>
<td>NEVAC, SA</td>
<td>N/A, EL N/A, NETID = NETB</td>
</tr>
<tr>
<td>IST482I</td>
<td>C01M</td>
<td>NEVAC, SA</td>
<td>N/A, EL N/A, NETID = NETC</td>
</tr>
<tr>
<td>IST482I</td>
<td>C02M</td>
<td>NEVAC, SA</td>
<td>N/A, EL N/A, NETID = NETC</td>
</tr>
<tr>
<td>IST482I</td>
<td>C11M</td>
<td>NEVAC, SA</td>
<td>N/A, EL N/A, NETID = NETC</td>
</tr>
<tr>
<td>IST482I</td>
<td>D09M</td>
<td>NEVAC, SA</td>
<td>N/A, EL N/A, NETID = NETD</td>
</tr>
<tr>
<td>IST314I</td>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. If the 10 second duration is exceeded, no more messages are 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:
1. The SAFE stage reads the messages, which is one MLWTO, from the queue named HOLDMSG and writes them to the pipeline.
2. The SEPARATE stage breaks the MLWTO into single-line messages.
3. The LOCATE stage selects messages containing the text string 'ACTIV'.
4. The CONSOLE stage reads the messages from the pipeline and displays them on the screen.
Building Large PIPE Commands: INTERPR

The INTERPR stage enables the user to build and run pipeline stages from input command data. The commands are processed by the INTERPR stage as an inner pipeline in a PIPE-within-a-PIPE structure. In some command procedure environments, commands are limited to 240 characters. The INTERPR stage allows pipeline specifications to exceed these limitations.

Using the INTERPR Stage

Example: Building a Pipeline with INTERPR

This example shows how to use the INTERPR 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
&CONTROL ERR *
*******************************************************************
** THIS CLIST USES THE PIPE COMMAND AND INTERPR 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 INTERPR * +
   SAFE MYKEYS +
   CONSOLE
*******************************************************************
** WRITE RETURN CODE INFORMATION TO TERMINAL AND EXIT CLIST **
*******************************************************************
&WRITE RETURN CODE = &RETCODE
&EXIT
```

Output from PIPEEX10 follows:
**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 INTERPRRT stage reads the MLWTO, creates and invokes a pipeline specification, 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, see IBM Tivoli NetView for z/OS Programming: PL/I and C.

A filter is a stage 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 stages 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 is not used, the pipeline output looks similar to the following example:

```
NETVIEW Pipeline Filters
270
Programming: Pipes
```
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 is not used, the pipeline output looks similar to the following example:

Selecting Messages by Content: LOCATE, NLOCATE, TOSTRING

Several stages 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.

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, see IBM Tivoli NetView for z/OS Programming: 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:

\texttt{PIPE NETVIEW LIST STATUS=TASKS | LOCATE /OST/ | CONSOLE}

Output from the pipeline looks like this:

```
NCCF NETVIEW CMN01 OPER6 05/17/10 13:38:00
* CMN01 PIPE NETVIEW LIST STATUS=TASKS | LOCATE /OST/ | CONSOLE
- CMN01 TYPE: OST TASKID: OPER6 RESOURCE: A01A701 STATUS: ACTIVE
- CMN01 TYPE: OST TASKID: RESOURCE: A01A702 STATUS: NOT ACTIVE
- CMN01 TYPE: OST TASKID: RESOURCE: A01A703 STATUS: NOT ACTIVE
- CMN01 TYPE: OST TASKID: RESOURCE: A01A704 STATUS: NOT ACTIVE
- CMN01 TYPE: OST TASKID: RESOURCE: A01A705 STATUS: NOT ACTIVE
- CMN01 TYPE: OST TASKID: RESOURCE: A01A706 STATUS: NOT ACTIVE
- CMN01 TYPE: OST TASKID: RESOURCE: STATUS: NOT ACTIVE
- CMN01 TYPE: OST TASKID: RESOURCE: STATUS: NOT ACTIVE
- CMN01 TYPE: OST TASKID: RESOURCE: STATUS: NOT ACTIVE
- CMN01 TYPE: OST TASKID: RESOURCE: STATUS: NOT ACTIVE
- CMN01 TYPE: OST TASKID: RESOURCE: STATUS: NOT ACTIVE
- CMN01 TYPE: OST TASKID: AUTO1 RESOURCE: AUTO1 STATUS: ACTIVE
- CMN01 TYPE: OST TASKID: AUTO2 RESOURCE: AUTO2 STATUS: ACTIVE
- CMN01 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:

\texttt{PIPE NETVIEW LIST DEFAULTS | SEPARATE | LOCATE /LOG/ /DISP/ | CONSOLE}

Output from the pipeline looks similar to:

```
NCCF NETVIEW CMN01 OPER6 05/17/10 13:40:30
* CMN01 PIPE NETVIEW LIST DEFAULTS | SEPARATE | LOCATE /LOG/ /DISP/ | CONSOLE
' CMN01 DMN0541 DISPLAY DEFAULTS
' CMN01 SYSLOG: NO
' CMN01 NETLOG: YES
' CMN01 HCYLOG: YES
' CMN01 DISPLAY: YES
```
### NetView Pipeline Filters

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:

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

Output (first page) from the pipeline looks similar to:

```
NCCF  NETVIEW  CNM01 OPER6 04/14/10 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: DSIORVS TASKNAME: DSIORVS STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSILOG TASKNAME: DSILOG STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIORVS TASKNAME: DSIORVS STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSILOG TASKNAME: DSILOG STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIORVS TASKNAME: DSIORVS STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSILOG TASKNAME: DSILOG STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIORVS TASKNAME: DSIORVS STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSILOG TASKNAME: DSILOG STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIORVS TASKNAME: DSIORVS STATUS: ACTIVE
```

Output (second page) from the pipeline looks similar to:

```
NCCF  NETVIEW  CNM01 OPER6 04/14/10 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: DSIHREM TASKNAME: DSIHREM STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIMDS TASKNAME: DSIMDS STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIKREM TASKNAME: DSIKREM STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIUDST TASKNAME: DSIUDST STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIHREM TASKNAME: DSIHREM STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIMDS TASKNAME: DSIMDS STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIKREM TASKNAME: DSIKREM STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIUDST TASKNAME: DSIUDST STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIHREM TASKNAME: DSIHREM STATUS: ACTIVE
- CNM01 TYPE: OPT TASKID: DSIMDS TASKNAME: DSIMDS STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: OPER6 RESOURCE: AO1A702 STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: AUTO1 RESOURCE: AUTO1 STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: AUTO2 RESOURCE: AUTO2 STATUS: ACTIVE
- CNM01 TYPE: OST TASKID: DSILOG TASKNAME: DSILOG STATUS: ACTIVE
- CNM01 END OF STATUS DISPLAY
```

Processing Steps:

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 05/17/10 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 02/01/10 10:11:15
* CNM01 PIPE NETVIEW LIST 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 is changed to insert the SEPARATE stage prior to the NLOCATE stage, the pipeline results change. The SEPARATE stage breaks the output of the LIST command, an MLWTO, into single-line messages. NLOCATE reads its input stream, and examines 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 when a match is found, processing for this stage terminates along with any outstanding CORRWAIT. Any unprocessed records in the input stream are 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

**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 02/01/10 13:38:00
  * CNM01 PIPE NETVIEW D NET,CDRMS | CORRWAIT 60 | TOSTRING LAST 1.7
  /IST314I/ CONSOLE
  CNM01 IST097I DISPLAY ACCEPTED
  ' CNM01
  IST350I DISPLAY TYPE = CDRMS
  IST089I AGICDRM 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, which include an MLWTO to the output stream.

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 03/26/10 16:50:00
  * CNM01 DISP482I
  CNM01 IST097I DISPLAY ACCEPTED
  ' CNM01
  IST350I DISPLAY TYPE = CDRMS
  IST089I AGICDRM TYPE = CDRM SEGMENT , ACTIV
  IST482I A01M ACTIV, SA 1, EL 1, NETID = NETA
  CNM01 RC IS 0
```
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. 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 03/01/10 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, because of 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 03/20/10 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:
Processing steps:
1. The NETVIEW stage runs the MAJNODES command list. Usually, this displays active major nodes in a domain on your console. When the command list is issued within a pipe, the output messages are placed in the pipeline.
2. The CORRWAIT stage allows a 10-second wait for messages to be returned from VTAM.
3. Before the 10 seconds expire, the message DISPLAY NET MAJNODES returns to the pipeline and is passed to the next stage. The DROP stage reads its input stream and discards the message.
4. Again, before the 10 seconds expire, the second message DISPLAY ACCEPTED returns to the pipeline, is read, and discarded by the DROP stage.
5. Again, before the 10 seconds expire, the MLWTO is passed to DROP, and because DROP has reached its maximum discard count, the MLWTO is preserved in the pipeline.
6. The CONSOLE stage reads the messages and displays them.

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.
NetView Pipeline Filters

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 the NetView program to create, manage, and terminate full-screen applications for 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 a NetView system. 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 295.

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 15 on page 282 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 15 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 16 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 is displayed on a hardware screen, available to a full-screen automation program. For more information on virtual screens, see “Virtual OSIs” on page 284.

Figure 16. Browse Changeable Configuration Panel

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 GETCONF)',
'| VET NEXT ROWS',
'| CORRWAIT 60',
'| SEPARATE',
'| NOT TOSTRING /ALARM THRESHOLDS/',
'| TOSTRING NOINCL /HIT ENTER/',
'| CONSOLE'
```

Figure 15. Simple Full-Screen Automation Example: REPTALRM
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 16 on page 282 to the pipeline in the following multiline message:

   
   BNH150I ROWS/NEXT OUTPUT FOR ' ' RECEIVED FROM MDMCNFG
   N   E   T   V   I   E   W   NCF01 OPER1  11/16/10 11:55:13
   * BROWSE CHANGEABLE 7861 CONFIGURATION PARAMETERS *
   ID = NCP1      STATION = TERMA      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)
   ANTI STREAMING 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 17. 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 290. Also, see the VET command information in "Interacting with Virtual OSTs" on page 286.

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 18 on page 284 the selected lines are written to the CONSOLE.
8. The REXX EXEC terminates and the virtual screen running MDMCNFG is automatically detached.
Full-Screen Automation

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 285.

Note: Additional examples can be found in CNME2011 and CNMS1101. CNME2011 contains a SESSMGET example and CNMS1101 contains an NPDA example.

Virtual OSTs

The NetView program 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, see 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 285.

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.

Figure 18. REPTALRM Results

See Chapter 2, “Pipeline Stages and Syntax,” on page 19 for information on pipe stages.
Attaching and Detaching Virtual OSTs

The NetView program 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. See 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 can 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:
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 285, 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 can end the VOST before all VOST processing completes. You can 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 creates an independent VOST named VOST1 running NPDA. The second creates 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 290 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 that 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 290 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 is usually 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 is 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.

**Note:**

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 statements:

```
  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:

```
STATMON.DSS
HOST: HOST126

1 NCP/CA/LAN/PK    1     |
1 LINES            1     |
1 PUS/CLUSTERS    1     |
1 LOCAL MAJ NDS   1     |
5 LUS/TERMS       5     |
1 APPL MAJ NDS    1     |
49 APPLICATIONS   8     |
1 CDRM MAJ NDS    1     |
1 CDRMS           2     |
1 CDRSC MAJ NDS   1     |
603 CDRSCS        302    |
..815 TOTAL NODES .324    |

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 (|).

```
.....1 NCP/CA/LAN/PK | *1*| *2*| *3*| *4*
.....1 LINES |.....|.....|.....|.....|
.....1 PUS/CLUSTERS |.....|.....|.....|.....|
.....1 LOCAL MAJ NDS |.....|.....|.....|.....|
.....5 LUS/TERMS |.....|.....|.....|.....|
.....1 APPL MAJ NDS |.....|.....|.....|.....|
...49 APPLICATIONS |.....|.....|.....|.....|
.....1 CDRM MAJ NDS |.....|.....|.....|.....|
.....1 CDRMS |.....|.....|.....|.....|
.....1 CDRSC MAJ NDS |.....|.....|.....|.....|
...603 CDRSCS |.....|.....|.....|.....|
...815 TOTAL NODES |.....|.....|.....|.....|
```

The following REXX program written to interact with this panel requests a STATMON detail panel on NCP/CA/LAN/PK. An end user places 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 lines in your REXX program:

```
input. = '
input.5 = 'X'
input.0 = 5

PIPE (NAME REQNCP)
  STEM input.
  VET 1.1
```
Full-Screen Automation

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:** Consider the following items when coding your full-screen automation program:
- The /string/ coded on a VET command can be a pipeline specification.
- Before detaching a VOST, you might 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 is 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 contains '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 returns:

![Figure 19. Example Screen for VET CURRENT ROWS](image_url)

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 can 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 is 3.10 and 6 characters are in data. Fields can be contained in one row or can 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 can be included in the reserved area. Information contained in the reserved area is subject to change and must 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 must 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
text is displayed on the console:

|BNH150I  FIELDS/NEXT OUTPUT FOR 'NPDA' RECEIVED FROM NPDA
WRITE 24.8 BEEP
  22.1 FA IN % BNJ905I INVALID COMMAND ENTERED OR INCORRECT OPERANDS
  23.1 FA IN %
  23.80 FA IH % CMD=>
  24.8 FI IH % BAD COMMAND

Figure 20. 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 % CNMKNCCF
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 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 21 on page 295]** 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 might 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 21. Message and Full-Screen Returned to VET

Figure 22 on page 296 shows a partial dump from ATTACH DUMP NPDA:
Dump data is 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 24 on page 297 returns alert history information. Although HALERT can be entered as a command, the example output shown in Figure 23 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.

---

**Figure 22. Sample Partial DUMP of VOST Data**

Dump data is returned each time data is received or sent from the application running on the VOST.

**Figure 23. Alert History Automation Results**
Full-Screen Automation

/* HALERT: Retrieve Alert History */
'ATTACH NPDA' /* Output, including messages will be */ /* saved for future VET call */

'PIPE (NAME AHIST1 END +)', /* Start a pipe */
' VET NEXT ROWS', /* give update when it arrives, as image*/
CORRWAIT MOE 60', /* Wait 60 sec for first NPDA screen */
' NLOCATE 1.7 /BNH150I/', /* I KNOW what first screen looks like */
' CONSOLE', /* show badness */
' A: LOCATE 1.7 /DWO369I/', /* Separate time-out message (MOE) */
' VAR timeout', /* save time-out message */
' A:', /* other message? */
' VAR npdamsg' /* save first for test */

If symbol('timeout') = 'VAR' THEN /* time out? */
Signal TIMEOUT /* handle unexpected error */
If symbol('npdamsg') = 'VAR' THEN /* some NPDA error */
Exit 20 /* messages from NPDA already shown */

 assistants --- Down to business  """
/* Type a 'ALH' (Alerts History) in the command area and push enter. */
'VET /ALH/' /* Sending the 'ALH' and an enter key. */

Do UNTIL(thispage = lastpage | RC=0)

'PIPE (NAME AHIST1P END +)', /* start a big pipe */
' VET NEXT ROWS', /* give update when it arrives, as image*/
CORRWAIT MOE 60', /* Wait 60 sec for rnd trip to DST... */
' SC: LOCATE 1.7 /BNH150I/', /* expected screen with data... */
' SEPARATE', /* handle lines individually... */
' PG: DROP 4', /* drop header area... */
' DROP LAST 1', /* command line */
' MSG: DROP LAST 4', /* message area, hopefully blank */
' STRIP TRAILING', /* shorten line ending in blank, so we */
' LOCATE 1 //', /* can toss out blank lines */
' CONSOLE', /* Report data out to user */
' TAKE 1', /* AND use ONE new data line to trigger */
' NETV VET /FORWARD/', /* ...new data, then go to next screen */
' MSG:', /* Immed message area from NPDA */
' LOCATE 2.3 /BNJ/', /* any error message in inmmed area */
' CONSOLE', /* REPORT it, too. */
' VAR npdamsg', /* save for test */
' SC:', /* message instead ?? This is bad. */
' A: LOCATE 1.7 /DWO369I/', /* Separate time-out message (MOE) */
' VAR timeout', /* save time-out message */
' A:', /* other message ? ?? This is bad. */
' CONSOLE', /* show badness */
' VAR npdamsg', /* save first for test */
' PG:', /* get header area */
' DROP 2', /* drop BNH150 and NPDA head-date line */
' VAR pagedata' /* save pagedata for test */

If symbol('timeout') = 'VAR' THEN /* time out? is possible? */
Signal TIMEOUT /* handle unexpected error */
Parse var pagedata . 'PAGE' thispage . lastpage /* parse out page nos. */
End /* */
EXIT 0 /* And do automatic DETACH for NPDA's VOST */

Figure 24. Sample Full-Screen Automation Program — Capture Alert History

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 can 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, are handled by the simple pipe connected with the $SC$: 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 Tivoli NetView for z/OS SQL Stages for Access to DB2

This chapter describes how to use the Tivoli NetView for z/OS 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 Tivoli NetView for z/OS 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)

Tivoli NetView for z/OS SQL stages can use DB2 if the Tivoli NetView for z/OS task DSIDB2MT is started and is connected to a DB2 subsystem. Records are passed between DB2 and the Tivoli NetView for z/OS 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 Tivoli NetView for z/OS SQL stages, formats a query and converts data from internal representation to character strings:

```
SQSELECT * from inventory where substr(description,1,4)='S390'
```

<table>
<thead>
<tr>
<th>SUBAREA_NO</th>
<th>DESCRIPTION</th>
<th>JOBS_ACTIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+221</td>
<td>S390 Raleigh NC</td>
<td>+2000</td>
</tr>
<tr>
<td>+222</td>
<td>S390 Austin TX</td>
<td>+1250</td>
</tr>
</tbody>
</table>

The SQL Stage

The SQL stage connects the Tivoli NetView for z/OS program to the relational database product IBM DATABASE 2 (DB2) for MVS. Potential users of SQL include the following:

- Novice users of DB2 and Tivoli NetView for z/OS pipelines:
  - SQSELECT is a Tivoli NetView for z/OS REXX procedure to format a query for display.
  - Input lines read by a Tivoli NetView for z/OS 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.

- Tivoli NetView for z/OS SQL stage users who are new to DB2. Advanced DB2 education is outside the scope of this book. See 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 Tivoli NetView for z/OS pipelines eliminates many of the programming details, such as DB2 precompilation, reentrant considerations, and the details of memory management.

Defining DB2 to the Tivoli NetView for z/OS program

- The DSIDB2DF member of DSIPARM is read when task DSIDB2MT starts, and defines the DB2 subsystem (SUBSYSTEM=DB2) to which the Tivoli NetView for z/OS system connects. It uses the following definition statement:
Using Tivoli NetView for z/OS SQL Stages for Access to DB2

SUBSYSTEM=DB2
The SUBSYSTEM definition identifies which DB2 is the Tivoli NetView for z/OS default. The DB2ACCESS definition sets which DB2 interface is used for the Tivoli NetView for z/OS program unless the DB2SEC statement in the CNMSTYLE member 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 Tivoli NetView for z/OS program is running.

- The Tivoli NetView for z/OS SQL stage must be defined to DB2. The Tivoli NetView for z/OS SQL stage is defined once in a process called binding the plan. Sample CNMSJSQL 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:

```bash
//SYSUT2 DD DSN=USER2.DBRMLIB(DSISQLDO),DISP=SHR

The following sample shows the BIND statement:
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,DB2L01.DSISQL14.DSISQLDP) ISOLATION(CS) OWNER(USER2)
```

DSISQL04, DSISQLDO and DSISQLDP are Tivoli NetView for z/OS names that do not change. Change IBMUSER to a value suitable to your installation to identify the Tivoli NetView for z/OS system that is using SQL. In general, the Tivoli NetView for z/OS plan name is DSISQLnn where nn is changed because of service or future releases. The CNMSJSQL sample is reshipped when a change to the DSISQLDO and DSISQLDP programs causes the plan to be incompatible.

To run CNMSJSQL, the job user ID must have BINDADD privilege.

- The Tivoli NetView for z/OS 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 Tivoli NetView for z/OS SQL stages. Unload the plan from the system where you have installed Tivoli NetView for z/OS SQL stages and bind it to the other systems.

- You must be registered as a 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 statement in the CNMSTYLE member.

If DB2SEC is set to RRS, the Tivoli NetView for z/OS 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 Tivoli NetView for z/OS program loads the CAF interfaces. You do not have operator level security, but you can continue to 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 remain active for tasks that access the DB2 subsystem that DSIDB2MT accesses. When starting the
DSIDB2MT task, ensure any SQL requests that do not specify which DB2 to access are accessing 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 Tivoli NetView for z/OS address space. This task connects the Tivoli NetView for z/OS system to a specific DB2 subsystem so that any task in the Tivoli NetView for z/OS 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 might 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:

```sql
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

| 404 | 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 |
```

Chapter 6. Using Tivoli NetView for z/OS SQL Stages for Access to DB2  301
In this example,SSID DB2 defines the DB2 subsystem named DB2 as the subsystem this task references.

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:

```
pipe SQL execute create table jtest (kwd char(8), text VARCHAR(80))
    in netview.xmydb cons
'pipe /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 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:

```
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 access that DB2 unless the keyword SSID is used.

The following is an example of an SQL SELECT query:

```
pipe SQL select * from jtest | console
""CCC"" ""TIVOLI NetView for OS/390 Automation"
""BBB"" ""Network Program"
""AAA"" ""Automated Banking System"
```

The double quotation marks 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 might 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 an easy-to-read, printable format with column headings.

**SQSELECT:**

```
SQSELECT ( SQL options ) DB2 SELECT operands
```

Note the following when using SQSELECT:

- 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 SSIDssidname 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 displays 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 Tivoli NetView for z/OS 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 can fail if it tries to change the subsystem name, which is the SSID 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.

```
PIPE SQL COMMIT WORK | CONS
```

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 that 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 might 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 Tivoli NetView for z/OS message type double quotation mark (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 Tivoli NetView for z/OS 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 Tivoli NetView for z/OS 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.

If one stage fails with a DB2 error, the unit of work is rolled back. All subsequent SQL stages in the pipeline ignore all input. Messages indicating the failure are issued. NOCLOSE is 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 a Tivoli NetView for z/OS system, 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 308 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 152.

**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.

See *IBM Tivoli NetView for z/OS Command Reference Volume 2 (O-Z)* 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, see 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.

See *IBM Tivoli NetView for z/OS Command Reference Volume 2 (O-Z)* 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 | CONSOOLE
```

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 DW0362E 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                  NETVIE W      CNM01 OPER6 03/28/10 12:15
* CNM01 PIPE NETVIEW CLOSE IMMED | CONSOOLE
- CNM01 DW0364E PIPELINE TERMINATED. NO STAGE CONSOOLE EXISTS.
- CNM01 DW0362E PIPELINE TERMINATED. ERROR IN STAGE 2 IN PIPELINE 'PIPE'.
```

Note: Stage names inserted with the INTERPRT stage 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 131 for more information.

Online Help

To display online help for the pipe command, enter:

```
HELP PIPE
```

To display online help for a specific stage, enter:

```
HELP PIPE stage_name
```

Where:

```
stage_name
```

is any NetView PIPE stage.

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"*/
  ' CONS0LE',          /* 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 DROP holds the line again. 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
/* REXX Fragment */

... PIPE (NAME CLOGGER END ")",
  '< somemember        /* generate data */
  ' A: DROP LAST 2',   /* last two lines goto "A" */
  ' FANIN',            /* could also be FANINANY */
  '\ LABS:'           /* give lines to secondary of COLLECT*/
```
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.

The DEBUG 2 option helps you to diagnose a clogged pipeline. For more information about DEBUG 2, see “Clogged Pipeline Details” on page 316.

Tracing Pipelines

When the pipeline syntax is correct and the pipeline runs, you might 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 can 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 might 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.
Debugging NetView Pipelines

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 */
|, /* --- END of simple pipeline, begin new pipeline ... */
| - 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 ".
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 ".

314  Programming: Pipes
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.
- 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 works:

  ```
  PIPE () (DEBUG) NETV LIST X|...
  PIPE | (DEBUG) NETV LIST X|...
  ```

- For stages 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:

```plaintext
PIPE (NAME SAMP DEBUG 1)
   | < MYFILE
   | STRIP TRAILING / /
   | JOINCONT TRAILING /$/
   | CONSOLE
```

Is the same as:

```plaintext
PIPE (NAME SAMP)
   | (DEBUG) < MYFILE
   | (DEBUG) STRIP TRAILING / /
   | (DEBUG) JOINCONT TRAILING /$/
   | (DEBUG) CONSOLE
```
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)',
  '  \< somelement /* 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.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Message</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP</td>
<td>(1) waiting to output to DROP</td>
<td>(2) on stream 1</td>
</tr>
<tr>
<td>COLLECT</td>
<td>(3) awaits input from DROP</td>
<td>(2) on stream 1</td>
</tr>
<tr>
<td>CONSOLE</td>
<td>(4) awaits input from COLLECT</td>
<td>(3) on stream 1</td>
</tr>
</tbody>
</table>

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. These 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 Troubleshooting 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 might 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 stages 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 might be a tape mount request message, IEF290E, which can be automated to display at the NetView operator's terminal. Only the first few lines might 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.

```
Figure 25. 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 first 2 lines, while discarding other messages from the pipeline. The last stage, CONSOLE, displays the pipeline contents to the operator's console.
```
Additional NetView Pipeline Examples

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 26. 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 de-emphasized 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 a 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 EXECS END */
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. AAAAAAAAAAAA BBBBBBBBBBBBBBB ';
   'CCCCCCCCCC DDDDDDDDDDDDD EEEEEEEEEE FFFFFFFFFFFF ';
   'Gggggggggggggggggggggg HHHHHHHHHHHH IEEEEEEEEEEE);
   'IIIJJJJJJJJJJJJJJ JJ KK KKKKKKKKKKK LLLLLLLLLLLL MM'|
   'NNNNNNNNNNNNN XXXXXXXXXXXXXXXXXXXXX 00000000000000 PP PPPPPP'|
   'PPPPPPP QQQQQQQQQQQQQQ'|
**************************************************************************
*/ READ THE VARIABLE INTO THE PIPELINE. */
*/ EXCMD TO OPER6 BOTH THE NAME OF THE CLIST */
*/ EXECRECV AND THE PIPELINE CONTENTS */
**************************************************************************

'PIPE VAR LONGVAR',
' | NETVIEW EXCMD OPER6,EXECRECV'
```
IF RC =\ 0 THEN
  SAY 'RC=\ RC ' FROM PIPE '

IF RC = 0 THEN
  SAY 'TRANSFER SENT SUCCESSFULLY'
EXIT

The results displayed on the terminal of OPER3, the sender, follows:

<table>
<thead>
<tr>
<th>ADDRESS NETVASIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNM19 EXECSEND</td>
</tr>
</tbody>
</table>

Figure 27. Transfer 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:

<table>
<thead>
<tr>
<th>ADDRESS NETVASIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNM19 EXECSEND</td>
</tr>
</tbody>
</table>

Figure 28. Transfer Received Results Screen

Note: The data can also be sent to a task in a remote NetView system. See "PIPE
CORRCMD" on page 46.
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:

/* 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
  DO
    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
/***********************************************/
/* SEPArate DISPMOD MLWTO output into single lines, */
/* LOCate APARs/PTFs supplied as arguments */
/* COLlect the matched lines into a single MLWTO */
/* Display to the CONSoile ONLY */
/* (do not put into NetView log or automate) */
/***********************************************/
'PIPE NETV DISPMOD ALL ALL',
  'SEP',
  'LOC ' srchtxt,
  'COL',
  'CONS ONLY'
EXIT

Figure 29. 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:

/***********************************************/
/* 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 */
  /* 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. */
Additional NetView Pipeline Examples

'PIPE SAFE GDSSAFE',
  '  COL',
  '  CONS CLEAR ONLY'
EXIT

Output from the pipeline (DSIGDS in active status) follows:

<table>
<thead>
<tr>
<th>NCCF</th>
<th>NETVIEW</th>
<th>CNM01 OPERI</th>
<th>05/17/10 12:20:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE: OPT TASKID: DSIGDS TASKNAME: DSIGDS STATUS: ACTIVE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IST271I JOBNAME = MYENV, STEPNAME = MYENV, DSPNAME = 000091ST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSOLICITED DSRBS: 1 USED: 0 FREE: 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOLICITED DSRBS: 5 USED: 0 VSAM REDRIVE: 0 FREE: 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL DSRBS: 6 USED: 0 VSAM REDRIVE: 0 FREE: 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TASKNAME TYPE DPR CPU-TIME N-CPU% S-CPU% MESSAGEQ STORAGE-K CMDDLIST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----- ---- ---- ---- ---- ---- ---- ----</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSIGDS DST 254 0.01 0.00 0.00 0 35 N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 30. 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 list 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 CNM01CNFT sample supplied with the NetView product for more information.

Note: APAR OY52941 must be applied for the JES commands in this command list to have correlated output.

/* Sample REXX command list Named: JES2JOB */
/* Syntax: JES2JOB parm1 parm2 */
/* where: */
/* parm1 is one of the following: */
/*  jobname {A JOBNAME} */
/*  job* {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) */

/* Get input jobname parm and */
ARG jobname arg2
/* display/cancel parm. */
IF (jobname = '') | (jobname = '?') THEN
DO
  SAY 'Enter BR JES2JOB for help and syntax'
EXIT
Additional NetView Pipeline Examples

END
/*********************/
/* Use parml to create a joblist */
/*********************/
i = LENGTH(jobname)
IF SUBSTR(jobname,i,1) = '*' THEN /* If using wildcard */
   DO
      jobname = STRIP(jobname,T,'*') /* Remove '*' from jobname.*/
i = i - 1 /* Decrement removed '*' from leng.*/
   END
ELSE
   DO
      jobname = jobname || '' /* Add '' to jobname so we do not */
i = i +1 /* match other jobs that start with */
   END /* the characters of our jobname. */
/*********************/
/* Issue MVS $DA command on the */
/* NETVIEW stage to get info about */
/* active batch jobs and jobs */
/* waiting for resources. The cmd */
"PIPE NETV MVS $DA,ALL,L=Z',
   CORR 30',
   TAKE 1',
   SEP',
   DROP FIRST 1',
   LOC 10.'i+10' \'jobname\',
   STEM jobline.',/* Store matching lines in */
   */
   */
   */
   */
   */
   */
   /* Issue MVS $DN command on the */
   /* NETVIEW stage to get info about */
   /* jobs waiting for execution. The */
   "PIPE NETV MVS $DN,ALL,L=Z',
   CORR 30',
   TAKE 1',
   SEP',
   DROP FIRST 1',
   DROP LAST 2',
   LOC 10.'i+10' \'jobname\',
   STEM jobline. APPEND'
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 */
/*********************/
="/***********************************/
/* Use STEM stage to read matched */
"PIPE STEM jobline.',
   COL',
   CONS ONLY'
   */
   */
   */
   */
   */
   /* Gather all output into 1 MLWTO. */
   /* Display. Do not automate or log.*/
/*********************/
/*********************/
/*********************/
/*********************/
Additional NetView Pipeline Examples

/* Uncomment the following section to display more detailed */
/* information on each job. */
**************************

/*
DO j = 1 TO jobline.0  /* Go thru matched job info.*/
  jobnum = SUBSTR(jobline.j,4,5)  /* Get job number. */
  /
'PIPE NETV MVS $D$jobnum','LONG',  /* Put JES display in pipe. */
  'CORR 30',  /* Wait for JES response. */
  'TAKE 1',  /* Terminate wait when */
  /*
  'SEP',  /* Split up lines of MLWTO */
  '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 $C$jobnum','P',  /* Cancel job, purge output */
  'CORR 30',  /* Wait for JES response. */
  'TAKE 1',  /* Terminate wait when */
  /*
  'SAFE cmdsafe APPEND'  /* Store for later display. */
END
'PIPE SAFE cmdsafe',  /* Display output from JES */
'CONS ONLY'  /* cancel or display cmds. */
/* Do not automate or log. */
END

GETOUT:  /* Exit. */
EXIT

Output from the JES2JOB display command list follows:

Figure 31. JES2JOB Display Command Output Example

Output from the JES2JOB cancel command list follows:
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be
incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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

IBM Corporation
2Z4A/101
11400 Burnet Road
Austin, TX 78758
U.S.A.

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

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM’s application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:
Programming Interfaces

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

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at http://www.ibm.com/legal/copytrade.shtml

Adobe is a trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft 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.

Other product and service names might be trademarks of IBM or other companies.

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, See IBM’s Privacy Policy at http://www.ibm.com/privacy and
Index

Special characters
$STEM stage 202
  saving records processed counts, example 205
writing, stemmed variables, example 205
$VAR stage 222
* (asterisk)
  LOGTO option 144
  SAFE option 186
> (To Disk) stage command 248
< (From Disk) stage command 245
  reading file contents into pipelines, example 247

A
accessibility xiii
action key, VET option 232, 288
ALL, LOGTO option 144
APAR (authorized program analysis report) 322
APPEND
  $STEM option 203
  SAFE option 186
  STEM option 203
APPEND stage 26
asynchronous messages
  converting, synchronous 154
ATTACH command 285
authorized program analysis report (APAR) 322

B
BETWEEN stage 28
BNH150I
  FIELDS format 291
  ROWS format 291
books
  see publications ix

C
CANZLOG, LOGTO option 144
CART (command and response token) 154
CART (stage and response token) 43
CASEI stage 30
CC stage (CORRCMD synonym) 46
CCDEF 47, 50
CHANGE stage 31
CHOP stage 34
CLEAR, CONSOLE option 42
clearing screens, CONSOLE CLEAR option 42
clogged pipelines 311
CNMS1098 sample 295
CNMS1101 23
COLLECT option
  $STEM command 203
  STEM command 203
COLLECT stage 36
  avoiding misleading output from CONSOLE 313
  collecting single-line messages 269
command and response token (CART) 154
compare and swap 225
complex pipelines 7
complex pipelines, creating 7
CONSOLE stage 41
  avoiding misleading output 313
taking snapshots, pipeline messages 313
task interaction 249
conventions
typeface xiv
COREVENT stage 44
COREVTDA stage 45
CORRCMD stage 46
correlated output, HOLE stage 128
correlation method for converting asynchronous messages 154
CORRWAIT secondary output 50
CORRWAIT stage 49
COUNT stage 54
CPDOMAIN stage 57
CURRENT, VET option 232, 287
CZR stage 59
  CZR 59

D
data streams
  connected 5
  defined 5
  disconnected 5
  primary stream 4
  secondary stream 4
tertiary stream 4
DB2 299
deadlock 311
DEBUG 20
DEBUG parameter of PIPE 20
documentation
  primary stream 4
tertiary stream 4
tedium stream 4
default SAFE 185
DELDUPES stage 61
DELETE, CONSOLE option 42
deleting operator message (DOM) 42
delimiters, usage 22
description 129, 131, 134, 140, 141, 143, 156, 166, 185, 188,
  190, 192, 202, 209, 210, 218, 229, 230
detachment command 286
device drivers, placement dependency 249
directory names, notation xv
diskonly, < (from disk) option 246
displaying
  pipeline messages, CONSOLE stage 249
divert stage 63
dom (delete operator message) 42
double-byte character set data 304
DROP stage 64
  example, discarding messages 65
examples (continued)
displaying translated messages 158
dividing text 194, 195
DROP stage 65
DUPLICAT stage 66
EDIT stage 113
ending a wait, TOSTRING 211
ending, INSTORE stage 131
ENVDATA stage, environment data 115
ENVDATA stage, genealogy data 115
error messages 133
executing commands, DSPARM 221
generating return codes, NETVIEW stage 257
generating return codes, VTAM stage 259
handling asynchronous messages 53
HELDMSG stage 127
hiding members, procedure family 131
HOLE stage 128
IDLEOFF 148
inserting and logging text strings 144
inserting multiple text strings 254
inserting text strings, pipelines 141
inserting text, embedded command list function 254
INSTORE stage 131
INTERPRT stage 132, 133
issuing commands 155
issuing MVS commands 256
issuing VTAM commands 259
VTAM commands 237
keeping first messages, pipeline 277
keeping last messages, pipeline 278
keeping messages, multiple text strings 272
keeping messages, specified text strings 272
listing current UNIX working directory 221
LITERAL stage 141
loading members, disk into storage 131
LOCATE stage 143, 155, 247
locating messages by content 165
locating messages, content 143
LOGTO stage 144
LOOKUP stage 147, 148
managing load members (INSTORE) 131
MLWTO processing with NLocATE 274
multiple CONSOLE stages 251
multiple CONSOLE stages, COLLECT stage 252
named SAFE as message queue 187
NLocATE stage 157
NLS stage 158
NOT stage 159
passing messages, second PIPE command 187
PICK stage 165
PIPEND stage 167
processing MLWTOs, TOSTRING 275
processing pipeline data 257
processing pipeline messages 258
processing single-line messages, TOSTRING 276
QSAM stage 179
reading and writing messages, SAFE 265
reading data from DASD 261
reading file contents into pipelines 247
reading variables, STEM 264
reading, DSICLD ddname data sets 261
receiving messages from XCF groups 239
reset line counts 57
retrieving the state field 242, 243
REVERSE stage 181
routing held messages 252

E

EDIT stage 67
education
   see Tivoli technical training xiii
end character, pipeline 9
END parameter, PIPE 19
ENVDA stage 114
environment variables, notation xv
ESC parameter, PIPE 19
examining return codes, CORRCMD stage 47
example, NPDA automation 296
examples
   $STEM stage 205
   alert to Tivoli Event Console 113
   automation table sample 218
   BETWEEN stage 29
   breaking multilime messages 189
   building pipeline specifications 132
   building pipelines, INTERPRT 267
   CASEI stage 30
   causing a wait 53
   CHANGE stage 33
   changing separation character 23
   CHOP stage 35
   CNMS1101 23
   COLLECT stage 39, 247
   collecting single-line messages into multiline message 270
   comparing and setting variables 227
   comparing values contained in two stems 147
   compiling and executing a Java sample 222
   CONSOLE stage 43
   converting ASCII text 244
   converting single-line messages to multiline 39
   copying task globals 227
   CORRWAIT stage 53
   COUNT stage 56, 57
   counting comment lines in files 247
   counting messages 56
   CPDOMAIN stage 59
   creating a command 113
   creating four consecutive utilization reports 66
   creating named SAFEs, NULL messages 187
   DELDUPES stage 63
   deleting held messages 127, 250, 253
   determining, command response correlated to command 279
   determining, named SAFE 186
   discarding
      messages 65
      messages by content 157
      messages containing specified text strings 273
      pipeline contents 128
   discarding first messages 278
   discover ISO stacks serving a user 216
   display last logtime 63
   displaying messages in dump format 43
   displaying messages on consoles 43
   displaying part of automated message 319
   displaying results and avoiding logging 249

332  Programming: Pipes
examples (continued)
running large pipelines 133
running VTAM commands, remote domains 260
SAFE stage 186, 187
saving records processed counts 205
searching, APARs and PTFs 322
selecting last message 210
selecting single-line messages, multiline messages 190
selecting, words 113
sending 241
sending messages to XCF groups 239
sending system log messages 135
SEPARATE stage 186, 187
separating records processed counts 205
searching, APARs and PTFs 322
selecting last message 210
selecting single-line messages, multiline messages 190
separating variables 227
SORT stage 192
sorting 192
SPLIT stage 194, 195
splitting text 194, 195
STAGESEP for DBCS problems 225
STEM stage 205, 225, 245, 247
TAKE stage 210
terminating a CORRWAIT 53
TOSTRING stage 211
transferring large variables 320
translating an ASCII value to EBCDIC 245
TSO stage 216
TSROUTE stage 218
UNIX stage 221, 222
unloading member, storage 131
update current group members 228
using XCFMSG 239
using XCFQUERY 241
using XFCTABLE 242, 243
using XLAITE COMMON 244
VARLOAD stage 227, 228
waiting five seconds 128
waiting for messages 53
writing null messages, STEM 264
writing, named variables 225
writing, stemmed variables 205, 263
EXPOSE stage 115
exposed message 42
full-screen automation (continued)
VOST
attaching a VOST 285
detaching a VOST 286
VOST, definition 284
G
graphic character strings 304
H
HCYLOG, LOGTO option 144
held messages, HELDMSG 126
HELDMSG stage
dexample, deleting held messages 127
task interaction 249
hold status, reversing 249
HOLE stage 127
IBM DATABASE 2, MVS 299
INCL, < (from disk) option 246
input strings, literals 19
inserting, pipelines 140
INSTORE stage 129
dexample, INSTORE stage, example 131
hiding members, procedure family, example 131
loading members, disk into storage, example 131
INTERPR stage 131
building pipeline specifications, example 132
error messages 132
managing load members, example 131
running large pipelines, example 133
IPLOG stage 134
J
JOINCONT stage 135
K
KEEP stage 137
L
label parameter, PIPE 19
label, pipeline 7, 9
limitations
controlling messages, multiple CONSOLE stages 43
delimiters 22
held messages, autotask 127
length restrictions, handling 131, 134
messages deleted, displaying 43
search strings 22
terminating CORRWAIT 53
literal 19, 140
LITERAL stage 140
inserting text strings, pipelines, example 141
task interaction 249
LOCATE stage 141
issuing commands, example 155
locating messages, content, example 143, 271
Index 335

PPI stage 168
PRESA TTR stage 173
changing color of selected data, example 175
program temporary fix (PTF) 322
PTF (program temporary fix) searching, PIPE command 322
publications accessing online xii
NetView for z/OS ix ordering xii
Q
QSAM stage 176
R
restrictions controlling messages, multiple CONSOLE stages 43
delimiters 22
held messages, autotask 127
length restrictions, handling 131, 134
messages deleted, displaying 43
search strings 22
terminating CORRWAIT 53
REVERSE stage 180
REVISRPT stage 182
ROUTE stage 182
ROWS, VET option 233, 287
S
SAFE default 185
named 185
SAFE stage 185
creating named SAFEs, NULL messages 187
determining, named SAFE, example 186
passing messages, REXX command list 187
passing messages, second PIPE command 187
taking snapshots, pipeline messages 313
using named SAFE, example 187
search strings, usage 22
selecting messages, position, TOSTRING 271
SEPARATE stage 188
breaking multiline messages, example 189
selecting single-line messages, multiline messages, example 190
separating data lines, example 190
separating multiline messages 269
separation character, changing 23
SEQCHOP, < (from disk) option 246
service xiii
service management connect xiii
setting timeout value, asynchronous messages 49
single-line messages 39
SMC xiii
snapshots, pipeline messages 313
SORT stage 190
sorting messages 191
sorting, example 192
SPLIT stage 192
dividing text 193
splitting, examples 194, 195
SQL 299
accessing DB2 299
SQL (continued)
concurrent SQL stages 305
creating, loading, and querying tables 302
defining unit of work 304
diagnosing output 306
querying databases and formatting results 303
SQL stage 195
SQLCODES stage 201
SQSELECT - Format a query 301
stage and response token (CART) 43
stage description 36, 41, 64
stages 36
$STEM 202
$VAR 222
> (To Disk) stage command 248
< (From Disk) 245
APPEND 26
BETWEEN 28
CASEI 30
CC 45, 46
CHANGE 31
CHOP 34
COLLECT 36
COLOR 173
CONSOLE 41
COREVENT 44
COREVTDA 45
CORRCMD 46
CORRWAIT 49
COUNT 54
CPDOMAIN 57
DELDUPES 61
DIVERT 63
DROP 64
DUPLICAT 66
EDIT 67
ENVDATA 114
EXPOSE 115
FANIN 117
FANINANY 118
FANOUT 120
FMTPACKT 121
HELDMSG 126
HOLE 127
INSTORE 129
INTERPR 131
IPLOG 134
JOINCONT 135
KEEP 137
LITERAL 140
LOCATE 141
LOGTO 143
LOOKUP 144
MEMLIST 148
MVS 150
NETVIEW 152
NLOCATE 156
NLS 157
NOT 159
NPDAEVD 160
PERSIST 161
PICK 164
PIPEND 166
PPI 168
PRESA TTR 173
QSAM 176
REVERSE 180
Programming: Pipes